Technical system Documentation

For

Big Data Integration and Analysis

Presented by

Gaurav Kumar

Héctor Ugarte

Miguel Mármol

Tina Boroukhian

University of Bonn

Summer 2015

Table of contents

1. Introduction

These days data streams from each and every activity of daily life: from phones and credit cards and televisions and computers; from sensor equipped buildings, GPS, trains, buses, planes, bridges, and factories. The data flows so fast that the total accumulation of the past two years—a zettabyte—dwarfs the prior record of human civilization. This huge amount of data is very important as it contains a lot of useful information and considering the volume, velocity and variety of data, cleaning and analyzing big data is a big challenge.

A real example of such challenge can be seen in Ecommerce company such as Amazon where they have huge amount of customer related data. This data is very important for company so that they can analyze data for future predictive modeling about most sought product by customers. [1][2]

In our project, we are aiming for Big Data Integration and Analysis.

1.1 Data Integration and Analyzing

Extracting, reformating, and integrating data is very important for developer. Data reshaping programs are difficult to write because of their complexity, but they are required because each analytic tool expects data in a very specific form and to get the data into that form typically requires a whole series of cleaning, normalization, reformatting, integration, and restructuring operations.

Analyzing big data is used to find meaning and discover hidden relationships in big data. The technological advances in storage, processing, and analysis of Big Data include the rapidly decreasing cost of storage and CPU power in recent years; the flexibility and cost-effectiveness of datacenters and cloud computing for elastic computation and storage; and the development of new frameworks such as Hadoop and Spark, which allow users to take advantage of these

distributed computing systems storing large quantities of data through flexible parallel processing. [1],[3],[4].

1.2 Apache Hadoop

Apache Hadoop [5] is an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware.

The core of Apache Hadoop consists of:

- Storage part: Hadoop Distributed File System (HDFS).
- Processing part: Hadoop MapReduce.

Hadoop splits files into large blocks and distributes them amongst the nodes in the cluster. To process the data, Hadoop MapReduce transfers packaged code for nodes to process in parallel, based on the data each node needs to process.

The base Apache Hadoop framework is composed of the following modules:

- Hadoop Common – contains libraries and utilities needed by other Hadoop modules;
- Hadoop Distributed File System (HDFS) – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- Hadoop YARN – a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications; and
- Hadoop MapReduce – a programming model for large scale data processing.

A MapReduce program is composed of a Map() procedure that performs filtering and sorting and a Reduce() procedure that performs a summary operation. It follows these steps:

- "Map" step: Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for

redundant copies of input data, only one is processed.

- "Shuffle" step: Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.
- "Reduce" step: Worker nodes now process each group of output data, per key, in parallel.

1.3 Apache Spark

Apache Spark [6] is an open-source cluster computing framework. It provides high-level APIs in Java, Scala and Python. In contrast to Hadoop's two-stage disk-based MapReduce paradigm, Spark's in memory primitives provide performance up to 100 times faster.

Spark runs locally on each node and executes in memory when possible. Based on Spark's Resilient Distributed Datasets (RDD), Spark can employ RAM for dataset persistence. Spark stores files for chained iteration in memory as opposed to using temporary storage in HDFS, as Hadoop does. Contrary to Hadoop, Spark utilizes multiple threads instead of multiple processes to achieve parallelism on a single node, avoiding the memory overhead of several JVMs.

The fundamental programming abstraction is called Resilient Distributed Datasets(RDD), a distributed memory abstraction that lets programmers perform in memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory.

It has two components:

- Driver
- Workers

Driver

- ➢ Defines and invokes actions on RDDs
- ➢ Tracks the RDDs' lineage

Workers

- ➢ Store RDD partitions
- ➢ Perform RDD transformations

Spark Runtime

The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

1.4 Operating Environment

This project can be run on Linux. The Apache Hadoop and Spark run as a backend and JDK, Eclipse IDE, Maven and Scala is required. Data Integration done by Apache Hadoop and Data Analysis done by Apache Spark.

1.5 Description

In this project we have two main part which are Data Integration and Data Analysis. Data Integration done by Apache Hadoop and Data Analysis done by Apache Spark. The type of input data should be CSV file(s). In Integration part Apache Hadoop convert CSV file to HDFS file; so input and output of Apache Spark is HDFS file(s).

2. Architecture Model

2.1 Architecture Diagram

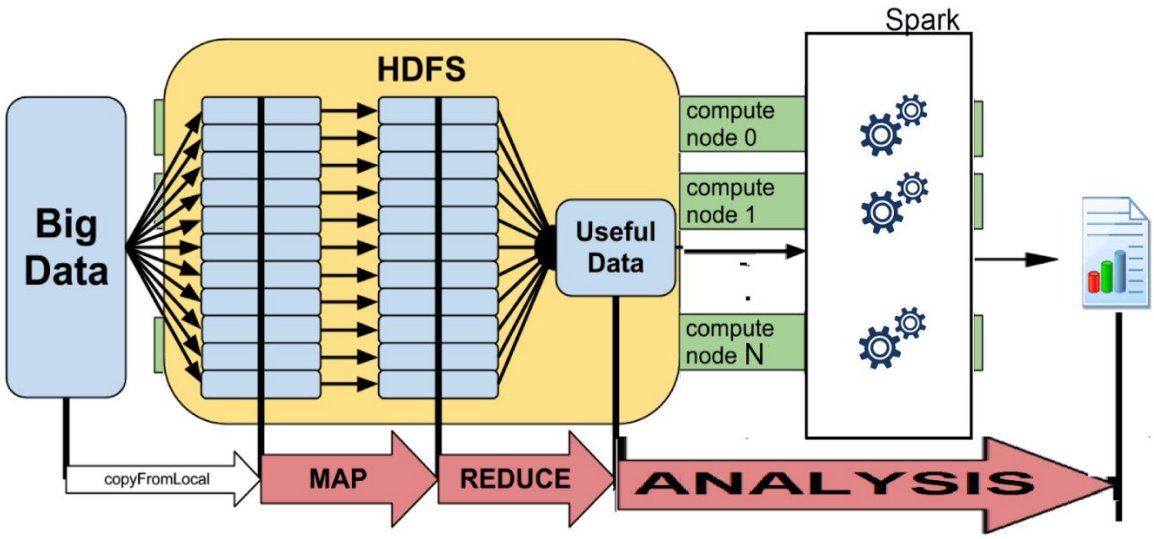


Figure 1: Architecture of Integration and Analyzing of Data [7]

3. Requirements

3.1 Functional Requirements

The functional requirements are grouped according to use case model.

A requirement has the following properties:

- Requirement ID uniquely identifies requirement.
- Title defines the functional group the requirement belongs to. It gives the requirement a symbolic name.
- Description the definition of the requirement.
- Priority defines the order in which requirements should be implemented. Priorities are designated (highest to lowest) "1", "2", and "3". Requirements of priority 1 must be implemented in the first productive system release. The requirements of priority 2 and lower are subject of special release agreement, which is out of scope of this document.
- Risk specifies risk of not implementing the requirement. It shows how the particular requirement is critical to the system. Next are the risk level and the associated

impact to the system if the requirement is not implemented or implemented incorrectly:

Critical (C) – will break the main functionality of the system. The system cannot be used if this requirement is not implemented.

High (H) – will impact the main functionality of the system. Some functions of the system could be inaccessible, but the system can be generally used.

Medium (M) – will impact some system's features, but not the main functionality. System can be used with some limitation.

Low (L) – the system can be used without limitation, but with some workarounds.

Req ID.	R01.
Title	**Get raw sample data in CSV files.**
Description	Get or collect data from our sources. Input data must be in CSV file format.
Priority	1
Risk	C
References	-

Req ID.	R02.
Title	**Move data to HDFS.**
Description	Move data to HDFS (hadoop Distributed file system) so that data can be processed with MapReduce programming model.
Priority	1
Risk	C
References	R01.

Req ID.	R03.
Title	**Clean data.**
Description	Clean and format data in a proper manner such as standard date and correct number formatting.
Priority	1
Risk	C
References	R01, R02.

Req ID.	R04.
Title	**Reduce columns.**
Description	Reduce the columns so that we keep only needed columns for desired results.
Priority	1
Risk	M
References	R02.

Req ID.	R05.
Title	**Analyze data.**
Description	Use analytic queries on above formatted data. Analytic queries contain aggregation functions such as finding max, min, average or count.
Priority	1
Risk	M
References	R04.

Req ID.	R06.
Title	**Get results.**
Description	get desired result and move output from HDFS to local machine.
Priority	1
Risk	M
References	R05.

Table 1: Functional Requirements

3.2 Non-Functional Requirements

3.2.1 Scalability

The system should be able to scale many nodes capable of processing increasing sizes of data within a reasonable span of time.

3.2.2 Performance

System should be working in high performance manner so that it can process huge amounts of data within a reasonable span of time.

3.2.3 Fault Tolerant

System should be able to behave as fault tolerant, i.e. in case of a failure of one or more machines, it will continue working.

3.2.4 Distributed Environment

System should be working in a distributed computing environment, where processes can run in parallel across multiple machines.

4. System Summary

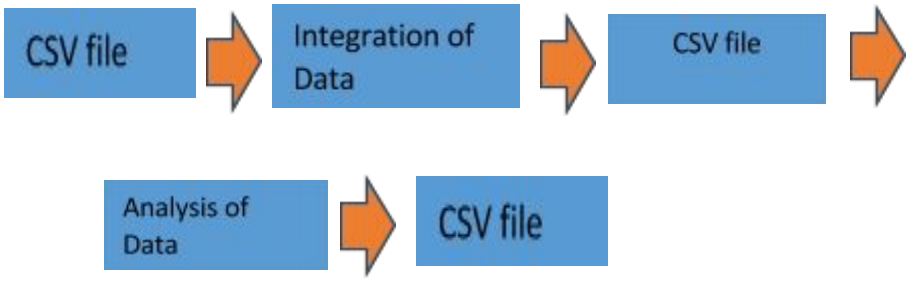
a. System Configuration

- Ubuntu 14.4
- Hadoop 2.6
- Spark 1.4
- Scala 2.10
- Maven
- Eclipse IDE
- Java JDK 1.8

5. Approach

5.1 Data Collection

We need to create the big data .So, we create big CSV files with scripts (Python). CSV file is input file in our project and will be stored on HDFS.



5.2 Integration of Data

We install Hadoop 2.6 http://pingax.com/install-hadoop2-6-0-on-ubuntu/ in Ubuntu14.4. We set it up for working as a single node and start developing in Java. In Java we need to write one Mapper, one reducer and one Driver. Now we can say that Mapper will do the task of Transformation and Restriction and the Reducer will add the Header into output file and the files will be stored in Hadoop file system (HDFS). We are using five different types of transformation operations using Map-Reduce as offered by the software: merge columns, Split columns, change the letter case of columns, change formatting of date columns and one for rename headers. In Restriction Operations section, we can restrict the output according to some criteria: For numeral values: =, <>, >, <,>=, <= for textual values: EQUAL, NOT EQUAL, CONTAINS. These restriction are used as a filter to output more desired and meaningful data. In splitting columns, we can split the dates in days and months-year combination which later can be used for analyzing queries based on days or months. Changing the case of strings is

useful for better reading. Date formatting is needed as per different regional need or user preference. Merging is required when some information can be viewed in one columns instead of multiple columns.

5.2.1 STRUCTURE OF GRAMMAR

We decide to define a grammar in this way would be easier to run a job using some keywords and parameters.

5.2.1.1 Keywords and Parameters

KEYWORDS

There are 10 keywords, can be written in uppercase or lowercase or combinations of both, for explanation purposes we will write them on uppercase:

INPUTFILE, OUTPUTFILE, SEPARATOR, PROJECTEDCOLUMNS, PROJECTEDCOLUMNSNAMES, MERGE, SPLIT, CASE, FORMATDATES, RESTRICTION

OBLIGATORY	OPTIONAL
INPUTFILE	MERGE
OUTPUTFILE	SPLIT
SEPARATOR	CASE
PROJECTEDCOLUMNS	FORMATDATES
PROJECTEDNAMES	RESTRICTION

Table 2: Some keywords are obligatory and others optional (according to the requirements of each task)

PARAMETERS

Parameters are defined with the following structure:

Par1|Par2|…

If we don't want to define parameters **on optional keywords,** we can use the keyword:

```
empty
```

Example: **MERGE** empty

or just don't write that statement.

INPUTFILE

Define the input CSV file name.

Par1: File name

Example: **INPUTFILE** Customers.csv

OUTPUTFILE

Define the output CSV file name.

Par1: File name

Example: **OUTPUTFILE** CustomersOutput.csv

SEPARATOR

Define the separator (delimiter) used on the input file. Supports only one character.

Most common used ones are Comma (,) Semi-colon (;) Pipe (|) Caret (^)

Par1: Separator character

Example: **SEPARATOR**

PROJECTEDCOLUMNS

Define the index of the columns to be projected (with 0 being the first index).

Par1: First projected column.

Par2: Second projected column.

…

ParN: N projected column.

Example: **PROJECTEDCOLUMNS** 1|3

PROJECTEDNAMES

Define the names of the columns to be projected.

Par1: First projected column name.

Par2: Second projected column name.

…

ParN: N projected column name.

Example: **PROJECTEDNAMES** Name|City

[MERGE]

Define the index of the two columns to be merged and the merge character.

Par1: First column index.

Par2: Second column index.

Par3: Merge character.

Example: **MERGE** 0|1|

[SPLIT]

Define the index of column to be splitted and the character.

Par1: Column index.

Par2: Split character.

Example: **SPLIT** 2|_

[CASE]

Define the index of column to case

Par1: Column index.

Par2: 0 for UPERCASE, 1 for LOWERCASE

Example: **CASE** 1|0

[FORMAT]

Define the index of date column to be formatted.

Par1: Column index.

Par2: DD/MM/YYYY MM/DD/YYYY YYYY/MM/DD

Example: **FORMAT** 3|MM/DD/YYYY

[RESTRICTION]

Define the index of column to be restricted, the operator used and the value.

Par1: Column index.

Par2: For numeral values: =, <>, >,<,>=,<= For textual values: EQUAL, NOT EQUAL, CONTAINS

Par3: value

etc

Example: **RESTRICTION** 0|>=|20|0|<|50

EXAMPLES:

Our example CSV input file is called Customers.csv, and has 6 columns: Customer_ID|Name|Address|City|ZipCode|Phone

A. We want as output 3 columns: Customer_ID, Address, City. And delimiter Customer_ID with Values >= than 20 and < 50.

```
INPUTFILE Customers.csv
OUTPUTFILE Output.csv
SEPARATOR |
PROJECTEDCOLUMNS 0|2|3
PROJECTEDNAMES Customer_ID|Address|City
MERGE empty
SPLIT empty
CASE empty
FORMATDATES empty
RESTRICTION 0|>=|20|0|<|50
```

Is also valid to omit the empty optional keywords, like:

```
INPUTFILE Customers.csv
OUTPUTFILE Output.csv
SEPARATOR |
PROJECTEDCOLUMNS 0|2|3
PROJECTEDNAMES Customer_ID|Address|City
RESTRICTION 0|>=|20|0|<|50
```

B. We want as output all the columns, but rename some of them as: : Customer_ID -> ID, ZipCode -> Postal_Code. And case Name to Uppercase.

```
INPUTFILE Customers.csv
OUTPUTFILE Output.csv
SEPARATOR |
```

```
PROJECTEDCOLUMNS 0|1|2|3|4|5
PROJECTEDNAMES ID|Name|Address|City|Postal_Code|Phone
MERGE empty
SPLIT empty
CASE 1|0
FORMATDATES empty
RESTRICTION empty
```

C. We want as output only Column Name, but we want to split it on two columns First_Name and Last_Name.

```
INPUTFILE Customers1.csv
OUTPUTFILE Output.csv
SEPARATOR |
PROJECTEDCOLUMNS 1|1000
PROJECTEDNAMES First_Name|Last_Name
MERGE empty
SPLIT 1|
CASE empty
FORMATDATES empty
RESTRICTION empty
```

5.3 Analyzing of Data

We install Spark 1.4 and Scala 2.10 http://blog.prabeeshk.com/blog/2014/10/31/install-apache-spark-on-ubuntu-14-dot-04/ in Ubuntu14.4. We set it up for working as a single node and start developing in Java. In spark with the help of SQL libraries, we are able to use SQL functionalities. We can select data and do some aggregation operation on the data. Here we are going to use the data which was generated after integration. We are supposed to write a SQL

query on input files. There is also a delimiter option added, in case files have different delimiter other than comma.

We can even join multiple files using join condition on common columns. These common column work as a foreign key relation between files. We can also operate aggregation operations using GROUP BY. In SQL queries we

can add single or multiple where clauses to filter data based on some required conditions.

Later on we created a cluster for spark, which contains one master node and 3 Slaves nodes. The data load is distributed equally on all nodes and querying is faster as compared to single node performance.

5.4. Virtual Machine Setup

1. On Ubuntu machine, we first install Java.
2. Then we install SSH for connection between nodes.
3. Now we create a Hadoop user for accessing HDFS and MapReduce.
4. Now configure SSH for generating keys.
5. Now configure xml configuration for Hadoop, as mentioned in tutorial http://pingax.com/install-hadoop2-6-0-on-ubuntu/
6. Start all services and verify from web interface.
7. Now we can extend Hadoop to a cluster using the steps mentioned in tutorial http://pingax.com/install-apache-hadoop-ubuntu-cluster-setup/
8. Here we have one master node and 3 slaves nodes.
9. We can verify from web interface that Hadoop cluster is running.
10. Now we install Scala as a prerequisite for spark setup.
11. Now download spark 1.4.1 from spark website.
12. Now we follow the steps as mentioned in tutorial http://www.trongkhoanguyen.com/2014/11/how-to-install-apache-spark-121-in.html
13. Now we can verify from web interface that spark is running. Afterwards we created Input directory and uploaded some files on HDFS, on which we ran llama.
14. Write SQL queries and software will result in desired dataset of queries.

6. Testing

6.1 Unit Testing

We are working with Java, therefore we will use Junit to test unit test cases.

In first case, we are verifying for presence of files on HDFS after the file is uploaded to HDFS. As shown in figure 2.

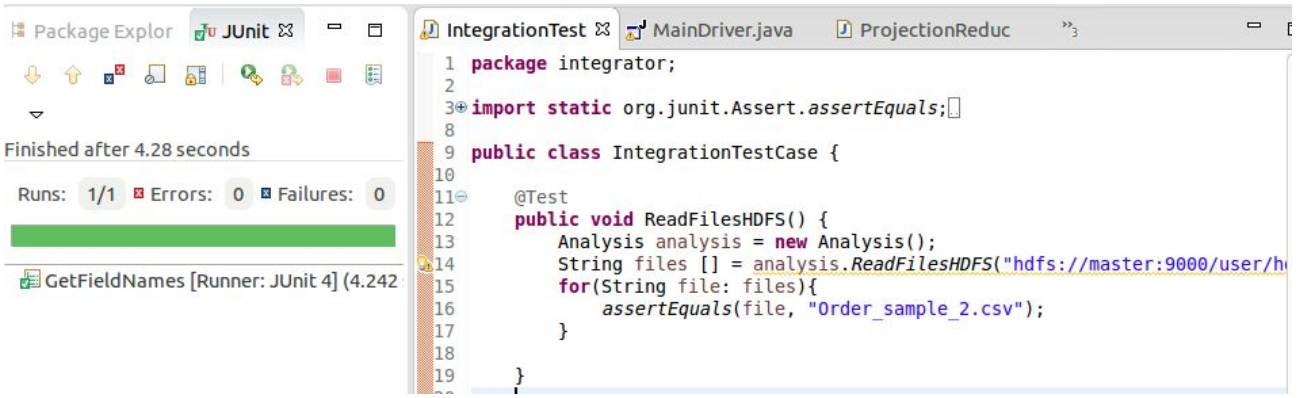


Figure 2

In second case, we are verifying for correct generated fields after we import the file. As shown in figure 3.

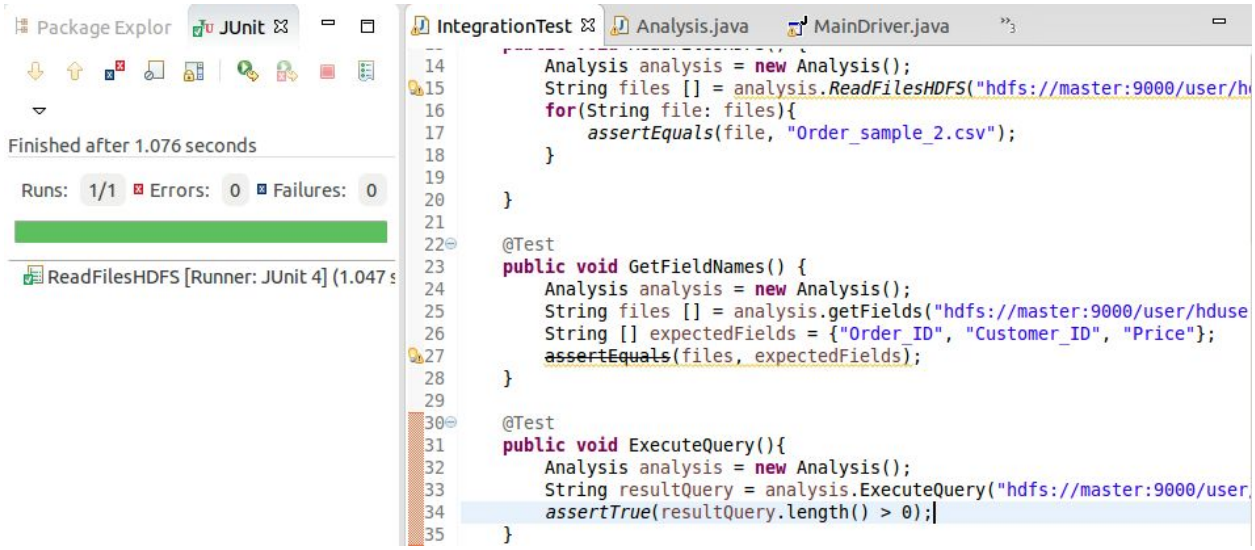


Figure 3

In third case, we are verifying for the presence of inserted query . As shown in figure 4.

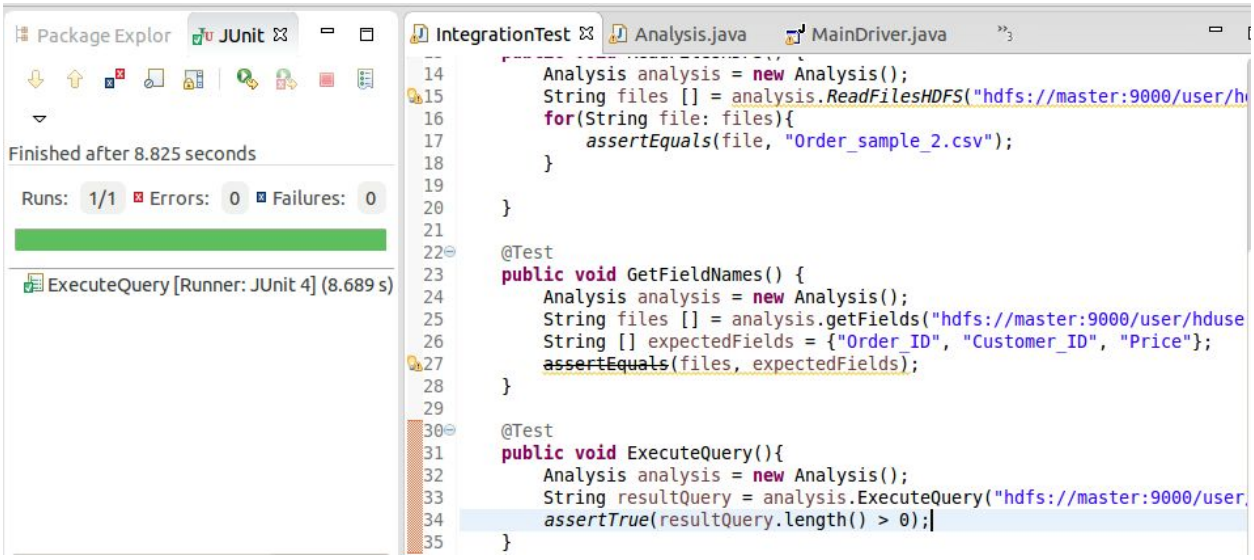


Figure 4

In fourth case, we are verifying for presence of output file generated after running the sql query on file.. . As shown in figure 5.

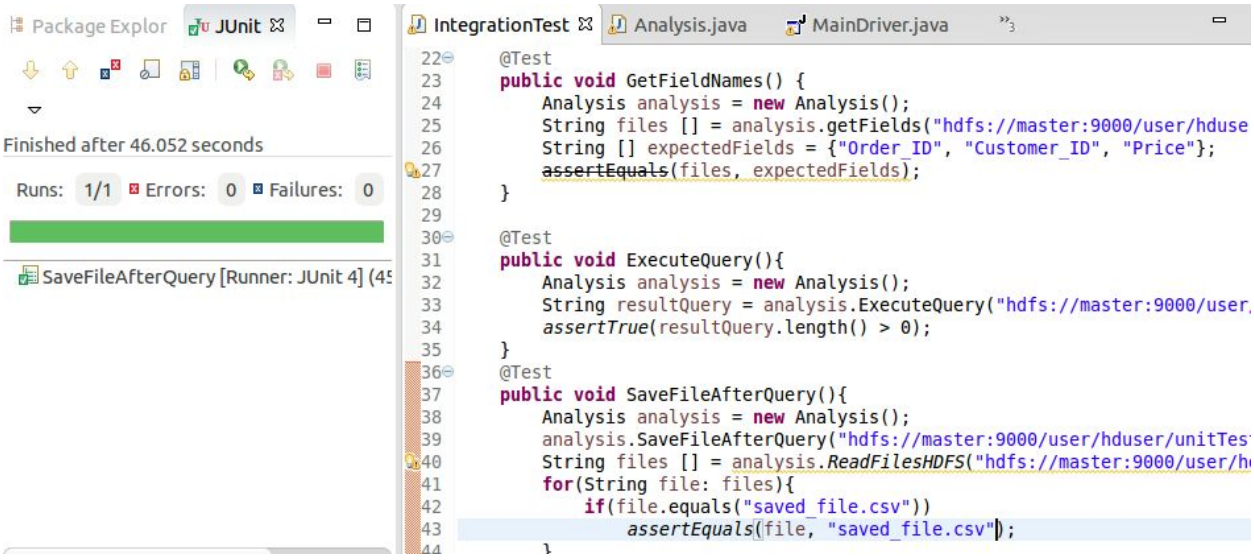


Figure 5

6.2 Integration Testing

We have tested Integration functionality on different test cases to check for merge, split, filter and formatting features.

6.3 Validation Testing

We have run specific SQL queries for which we run our software and verify the output with expected result. The correct results validated the software.

6.4 Performance Testing

6.4.1 Environment Data Integration

- Apache Hadoop 2.6
- Ubuntu 14.04
- Java 1.8

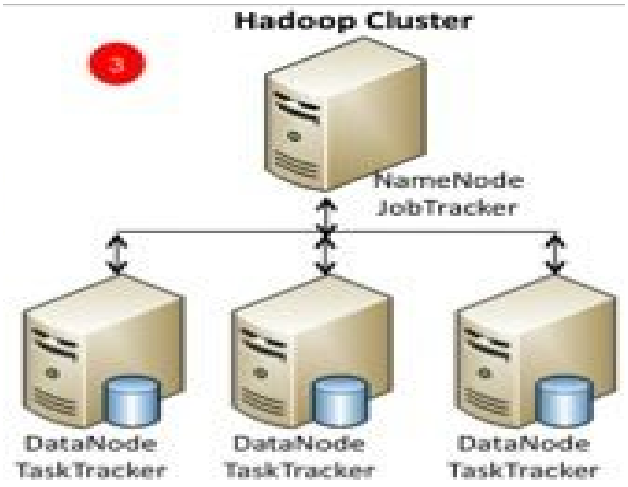


Figure 6: Hadoop Cluster Architecture

NameNode architecture:

Brand and Model	Apple Macbook Pro 13

CPU	Core i7 2.9 Ghz
RAM	8 GB
Hard Disk	750 GB

Table 7: NameNode architecture

DataNode 1 architecture:

Brand and Model	Dell Alienware m11x r1
CPU	Core 2 Duo 1.7Ghz
RAM	8 GB
Hard Disk	320 GB

Table 8: DataNode 1 architecture

DataNode 2 architecture:

Brand and Model	Dell Latitude 3450
CPU	Core i3 2.4Ghz
RAM	4 GB
Hard Disk	500 GB

Table 9: DataNode 2 architecture

DataNode 3 architecture:

Brand and Model	Asus Zenbook UX303L
CPU	Core i7 2.4Ghz

RAM	8 GB
Hard Disk	1000 GB

Table 10: DataNode 3 architecture

6.4.1.1 Input Testing Files

We have 4 input files:

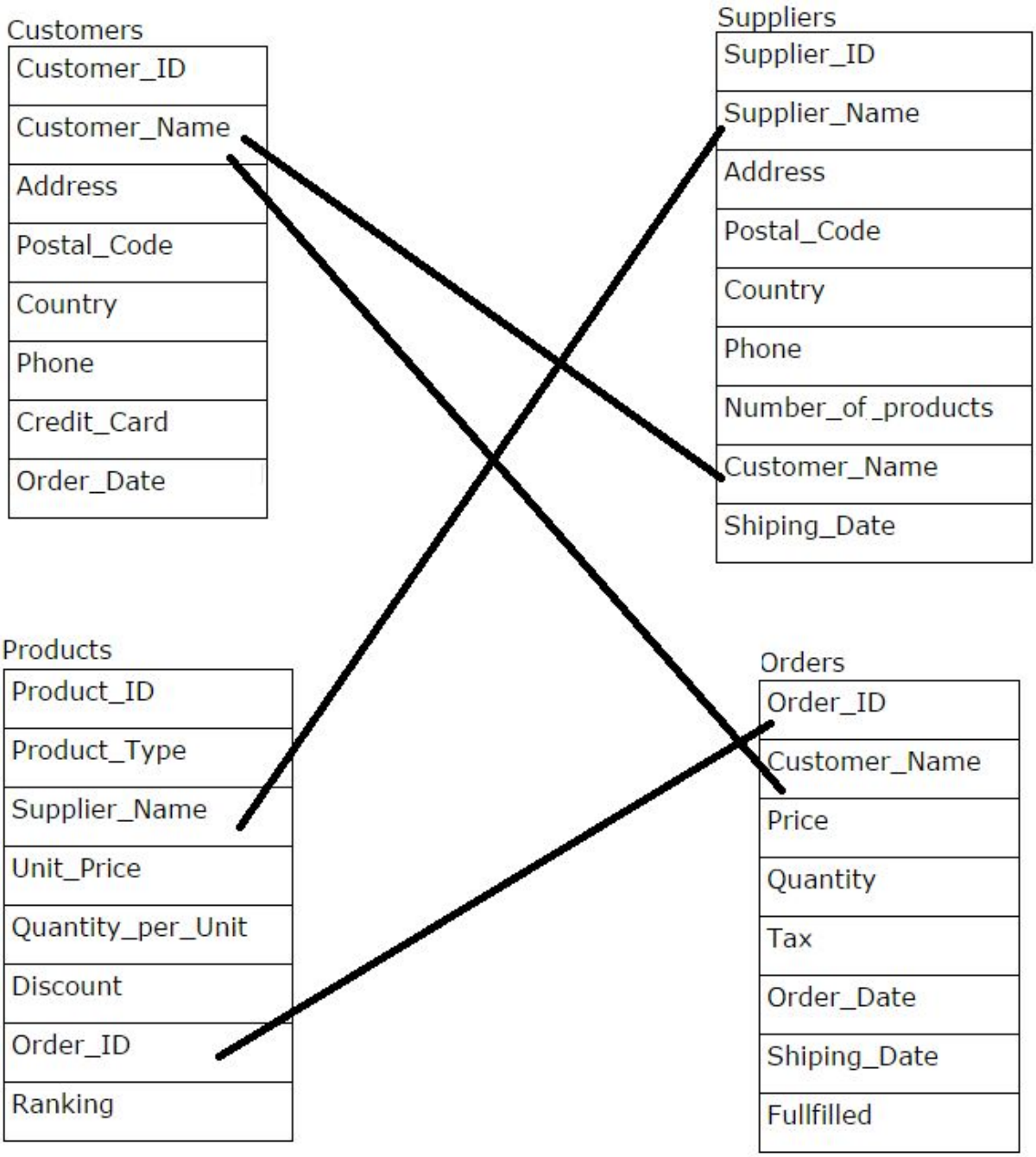
1) Customers.csv
2) Supplier.csv
3) Orders.csv
4) Products.csv

6.4.2 Environmental Data Integration

- Apache Hadoop 2.6
- Apache Spark 1.4.1
- Ubuntu 14.04
- Java 1.8

Spark Cluster configuration

	Master	**Slave 1**	**Slave 2**	**Slave 3**
Worker Memory	3GB	4GB	2GB	4GB
Executor Memory	1GB	1GB	1GB	1GB
Cores	1	1	2	2
Instances	3	4	2	4



6.4.3 Integration Test Case

CASE 1: Project from customers table Customer_name and Address and cast Address to Upper case.

SCRIPT:

```
INPUTFILE Customer.csv
OUTPUTFILE Customer_Output.csv
SEPARATOR ,
PROJECTEDCOLUMNS 1|2
PROJECTEDNAMES Customer_Name,Address
MERGE empty
SPLIT empty
CASE 2|0
FORMATDATES empty
RESTRICTION empty
```

CASE 2: Project from Orders table Order_id, Customer_name and Order_date and split Order_date on Month and DayYear.

SCRIPT:

```
INPUTFILE Orders.csv
OUTPUTFILE Orders_Output.csv
SEPARATOR ,
PROJECTEDCOLUMNS 0|1|5|1000
PROJECTEDNAMES Order_ID,Customer_Name,Month,DayYear
MERGE empty
SPLIT 5|/
CASE empty
FORMATDATES empty
RESTRICTION empty
```

CASE 3: Project from Supplier Supplier_Id, Company_Name, Address and Countryname and merge address and country_Name.

SCRIPT:

```
INPUTFILE Supplier.csv
OUTPUTFILE Supplier_Output.csv
SEPARATOR ,
PROJECTEDCOLUMNS 0|1|2
```

```
PROJECTEDNAMES Supplier_ID,Company_Name,Address_Country
MERGE 2|4|-
SPLIT empty
CASE empty
FORMATDATES empty
RESTRICTION empty
```

6.4.4 Analysis Test Cases

CASE 1: Find all supplier ID and Supplier Name with number of products delivered with total Unit price, total Quantity and total Discount given, only products having ranking <5

QUERY:

```
select Suppliers.Supplier_ID, Suppliers.Supplier_Name,
count(Product_ID),sum(Unit_Price),sum(Quantity_per_Unit),su
m(Discount) from Suppliers, Products where
Suppliers.Supplier_Name =Products.Supplier_Name and Ranking
< 5
group by Suppliers.Supplier_ID, Suppliers.Supplier_Name
```

CASE 2: find number of available suppliers for customers

QUERY:

```
select Suppliers.Customer_Name,count(Suppliers.Supplier_ID)
from Suppliers, Customers
where Suppliers.Customer_Name =Customers.Customer_Name
group by Suppliers.Customer_Name
```

CASE 3: Find number of available Products in fulfilled orders having no of products greater than 10

QUERY

```
select Products.Order_ID, count(Products.Product_ID)
from Products, Orders
where Products.Order_ID = Orders.Order_ID
and Orders.Fulfilled ='Yes'
```

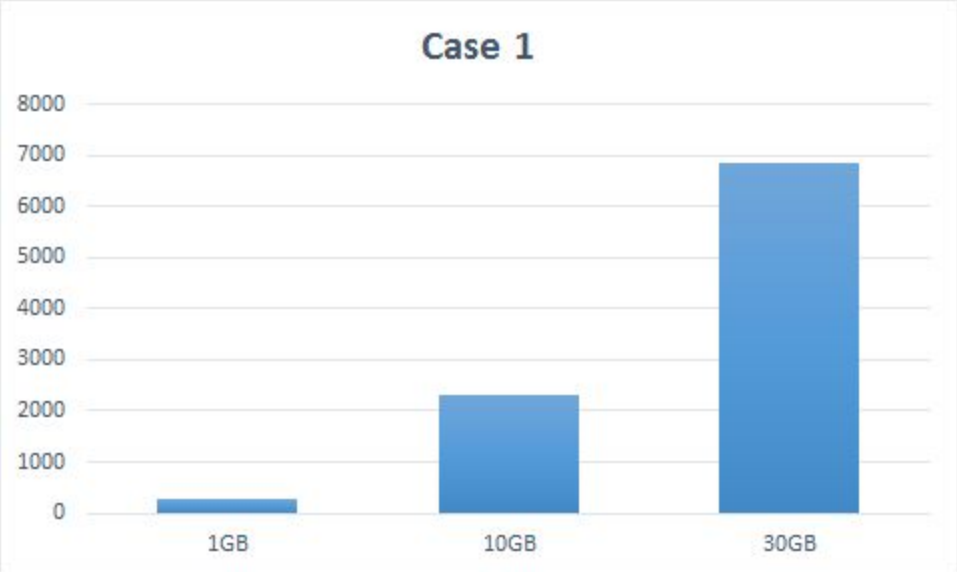
```
group by Products.Order_ID
having count(Products.Product_ID) > 10
```

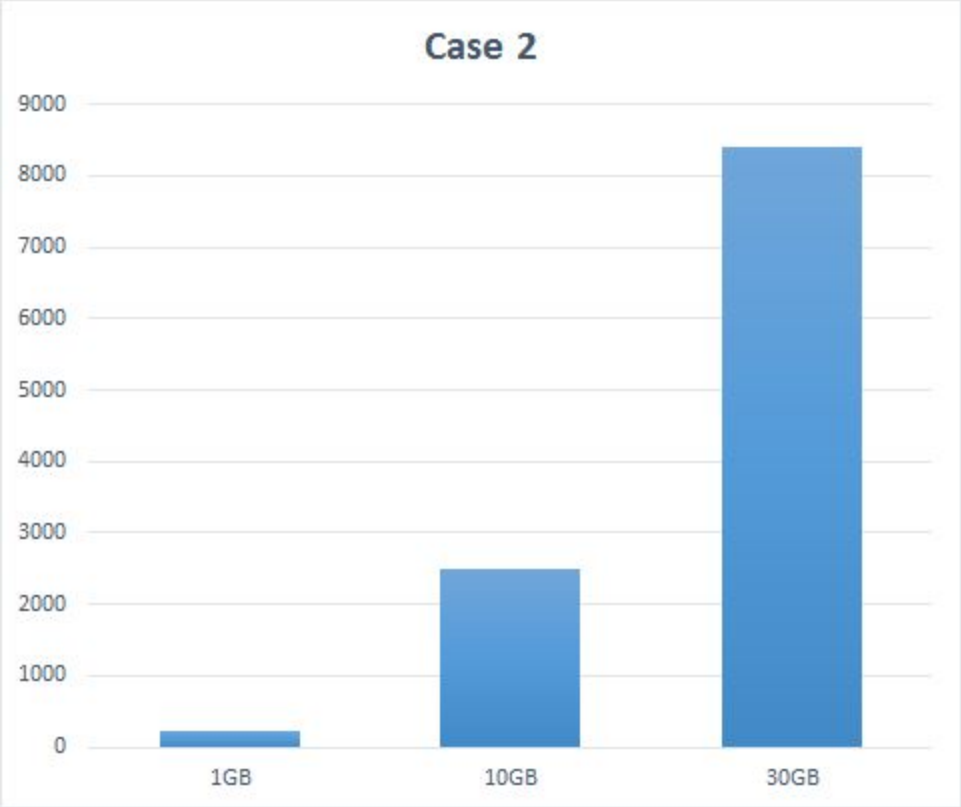
6.4.5 Result

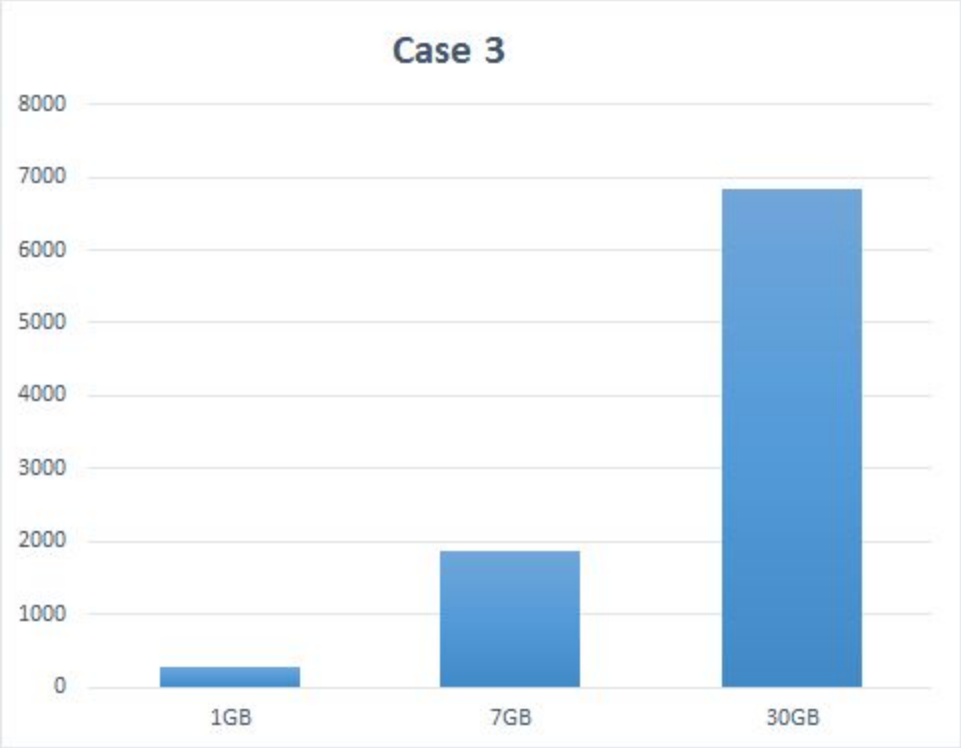
6.4.5.1 Integration

Case	Size	Tables	Time (seconds)
Case 1	1GB	Customer	276
Case 1	10GB	Customer	2304
Case 1	30GB	Customer	6872
Case 2	1GB	Order	223
Case 2	10GB	Order	2499
Case 2	30GB	Order	8396
Case 3	1GB	Supplier	267
Case 3	7GB	Supplier	1869
Case 3	30GB	Supplier	6842

Table 7: Result of Integration


Case 1
8000
7000
6000
5000
4000
3000
2000
1000
0
1GB
10GB
30GB

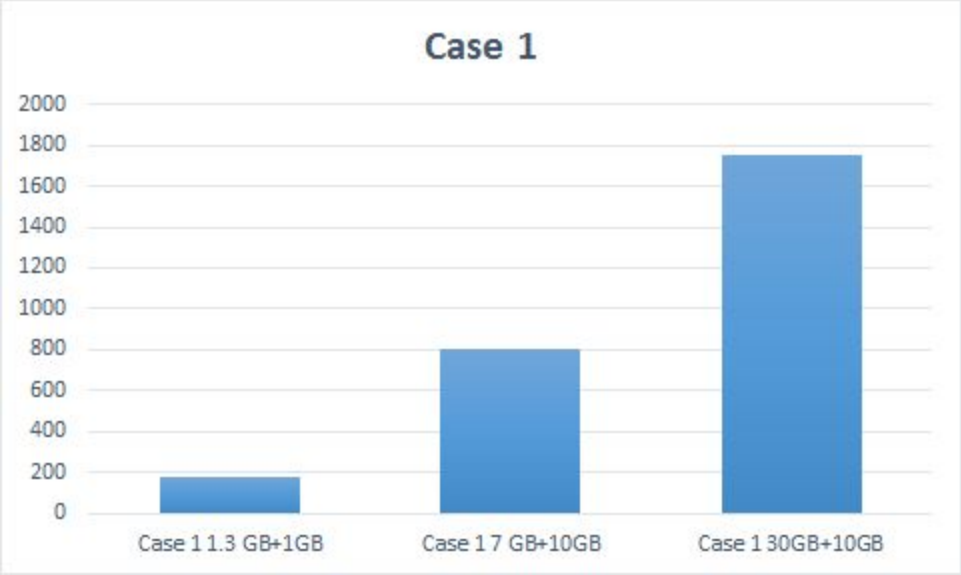

Case 2
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1GB
10GB
30GB


Case 3
8000
7000
6000
5000
4000
3000
2000
1000
0
1GB
7GB
30GB

6.4.5.2 Analysis

Case	Size	Tables	Time
Case 1	1.3 GB + 1GB	Suppliers, Products	180
Case 1	7 GB +10GB	Suppliers, Products	804
Case 1	30 GB+ 10GB	Suppliers, Products	1756
Case 2	1.3 GB + 1.3 GB	Customer, Suppliers	156
Case 2	10GB +7 GB	Customer, Suppliers	660
Case 2	28 GB +27 GB	Customer, Suppliers	2520
Case 3	1GB +1 GB	Products, Orders	253
Case 3	10GB+ 3 GB	Products, Orders	496
Case 3	10GB + 10GB	Products, Orders	986

Table 8: Result of Analysis


Case 1
2000
1800
1600
1400
1200
1000
800
600
400
200
0
Case 1 1.3 GB+1GB
Case 1 7 GB+10GB
Case 1 30GB+10GB


Case 2
3000
2500
2000
1500
1000
500
0
Case 2 1.3 GB+1.3 GB
Case 2 10GB+7 GB
Case 2 28GB+27GB

7. References

[1]http://www.oracle.com/technetwork/database/options/advanced-analytics/bigdataanalyticswpoaa-1930891.pdf

[2] http://harvardmagazine.com/2014/03/why-big-data-is-a-big-deal

[3] http://usc-isi-i2.github.io/papers/knoblock13-sbd.pdf

[4]https://downloads.cloudsecurityalliance.org/initiatives/bdwg/Big_Data_Analytics_for_Security_Intelligence.pdf

[5] https://en.wikipedia.org/wiki/Apache_Hadoop

[6] https://en.wikipedia.org/wiki/Apache_Spark

[7]http://www.glennklockwood.com/data-intensive/hadoop/mapreduce-workflow.png