Distributed databases and Apache Hive

Danilo Acosta Villalobos[2] and Ricardo Rojas Ruiz[3]

[1] Université Libre de Bruxelles, Avenue Franklin Roosevelt 50, 1050 Bruxelles
[2] danilo.acosta.villalobos@ulb.be
[3] ricardo.rojas.ruiz@ulb.be

Abstract. This document summarizes the principal characteristics of distributed databases; including its strengths, challenges, and design principles. Moreover, Apache Hive is used to explained a distributed system and a brief comparison against traditional SQL is done, showing the architectural challenges Apache Hive faces.

Keywords: Databases · Distributed Databases · Apache Hive · Apache Hadoop.

1 Background

1.1 From centralized to distributed databases

Databases technologies were developed due to a data integration necessity for applications, when local storage started being a restriction and non-viable. The earliest versions of this databases were developed under the centralized schema, considering the limitations regarding hardware and technology. Therefore, centralization is an ease on the implementation and a physical restriction, but not a principle of databases design. A database technology objective is the integration of data, not the centralization of it [1].

Nowadays, computers are highly accessible, networks support multiple types of technologies and heavy loads within a small time-frame. Moreover, databases started to face challenges such as volume and velocity where an individual processing entity have not been able to cope with, and, increasing the capabilities beyond the needs, is no longer an option by economic and hardware restrictions.

Considering the needs of the new data world and still following the integration principle, distributed databases are born as an alternative to face those challenges. Resources scalability stopped being a problem of single unit power and became a challenge of a divide and conquer paradigm where most of the challenges reside in the logic.

Even though the main centralized databases restrictions were overcome by distributed databases, a trade off was expected. The abstraction required to keep a similar interaction to centralized databases puts lots of effort on implementation and development tasks; reaching new challenges on networks, security, and general transparency of data.

1.2 Role of distributed databases in cloud based systems

Cloud based systems have gained strength over the last years due to the flexibility and accessibility. Developing and deploying applications to the cloud is available to everyone. Platforms such as Amazon Web Services or Google Cloud Computing are just some examples where any user can deploy and scale its system.

Still, considering the architecture design of the cloud and its paradigm, distributed databases have played an important role.

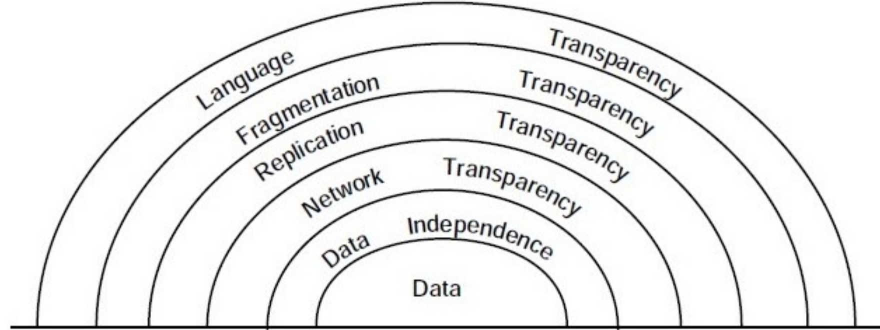
2 Distributed databases

Even tough distributed databases keep the basic functionality of any type of database, they face concerns specific of their implementation and most of the differences relay on the following promises:

- Data independence: On a distributed database, the data may be fragmented into several parts, replicated and distributed. This means that data logic structure should not affect the data retrieval (logic independence) and the storage structure neither should affect the upper layers (physical independence).
- Transparency: There exists a separation of the implementation layers. The goal is that a query should not care about fragmentation, replication or location of the data. Even though, there exists several levels of transparency that affects the upper levels and the queries:
  - Network transparency: This layer is entirely a responsibility of the Distributed Database Management System(DDBMS), so it should be shielded the same way the data is. This means location or object naming should be careless to the user as it should work the same way as a centralized database works.
  - Replication transparency: Due to performance, reliability and availability, it is usual to replicate data among nodes, but this means managing the replication without the user knowing the data has copies.
  - Fragmentation transparency: The network also needs to translate a global query to a fragment query. Within this scenario a horizontal fragmentation splits the data by sub-relations such as tuples, and a vertical fragmentation splits by attributes, such as columns; but the user only cares about the conceptual model of the data.

As the figure 1 shows, data is covered by abstraction layers that cover all the promises on a DDBMS. The closer a promise is to the data, the more important the abstraction is, as the implementation is more related to the distributed schema and worthless to expose to the user. The outer the promise, the more valuable it can be to the user and the less the DDBMS is willing to take care of it. As an example, a user can take care of the network, but knowing the internal network and all the protocols is more work to the user and almost no value

Fig. 1. Promises of a Distributed Database Management System.[1]

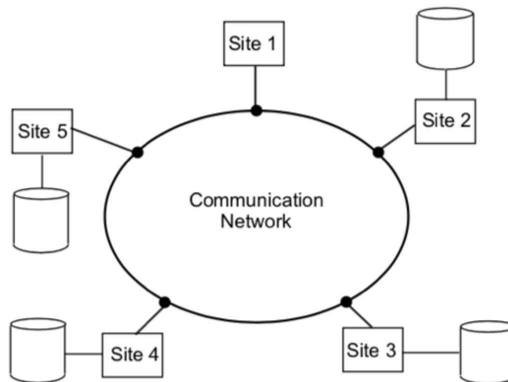


shown. On the other hand, a user may want to specify how fragmentation and replication will work, as the user know the environment, the user can optimize how many replicas, where should the replicas be stored and even how to fragment the data; as it may be geographically closer, for example.

To summarize this section, we will provide the formal definition of defining two critical terms, along with a graphic representation of the typical topology of a distributed database.

– Distributed database: collection of multiple, logically interrelated databases distributed over a computer network.[1]
– Distributed database management system: software system that permits the management of the distributed database and makes the distribution transparent to the users.[1]

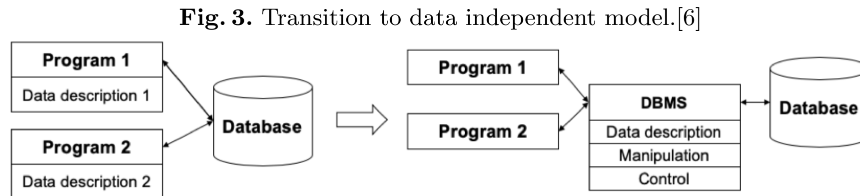
Fig. 2. Distributed database topology example.[1]



2.1 Data independence

Data independence is the concept responsible for the creation of DBMSS. Before, each application had to understand and manage their own data, then, store it in a safe repository. This did not only mean that application development was more complex, but it limited resource sharing because of possible heterogeneous physical and logical structures. In order to relieve applications from the burden of data management and to make all data standardized (at least in a single site), centralized DBMS were created. In this new model, even if the physical or logical structure of the data is modified in a DBMS, all applications consuming its data will continue to function properly without the need to perform modifications.

Fig. 3. Transition to data independent model.[6]



Centralized DBMS have inherently the same physical structure for all data it contains, and the logical structure (schemas and mappings, among other information) is stored in a data directory. However, on a DDBMS different sites may be both physically and logically diverse. Physical structure is of no relevance to us, since each DBMS is able to manage it independently, but there should be a logical structure that pertains to the whole DDBMS; the global directory. The global directory must have the schema and mappings for information on all of its sites. This directory can be centrally stored in one site or distributed over all sites, each with its own drawbacks. Centrally stored directories will have to deal a bottle-neck of access to them, while looking for data on a distributed directory would be inefficient. This global directory could also be replicated throughout sites, but then maintaining all copies up to date would be more difficult.

2.2 Replication transparency

Distributed databases are usually replicated because of the many advantages related to it. First off, since data is replicated, if a one or several sites are down, data can be accessed in other sites, thus improving system availability. Second, since communication costs should now be considered during query optimization, placing data closer to the sites that will require it will significantly improve the DDBMS performance. Finally, since systems tend to grow in terms of sites (eg. a company opening a new operations hub) and transactions, data replication allows to maintain performance standards regarding response times.

As we can see, replication of data has several benefits, but keeping all copies up to date and consistent can be a challenge. The DBMS should be capable

of replicating all modifications done to the data to all copies throughout the system; the number of transactions will therefore depend on whether or not a system is partially of fully replicated. Depending on the DDBMS architecture, updates could be performed either only in the master copy or on distributed copies. Another aspect of the architecture that must be considered is the update propagation, which, according to literature, can be eager or lazy. The former performs updates on all copies before the transaction commits, while the latter updates the information after the transaction commits. Regardless of the configuration, the database is never considered mutually consistent until all copies of all data elements contain the same values.

2.3 Fragmentation transparency

Fragmentation transparency ensures that the user is not aware of and is not involved in the fragmentation of the data [6]. Fragmentation, along with data replication, are cornerstone to having concurrent transactions and parallel execution in a DDBMS, thus improving performance and availability of the data.

However, deciding whether or not, or to at what degree you should implement fragmentation is not a simple decision. If data is not fragmented enough, operations over such fragment would be more expensive, while having overly fragmented data would affect performance by complicating fragment retrieval and integrity checks.

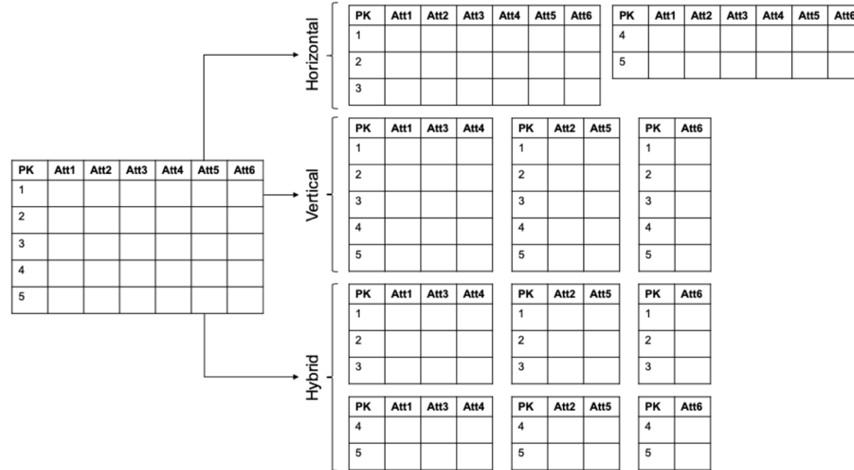
Before describing the three different typed of fragmentation, we will like to state the three rules of fragmentation. One, every tuple should be in one fragment of the original relation, this way data is not lost and the fragmentation is complete. Two, it should be possible to reconstruct the original relation by joining all of its fragments. And three, no tuple should be repeated in the same or any other fragment. These three rules will ensure the integrity of the original relation, regardless of the type of fragmentation implemented.

There are three different types of fragmentation: horizontal, vertical and hybrid as depicted in figure 4. Horizontal fragmentation means that a relation is going to be split by tuples; Vertical fragmentation means that a relation is going to be split by relation attributes or columns; lastly, hybrid is the nested implementation of the previous. While horizontal fragmentation is quite straightforward and usually done through the selection of one attribute, vertical fragmentation requires the implementation of heuristics to determine which attributes should be split (since the number of combinations grows exponentially). It should also be noted that while doing a vertical fragmentation, one must always repeat the primary key attribute in every resulting fragment.

2.4 Query processing

Non-procedural languages like SQL allow users to access and manipulate data in a simple manner. This is partly possible because the query processor is able to transform such query into an optimal procedure utilizing information about

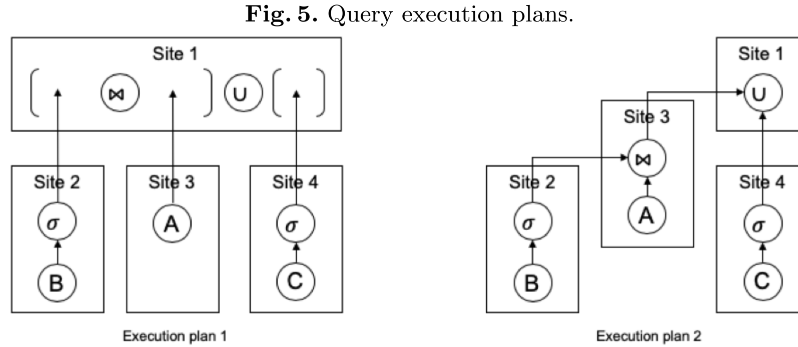
Fig. 4. Fragmentation types.



data. In DDBMS, it is even more complicated since one has to take other parameters into consideration, particularly associated with fragmentation, replication and network related costs of communication. Thus in a DDBMS the cost to be minimized must not only consider I/Os and CPUs, but communication costs as well. In order to understand how query processing is done, we will follow the 4 step methodology proposed by Tamer and Valduriez, where the first 3 steps are done by the control site and the fourth is done in local sites.

1. Query decomposition: the original query is translated into algebraic query in the same way as it is done in a centralized DBMS, utilizing the DDBMSs global schema as if it was not distributed. This means that it has to be normalized, analyzed, simplified and restructured while optimizing by I/Os and CPUs.
2. Data localization: in this step we determine all involved fragments of relations by means of from the fragment schema. This is done by substituting each relation by its reconstruction program, then it is again simplified and restructured.
3. Global optimization: it receives algebraic queries on fragments and finds the best order of operators that minimizes the cost function based on statistics, cost formulas and communication costs to the relevant sites (it is no longer the case when communication costs took longer than I/Os). Figure 5 shows an illustrative example of two possible executions plans that take into consideration communication costs and differences in computing capabilities.
4. Distributed execution: the control site then sends all transactions to the corresponding sites so they can execute the operations on fragments and return the corresponding results.

Fig. 5. Query execution plans.



In case that the DDBMS is composed of heterogeneous DBMSs, a different process is needed since each DBMS will have different computing and optimization capabilities, processing costs possibly different languages. Another factor to consider is that individual DBMSs are fully autonomous and have no concept of cooperation [7]. In this case, a mediator site will be needed to transform the query according to the other DBMSs wrapper schema (containing schema, data and processing capabilities). The first two steps are done in the mediator site and the third is done in the other DBMSs or wrapper sites.

1. Rewriting: transforms the global query into local queries by means of the global schema.
2. Optimization and execution: just like the optimization step from before, it minimizes the cost function based on statistics, cost formulas and communication costs; the difference lies in that the mediator site utilizes a capabilities schema (from executing sites) to estimate costs. Once it has the distributed query execution plan, it sends the transactions to the executing sites.
3. Translation and execution: utilizing the wrapper schema, it translates the transaction, executes it and returns results to the mediator site.

2.5 Concurrency

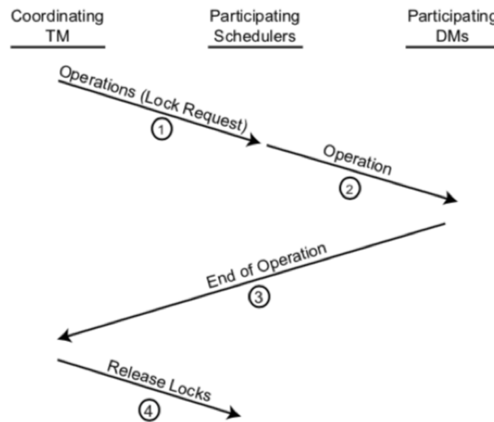
There is no better way to achieve database consistency than performing serialized transactions. This means that only one transaction may run at a single time. However, this is not consistent with the usage of modern systems, where multiple users are demanding information from the same DBMS.

This has led to develop concurrency algorithms that, given certain assumptions and implementing certain tools, allow for several transactions to be executed at the same time without compromising consistency. One must remember, though, that if two or more transactions are using the same data, they need to be serialized in order to avoid breaking consistency. Two of the most widely implemented algorithms for concurrency are the locking-based and timestamp ordering-based. Both of these algorithms are considered pessimistic, because it assumes that a transaction will have a conflict with other transactions.

The locking-based algorithm applies physical or logical locks on some portion of the database so that no other transaction can modify it while it is still in use. However, whenever there are locks, there is a way to create a deadlock. This is avoided to (some extent) with a wait-for graph, which is basically a directed graph that explores transactions in a queue to see whether or not it will create a deadlock. In DDBMSs, these wait-for graphs are done by lock managers at both global and local sites. Another less used possibility is to have the wait-for graphs only in local sites and have them communicate its graph to other local sites. The latter will then add the received structure to their own graphs in order to check for deadlocks.

The lock manager handles all lock responsibilities, but he will not serialize transactions, this is why we need to implement the two phase locking algorithm (2PL). The difference between the standard locking algorithm and the 2PL is that a transaction should not request a new lock after releasing a lock. Or rephrasing the previous statement, a transaction will not release locks until it is certain that no other lock is needed. Furthermore, if you implement the Strict 2PL, the lock manager will wait until the locking transaction is complete, then it will release all locks, reducing the chance for aborting transactions. Figure 6 represents the communication structure of a distributed 2PL, where TM is the transaction manager, Participating Schedulers are the other sites local managers, and DM are the data processing sites.

Fig. 6. 2PL communication flow on distributed databases. [1]



On the other hand, the timestamp ordering-based algorithm organizes execution according to the order in which transactions were queued. Instead of locking to ensure serialization, they select a serialization order before executing. This is done by assigning incremental timestamps to transactions. In a centralized DBMS, there is a global counter that ensures that every timestamp is unique; on DDBMS, this global counter is harder to implement, so each site maintains

its own counter and they add their site identifier to the timestamp to ensure uniqueness in the system. Since the site identifier is assigned to the least significant position of the timestamp, it will only be processed when two timestamps have the same local value. Furthermore, data has a read and write timestamp of the last time it was read or written; these timestamps are used to approve or reject operations. If an operation from a transaction is rejected by the scheduler, the transaction is restarted with a new timestamp; this algorithm avoids creating deadlocks, but its implementation may lead to requiring multiple tries to execute a transaction.

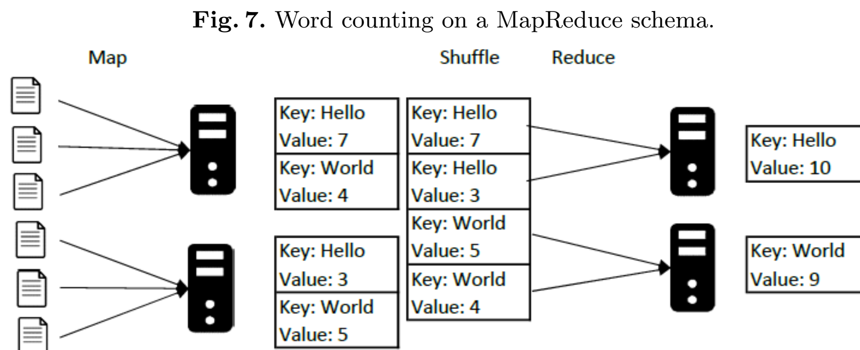
3 Apache Hive

3.1 Background

Apache Hadoop is a software developed to deal with the Big Data world, where usual tools stopped being feasible. This software framework provides utilities to work on distributed databases by using the MapReduce programming model. Such programming model allows to process information, in parallel, within a distributed environment and therefore, with high throughput.

On the figure 7, the MapReduce process is explained with the example of a word counting job. The first step is to split the data and map it to the available instances. Then, each instance, in parallel, counts the word for each chunk of data the framework mapped to it. The output of all instances is sorted and becomes the input of the reducer. On the reducer step, each instance will take a chunk of the sorted output and will reduce it to the final expected output.

Fig. 7. Word counting on a MapReduce schema.



Apache Hadoop will take charge of the mapping, shuffling, and reducing steps. It will also solve the task scheduling, states monitoring and re-schedule when a task fails. Moreover, it will improve the performance by running multiple jobs at the same time and considering DDBMS promises such as fragmentation and replication.

Notwithstanding the effectiveness of Apache Hadoop to manage distributed databases, the implementation is not simple or trivial. Each job needs to implement the map and reduce phase in Java and this is not straightforward. Specially for those coming from the relational world where SQL provides a high level language compared with Java, the learning curve tends to be complex and an entry barrier to Apache Hadoop. On the listing 1.1, there is an implementation of the word count example. The job, essentially, is not a complex algorithm, but the coding must adapt to the MapReduce architecture and it becomes messy and difficult to implement and read.

Listing 1.1. Word Count Java job for Apache Hadoop [3]

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

   public static class Map extends MapReduceBase
        implements Mapper<LongWritable, Text,
        Text, IntWritable> {
     private final static IntWritable one;
     one = new IntWritable(1);
     private Text word = new Text();

     public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter)
        throws IOException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         output.collect(word, one);
       }
     }
   }

   public static class Reduce extends MapReduceBase
        implements Reducer<Text, IntWritable,
        Text, IntWritable> {
     public void reduce(Text key, Iterator<IntWritable> values,
```

```
            OutputCollector<Text, IntWritable> output,
            Reporter reporter)
            throws IOException {
          int sum = 0;
          while (values.hasNext()) {
            sum += values.next().get();
          }
          output.collect(key, new IntWritable(sum));
        }
      }

      public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        JobClient.runJob(conf);
      }
    }
```

Since most of the traditional database paradigm is SQL based. Moving and migrating from SQL paradigm to a MapReduce paradigm does not look like an easy task. There is a big change on the implementation of the applications, the learning curve of the developers and most of the SQL-related code can not be re-used. Going back to the promises of a DDBMS, Apache Hadoop solves most of the transparency layers, except the language layer, and this can be a deal breaker while adopting a new technology.

Apache Hive comes to solve the language transparency layer, by providing an SQL-like syntax for the MapReduce on Apache Hadoop. HQL, HiveQL or Hive Query Language, allows the user to manage a distributed database by using SQL commands. Therefore, it allows to manage a distributed database as the traditional centralized relational database schema. An example can be seen in the listing 1.2; opposite to the listing 1.1, the implementation of the word count job is cleaner, easier to read and to implement with the knowledge of SQL.

Listing 1.2. Word Count query for Apache Hive

```
SELECT word, count(1)
FROM
    (SELECT explode(split(line, ' ')) AS word
    FROM document)
GROUP BY word;
```

3.2 Hive Query Language

Hive Query Language (HiveQL) has been created to assimilate SQL as much as possible. Still, queries are restricted to Apache Hadoop and MapReduce limitations. Each newer version tries to create a framework as close as SQL-92 standard, but there are some differences that are good to know and are explained in the following sections.

One of the major differences between the typical SQL and HiveQL is that traditional databases work under a schema on write, meaning the schema is checked during the data load. Counter to Hive which works on schema on read, meaning the schema is checked during the interpretation of HiveQL queries. This allows Hive a fast data load without worrying of a data schema, specially when queries are not being issued yet [4]. Therefore, HiveQL allows the user to create queries to load the original data without any ETL. Also, the schema can change over time, different schemes can be used over the same data and we can load data without worrying of structuring, so it provides more flexibility on managing the data. Nevertheless, since HiveQL works almost as SQL, the ideal scenario is to have structured or semi-structured data. This also means that Apache Hive is not suitable for OLAP applications, as processing queries on real time is not a strong point on Hive due to the schema on read and the overhead of applying the schema on each query.

3.3 Hands on Apache Hive

The objective on this section is present the basic configuration and some basic usage of the tool, to be able to compare it against the traditional SQL. The following implementation was done on Ubuntu 18.04.1 LTS, using Apache Hive 3.1.1 and Apache Hadoop 3.1.1, changes between versions or operating system may differ greatly the Bash commands presented here.

First, set up Apache Hadoop and Apache Hive. The first step is to download and uncompress the files from the official mirrors:

```
wget http://apache.belnet.be/hive/hive-3.1.1/apache-hive
    -3.1.1-bin.tar.gz
wget http://apache.belnet.be/hadoop/common/current/hadoop
    -3.1.1.tar.gz
tar -xzvf apache-hive-3.1.1-bin.tar.gz
tar -xzvf hadoop-3.1.1.tar.gz
```

Then, add the following configuration variables to bash by modifying the .bashrc, as both frameworks need them to function properly.

```
#Java Home directory configuration
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/
export PATH="$PATH:$JAVA_HOME/bin"

# Hadoop home directory configuration
export HADOOP_HOME=/home/hive/hadoop-3.1.1
export HADOOP_CONF_DIR=/home/hive/hadoop-3.1.1/etc/hadoop
export HADOOP_MAPRED_HOME=/home/hive/hadoop-3.1.1
export HADOOP_COMMON_HOME=/home/hive/hadoop-3.1.1
export HADOOP_HDFS_HOME=/home/hive/hadoop-3.1.1
export YARN_HOME=/home/hive/hadoop-3.1.1
export PATH=$PATH:/home/hive/hadoop-3.1.1/bin

export HIVE_HOME=/home/hive/apache-hive-3.1.1-bin
export PATH=$PATH:$HIVE_HOME/bin
```

After saving the file, use this command to refresh the current variables:

```
source ~/.bashrc
```

Now, specify which is the address of the NameNode, which is responsible for storing Hadoop metadata. In this case, set up a local NameNode, as there is no need of a cluster or a remote computer. Add the following property to the hadoop-3.1.1/etc/hadoop/core-site.xml file:

```
<property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
</property>
```

Now, within hadoop-3.1.1/etc/hadoop/hdfs-site.xml, add the following properties to keep the environment as simple as possible. This will set only one replication of the file system and set the current user as superuser of Hadoop:

```
<property>
    <name>dfs.replication</name>
    <value>1</value>
</property>
<property>
    <name>dfs.permission</name>
    <value>false</value>
</property>
```

To set a resource manager and job scheduling, modify `hadoop-3.1.1/etc/hadoop/mapred-site.xml`. Between the options are the classic MapReduce, a custom one or YARN. In this case, use Apache YARN by adding the next property, as is the most updated MapReduce (MPv2) [5]:

```
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

To configure YARN, modify the file hadoop-3.1.1/etc/hadoop/yarn-site.xml with the following properties:

```
<property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
</property>
<property>
        <name>
            yarn.nodemanager.auxservices.mapreduce.shuffle.class
        </name>
        <value>
            org.apache.hadoop.mapred.ShuffleHandler
        </value>
</property>
```

Also, it is ideal to configure Java home path on hadoop-3.1.1/hadoopenv.sh, since, by default, Hadoop will try the default Java version and it may not work. This is essential if there are multiple Java versions, specially since Hadoop versions are not compatible with all Java versions. To do so, modify the following line with your current Java version:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/
```

To apply all the previous configuration to the Hadoop NameNode, format the filesystem (note this will remove all the HDFS data), by using the following command:

```
hadoop-3.1.1/bin/hadoop namenode -format
```

Once everything is configured, just initialize all services by command:

```
hdfs --daemon start namenode
hdfs --daemon start datanode
yarn --daemon start nodemanager
mapred --daemon start historyserver
```

Hadoop should be up and running, now initializa the HDFS folders to storage the data:

```
hdfs dfs -mkdir -p warehouse
hdfs dfs -mkdir -p tmp
hdfs dfs -chmod g+w warehouse
hdfs dfs -chmod g+w tmp
```

To configure Hadoop for Hive, specify the heap size of Hadoop, the configuration inherited from the previous configuration, and the Hadoop path. This

can be done by modifying the file apache-hive-3.1.1-bin/conf/hive-env.sh, if the file does not exists, copy the template that remains on the same directory and modify the following lines with your current information:

```
export HADOOP_HEAPSIZE=1024
HADOOP_HOME=/home/hive/hadoop-3.1.1
export HIVE_CONF_DIR=/home/hive/hadoop-3.1.1/etc/hadoop
```

By default, Hive uses Apache Derby as database. First, configure Derby with the content of Listing 1.3 on the Anexes section, and place the code on the file apache-hive-3.1.1-bin/conf/hive-site.xml. After that, use the following command to start the database:

```
apache-hive-3.1.1-bin/bin/schematool -initSchema -dbType derby
```

By now, Apache Hadoop and Hive are up and running, and you can start loading data. You can use the los-angeles-international-airport-air-cargo-volume.csv dataset located in https://www.kaggle.com/cityofLA/los-angeles-international - airport-data/version/67

Access the CLI of Hive by calling the command hive and create a table by using the dataset header name:

```
CREATE TABLE cargo_volume
(DataExtractDate            TIMESTAMP,
ReportPeriod                TIMESTAMP,
Arrival_Departure           STRING,
Domestic_International      STRING,
CargoType                   STRING,
AirCargoTons                INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

By now, you have a database on HDFS. If you want to see the location or other metadata of the table, call:

```
DESCRIBE FORMATTED cargo_volume;
```

To load the data from the csv you can use the LOAD DATA command:

```
LOAD DATA LOCAL INPATH '/home/dani/Desktop/hive/
los-angeles-international-airport-air-cargo-volume.csv'
OVERWRITE INTO TABLE cargo_volume;
```

To check if the data was loaded correctly, just get the top of the tale with a simple query:

```
SELECT *
FROM cargo_volume
LIMIT 10;
```

If you made a mistake during the process, you can drop the table or truncate the table to keep the schema the same way as in SQL:

```
DROP TABLE cargo_volume;
TRUNCATE TABLE cargo_volume;
```

Also, Hive allows you to create external tables. This tables are handy if you want to have multiples schemes over the same dataset or if the dataset is shared between more applications. When you create as external table, Hive will keep the original location of the dataset and any drop table will not affect the data, it will only drop the schema. The query is similar, with the difference of the metadata specification:

```
CREATE EXTERNAL TABLE IF NOT EXISTS cargo_volume_ext
(DataExtractDate             TIMESTAMP,
 ReportPeriod                TIMESTAMP,
 Arrival_Departure           STRING,
 Domestic_International      STRING,
 CargoType                   STRING,
 AirCargoTons                INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION '/user/hive/warehouse/cargo_volume';
```

Most of the basic operations in SQL are already implemented in Hive and using the same syntax, as you can see in the following queries:

```
--Filtering by selection and boolean operators
SELECT *
FROM cargo_volume
WHERE Domestic_International = "Domestic"
        AND CargoType = "Mail";

--Unique projection
SELECT DISTINCT CargoType
FROM cargo_volume;

--Ordering
SELECT CargoType, AirCargoTons
FROM cargo_volume
ORDER BY AirCargoTons DESC;

--Counting
SELECT COUNT(*)
FROM cargo_volume;

--Grouping
SELECT CargoType, COUNT(*)
FROM cargo_volume
GROUP BY CargoType;
```

```sql
--Agregation functions
SELECT MAX(AirCargoTons)
FROM cargo_volume;

--Join between tables
SELECT *
FROM cargo_volume JOIN cargo_volume_ext
        ON (cargo_volume.CargoType = cargo_volume_ext.CargoType)
```

Apache Hive also has more advanced functions that help to do complex queries on simple steps. One example is the one seen on the Listing 1.2 for word counting, where, on the same line, a split is done on blank spaces and then it returns the array as a table with the explode command; then, a simple grouping function is required to count words in a few lines using the MapReduce paradigm without worrying about it.

4 Conclusions

Even though distributed databases promise to be a viable solution for modern day applications, it is still one pretty undeveloped model. While data independence, network transparency and fragmentation seem to have acceptable solutions to ensure a DDBMS reliability; replication and concurrency algorithms are fields that have yet to be further explored, especially since they are so tied up together. Replication is cornerstone to a DDBMS performance, since it allows for parallelization of the workload, but one must be careful with its implementation, since maintaining database consistency would require background processing as well. Depending on the degree of replication and the nature of the transactional throughput (read/write), one could find the DDBMS spends more resources maintaining data consistency than satisfying user transactions; a clearly non-desirable scenario. As long as DDBMS administrators keep this in mind and further research is done, the outlook for DDBMS is good.

Apache Hive is highly suitable to migrate traditional relational databases as it provides a language transparency layer to keep the common SQL of relational databases. But, even though Apache Hive is a powerful framework by mixing MapReduce with the traditional view, it can not replace the traditional SQL systems. Apache Hive is not suitable for single inserts, and, since it uses Apache Hadoop on the background, it has performance problems if used as a transactional system. Moreover, the schema on read facilitates migrating and creating a database, since the schema is not necessary at this point. This also provides the flexibility to develop the schema along the way, to evolve the schema and even to have multiples schemes for different interpretations of the data.

Apache Hive behaves as a data warehousing framework, but is not suitable for doing real time analysis. Still, this framework provides enough flexibility by giving a SQL-like interface with the distributed databases and non-structured data functions.

5 Annexes

Listing 1.3. Derby configuration XML [6].

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?><!--
Licensed to the Apache Software Foundation (ASF) under one or more
contributor license agreements. See the NOTICE file distributed with
this work for additional information regarding copyright ownership.
The ASF licenses this file to You under the Apache License, Version 2.0
(the "License"); you may not use this file except in compliance with
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
-->
<configuration>
    <property>
        <name>javax.jdo.option.ConnectionURL</name>
        <value>
            jdbc:derby:;
            databaseName=/home/apache-hive-3.1.1-bin/metastore_db;
            create=true
        </value>
        <description>
            JDBC connect string for a JDBC metastore.
            To use SSL to encrypt/authenticate the connection,
            provide database-specific SSL flag in the connection URL.
            For example, jdbc:postgresql://myhost/db?ssl=true for
            postgres database.
        </description>
    </property>
    <property>
        <name>hive.metastore.warehouse.dir</name>
        <value>/user/hive/warehouse</value>
        <description>
            location of default database for the warehouse
        </description>
    </property>
    <property>
        <name>hive.metastore.uris</name>
```

```
        <value/>
        <description>
            Thrift URI for the remote metastore. Used by metastore
            client to connect to remote metastore.
        </description>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>org.apache.derby.jdbc.EmbeddedDriver</value>
        <description>
            Driver class name for a JDBC metastore
        </description>
    </property>
    <property>
        <name>javax.jdo.PersistenceManagerFactoryClass</name>
        <value>
            org.datanucleus.api.jdo.JDOPersistenceManagerFactory
        </value>
        <description>
            class implementing the jdo persistence
        </description>
    </property>
</configuration>
```

References

1. Ozsu, M. T.,& Valduriez, P. (2011). Principles of Distributed Database Systems, Third Edition. New York, NY: Springer New York.
2. Dean, J & Ghemawat, S. (2014). MapReduce: Simplified Data Processing on Large Clusters.
3. Apache. (2018). Hadoop 1.2.1 Documentation. Retrieved from: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
4. T. White. (2009). Hadoop: The Definitive Guide. O'Reilly, first edition edition.
5. Cloudera. (2018). Migrating from MapReduce 1 (MRv1) to MapReduce 2 (MRv2, YARN). Retrieved from: https://www.cloudera.com/documentation/enterprise/5-3-x/topics/cdh_ig_mapreduce_to_yarn_migrate.html
6. Edureka. (2018). Apache Hive Installation on Ubuntu. Retrieved from: https://www.edureka.co/blog/apache-hive-installation-on-ubuntu
7. J. Gamper. (2009). Distributed Databases, Chapter 1: Introduction. Retrieved from: http://www.inf.unibz.it/dis/teaching/DDB/ln/ddb01.pdf
8. M. Bhlen. (2018) . Distributed Database Systems: Introduction SL01. Retrieved from: https://files.ifi.uzh.ch/dbtg/ddbs/HS16/sl01.pdf
9. Silberschatz, Korth and Sudarshan. (2005). Database System Concepts. Retrieved from: https://www.cse.iitb.ac.in/~sudarsha/db-book/slide-dir/ch22.pdf
10. H. Rababaah. (2005). Distributed Databases Fundamentals and Research. Retrieved from: https://www.researchgate.net/publication/241097817_DISTRIBUTED_DATABASES_FUNDAMENTALS_AND_RESEARCH

11. T. Mitakos. (2006). Distributed Databases. Retrieved from: https://e-class.teilar.gr/modules/document/file.php/CS164/EDB_2006/Distributed%20Databases.pdf
12. J. Bunn. (2001). Distributed Databases: Part 2. Retrieved from: http://pcbunn.cithep.caltech.edu/distributeddatabasespakistan.pdf
13. N. Kumar, S. Bilgaiyan and S. Sagnika. (2013). An Overview of Transparency in Homogeneous Distributed Database System. Retrieved from: https://www.researchgate.net/publication/270895053_An_Overview_of_Transparency_in_Homogeneous_Distributed_Database_System
14. R. Board. (1992). Distributed Database Systems. Retrieved from: https://iassistdata.org/sites/default/files/iqvol16_3_board.pdf
15. P. Raghavan. (2001). Distributed Databases: Lecture 13. Retrieved from: http://infolab.stanford.edu/cs347.2001.spring/