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 flows from each and every activity of daily life: from phones, ATMs[1], televisions and computers; also from sensor equipped buildings, GPS, trains, buses, planes, bridges, and factories. The data streams 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. Considering the volume, velocity and variety of data, cleaning and analyzing big data is a big challenge.

A real example of such challenges can be seen in e-commerce companies, such as Amazon, where huge amounts of customer related data are collected continuously. Analysing the data is crucial for the company to understand users interactions. This results in constructing predictive models about, for example, most sought product per age categories and seasons. [1][2]

In our project, we are aiming to build a system for big data Integration and Analysis.

1.1 Data Integration and Analyzing

Facilitating the extraction, reformating, and integration of data is very important for companies. Data reshaping programs are difficult to write because of their inherent complexity. However, these programs are highly 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.

Data is analysed 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

[1] Automated Teller Machine

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].

2. Project Overview

Llama is a Data Integration and Analysis System made of two components:

1. Data Integrator

Llama first loads structured plain files. Then data loaded is cleaned, re-formatted and filtered using a series of user-selected transformations. Integration jobs can be specified in two ways: either via (1) a graphical User Interface, or (2) a script written in **IJSL**, for **I**ntegration **J**ob **S**pecification **L**anguage. If the first method is used, at the end, the job can be exported as a IJSL script that can be used in a later run.

2. Data Analyzer

Once ready, the new data is analyzed by means of SQL queries. The results can be stored for possibly further analysis.

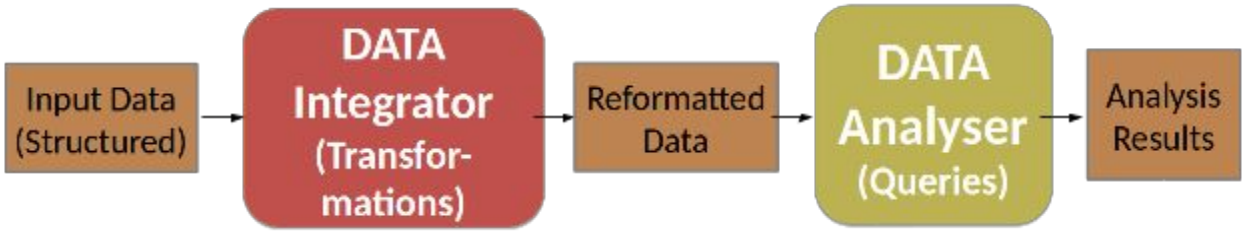

Figure 1: Project Overview: Integration and Analysis tasks

3. Architecture Model

3.1 Architecture Diagram

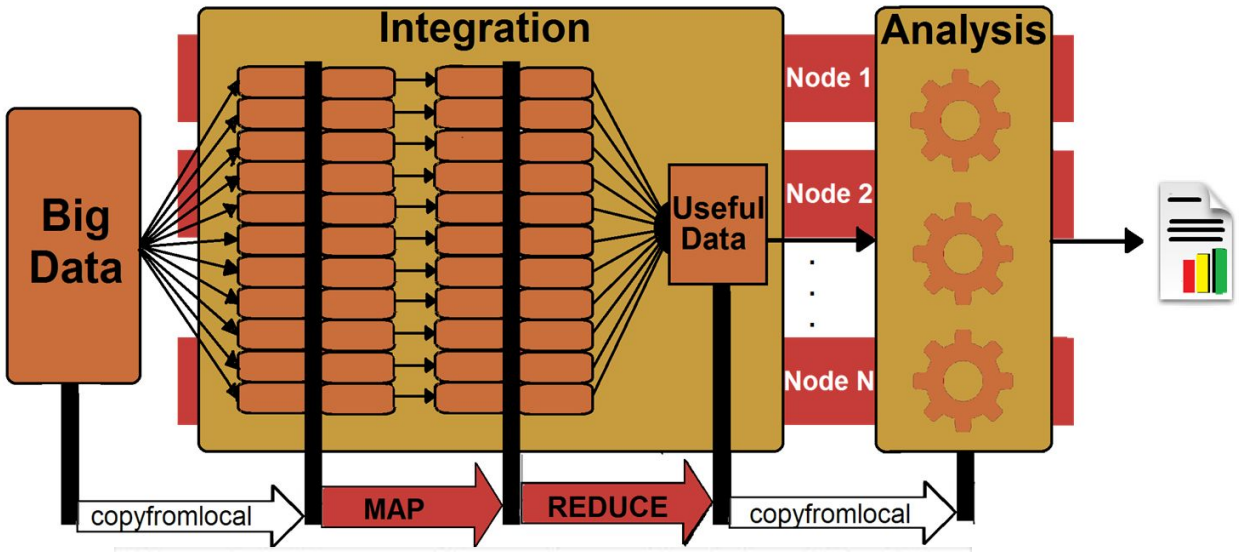

Figure 2: Architecture of Integration and Analyzing of Data [6]

4. Requirements

4.1 Functional Requirements

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 importance 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 the effect 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 i.e 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.

● **References** provides the list of Requirement IDs that we used as references in requirements.

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	H
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

4.2 Non-Functional Requirements

4.2.1 Scalability

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

4.2.2 Performance

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

4.2.3 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.

4.2.4 Distributed Environment

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

5. Implementation

5.1 Apache Hadoop

Apache Hadoop [4] 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.

5.2 Apache Spark

Apache Spark [5] 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...

- ➢ 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.

5.3 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.

5.4 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).

5.5 System Configuration

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

6. Approach

6.1 Data Collection

For development purposes, we create a (Python) script that generates big CSV files. The CSV files are then loaded into the distributed file system.

6.2 Data Integration

We install Apache Hadoop 2.6[2] in Ubuntu 14.4. We set it up for working as a single node and start developing. The language selected was Java. We need to write one Mapper, one Reducer and one Driver. Mapper performs Transformation and Restriction and the Reducer adds the Header into output file. Output files are stored in the Hadoop distributed file system (HDFS). We implement five different types of transformation operations using MapReduce: merge columns, split columns, change the letters case of columns, change formatting of date columns, and one for renaming fields. Additionally, we implement Restriction operations, that have for aim to filter data according to logical operators (1) for numeric values, we have =, <>, >, <, >=, and <=; and for textual values we have EQUAL, NOT EQUAL, CONTAINS.

Examples: Splitting operators can be used to split dates values into days, months and years, in order to be able, in the analysis phase, to group by years or months. Changing the case of strings is useful to normalize the data. Date formatting is needed as per different date standards, like dd/mm/yyyy and mm/dd/yyyy. Merging is required when some information can be viewed in one columns instead of multiple columns, like merging back month and year columns into one column.

6.2.1 Integration Job Specification Language

We propose a language to allow users to describe the integration jobs *declaratively*. We name it *Integration Job Specification Language*, *IJSL* for short.

6.2.1.1 IJSL Grammar

A IJSL script consists of keywords and parameters:

[2] Intallation guide can be found here http://pingax.com/install-hadoop2-6-0-on-ubuntu/

KEYWORDS

There are 10 keywords (which can be written in uppercase or lowercase). For clarity, will write them on uppercase throughout our report:

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

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

Table 2: Some keywords are obligatory and others optional

PARAMETERS

Parameters are defined with the following structure:

Par1|Par2|…

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

```
EMPTY
```

Example: **MERGE** EMPTY

or just don't write that statement.

INPUTFILE

Defines the input CSV file name.

Example: **INPUTFILE** Customers.csv

OUTPUTFILE

Defines the output CSV file name.

Example: **OUTPUTFILE** CustomersOutput.csv

SEPARATOR

Defines the delimiter of the input file. It can only one character.

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

Example: **SEPARATOR** ,

PROJECTEDCOLUMNS

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

Part1: First projected column.

Part2: Second projected column.

…

PartN: N projected column.

Example: **PROJECTEDCOLUMNS** 1|3

PROJECTEDNAMES

Defines the names of the columns to be projected.

Part1: First projected column name.

Part2: Second projected column name.

…

PartN: N projected column name.

Example: **PROJECTEDNAMES** Name|City

[MERGE]

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

Part1: First column index.

Part2: Second column index.

Part3: Merge character.

Example: **MERGE** 0|1|

[SPLIT]

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

Part1: Column index.

Part2: Split character.

Example: **SPLIT** 2|_

[CASE]

Defines the index of column to upper or lower the case

Part1: Column index.

Part2: 0 for UPERCASE, 1 for LOWERCASE

Example: **CASE** 1|0

[FORMAT]

Defines the index of date column to be formatted.

Part1: Column index.

Part2: DD/MM/YYYY MM/DD/YYYY YYYY/MM/DD

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

[RESTRICTION]

Defines the index of column to be restricted (filtered), the operator used and the value.

Part1: Column index.

Part2: For numeral values: =, <>, >,<,>=,<= For textual values: EQUAL, NOT EQUAL, CONTAINS

Part3: value

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

EXAMPLES:

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

A. We want as output 3 columns: Customer_ID, Address, City where Customer_ID values are greater than or equal to 20 and less than 50.

```
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: Customer_ID -> ID, ZipCode -> Postal_Code. And change Name case to capital letters.

```
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
```

6.3 Data Analysis

We install Apache Spark 1.4[3] and Scala 2.10 in Ubuntu 14.4. We set it up for working as a single node and choose Java for programming language. Apache Spark provides an API that allows to deal with structured data called Spark SQL. As its name says, it offers the possibility to query data using SQL. Thus, we are able to select fields and apply aggregation functions and group by fields. We can also join data by specifying the files to join and the join condition on the similar columns. We can add single or multiple where clauses to filter data based on some required conditions.

The input data to the analysis phase is the output that was generated from the integration phase.

7. Implementation

7.1 Virtual Machine Setup

1. On each single machine, we first install Java.
2. Then we install SSH for the 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.

[3] Installation guide can be found here
http://blog.prabeeshk.com/blog/2014/10/31/install-apache-spark-on-ubuntu-14-dot-04/

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.

8. Testing

8.1 Unit Testing

We are working with Java, therefore we will use Junit for unit testing

unit test cases.

In first case: verify the list of the available files in the HDFS. As shown in figure 3.

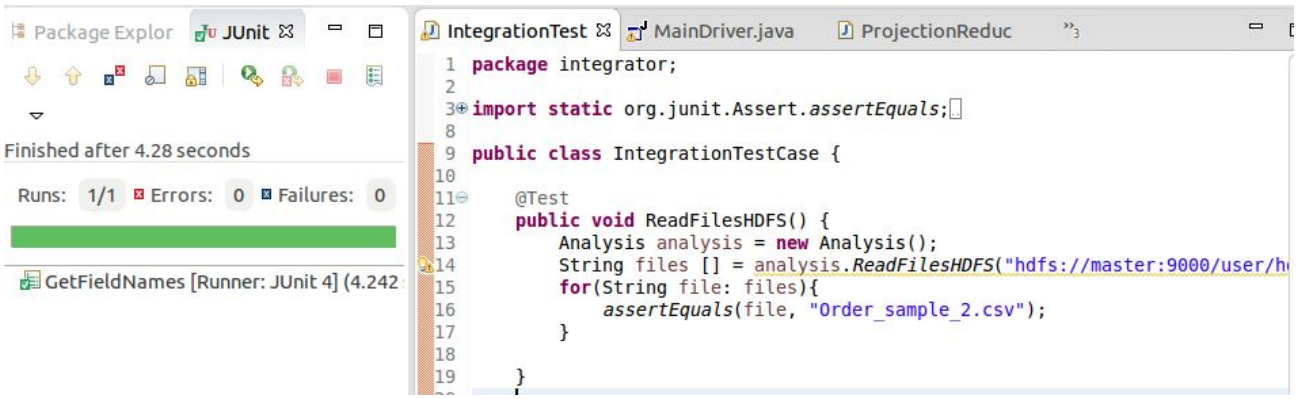

Figure 3: Unit test case1

In second case: verify that all the field names that the file has have been retrieved correctly, as shown in figure 4.

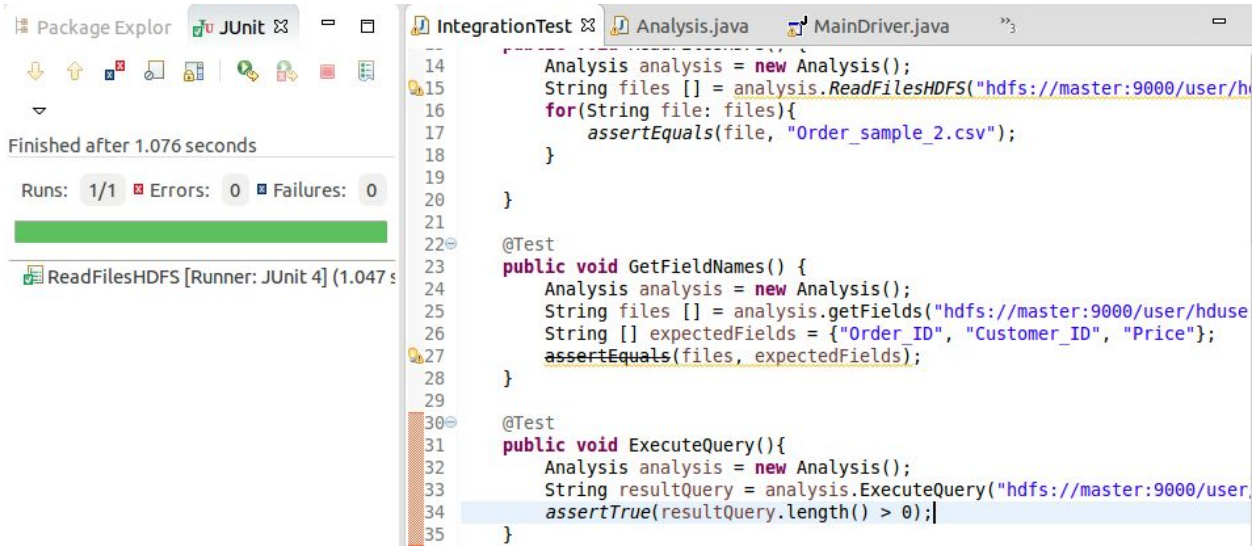

Figure 4: Unit test case2

In third case: verify the execution of the input query, as shown in figure 5.

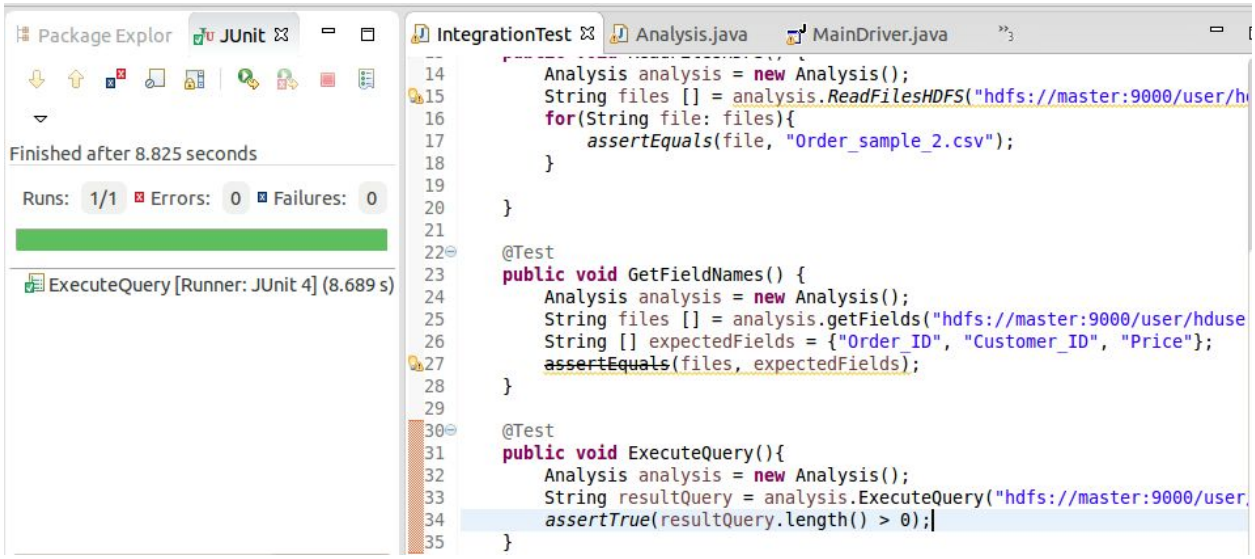

Figure 5: Unit test case3

In fourth case: verify the generation of the output file after the query has been executed, as shown in figure 6.

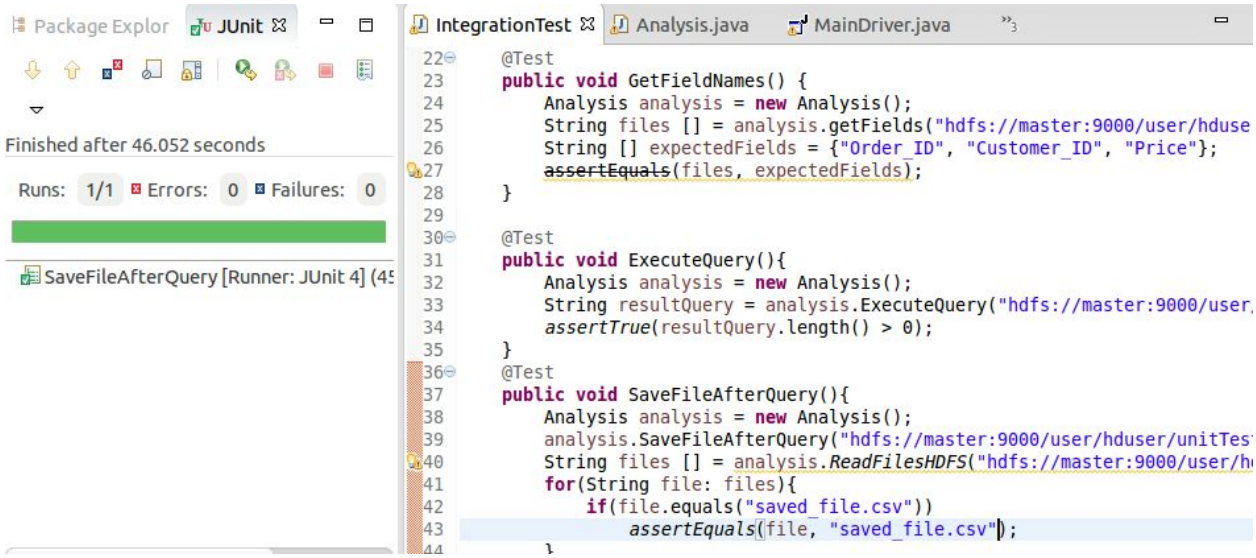

Figure 6: Unit test case4

8.2 Integration Testing

We have tested that the two modules (Integration and Analysis) are working correctly after being integrated in one. Integration functionality works on different test cases to check for merge, split, filter and formatting features. And analysis queries are being executed under only one solution.

8.3 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.

9. Evaluation

While working with 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.

9.1 Environment

- Apache Hadoop 2.6
- Spark 1.4.1
- Ubuntu 14.04
- Java 1.8
- Scala 2.10

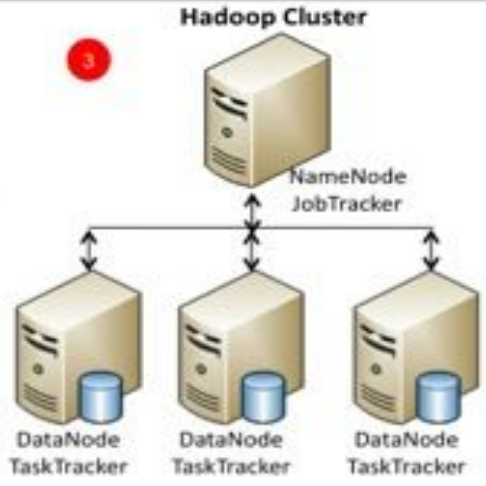

Figure 7: Hadoop Cluster Architecture

NameNode:

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:

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:

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:

Brand and Model	Asus Zenbook UX303L
CPU	Core i7 2.4Ghz
RAM	8 GB
Hard Disk	1000 GB

Table 10: DataNode 3 architecture

9.1.1 Input Testing Files

We have 4 input files:

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

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

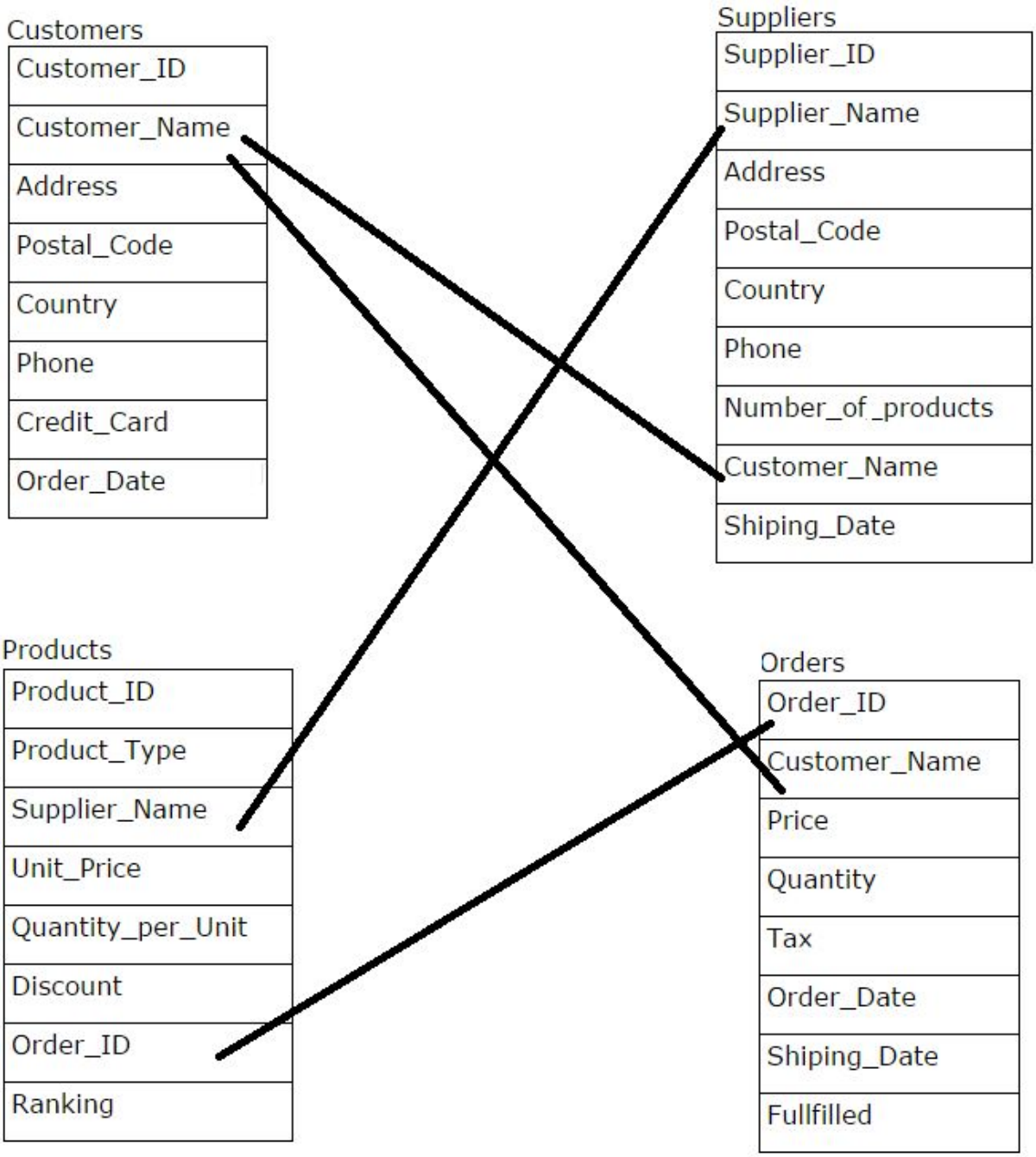

Figure 8: Schema of resource files

9.2 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
```

9.3 Analysis Test Cases

CASE 1: Find for each supplier the number of products delivered, the total Unit price, total Quantity and total Discount, of only products having ranking less than 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 the number of available suppliers by customer

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 the Orders and respective number of Products which are fulfilled and contain more than 10 products.

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
```

9.4 Results

9.4.1 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

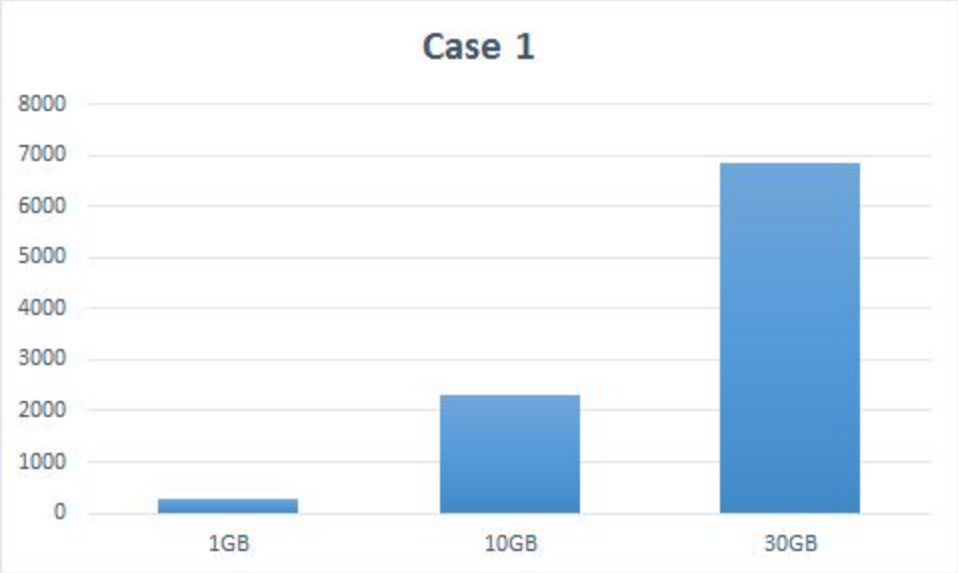

Figure 9: Integration Results for Case1

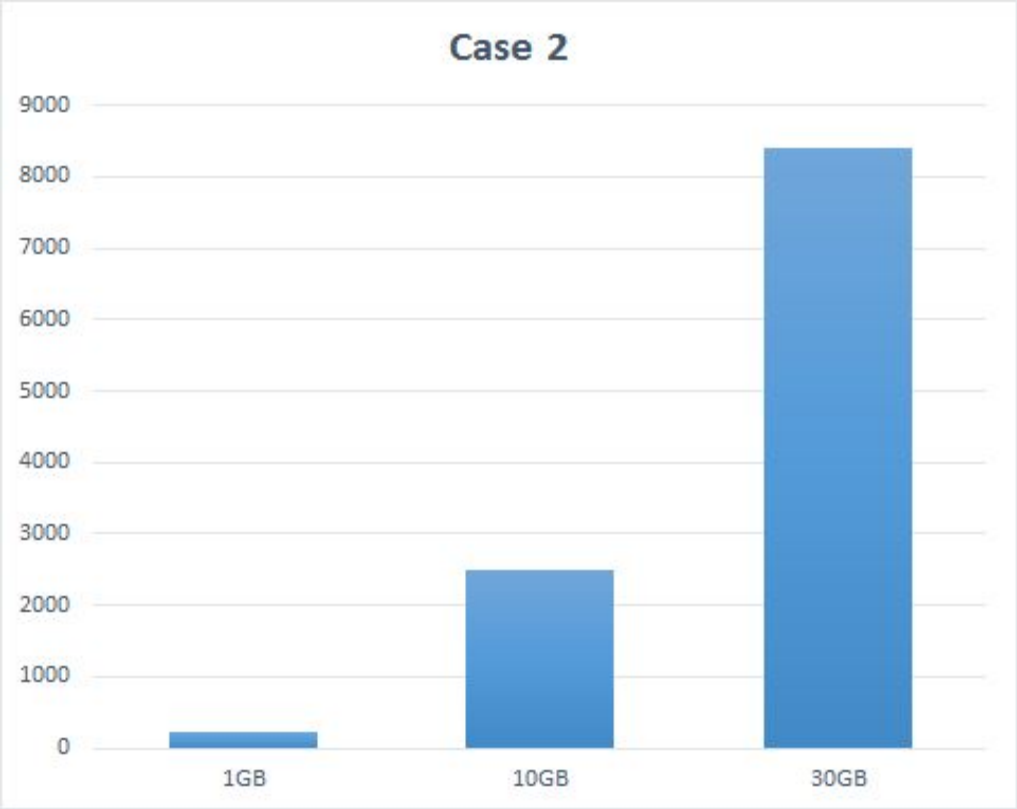

Figure 10: Integration Results for case 2

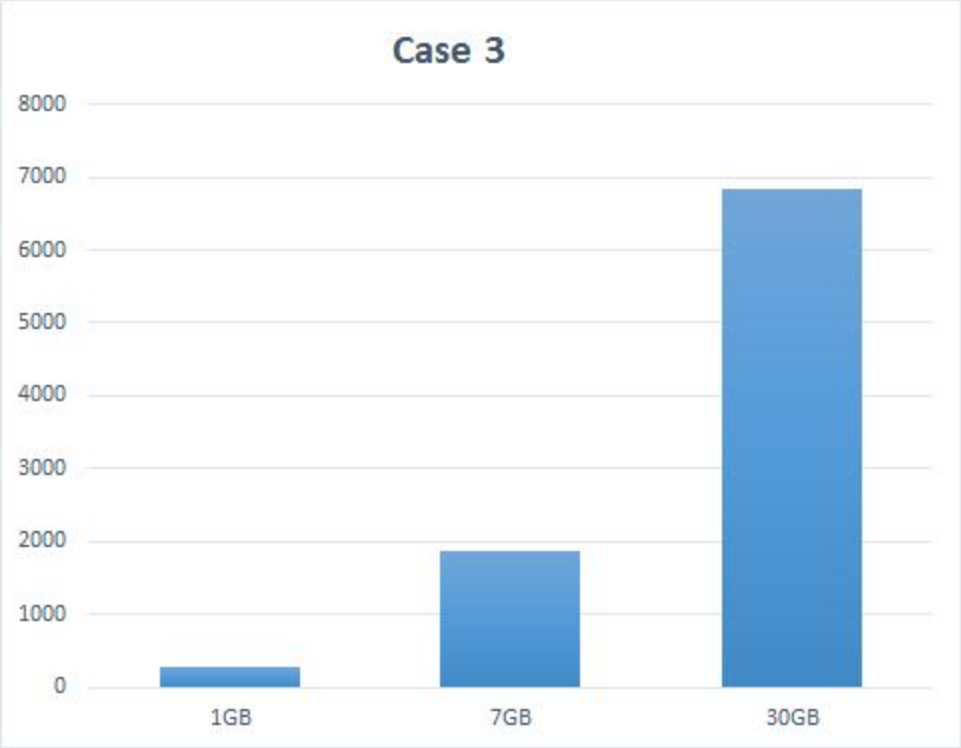

Figure 11: Integration Results for Case 3

9.4.2 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

Figure 12: Analysis Results for case1

Figure 13: Analysis Results for case2

Figure 14: Analysis Results for case3

10. References

[1]: An Oracle White Paper March 2013: Big Data Analytics

[2]: "Why Big Data Is Big Deal." *Harvard Magazine*.18 Feb. 2014. Web. 08 Oct. 2015.

[3]: Group, Big Data Working. *Big Data Analytics for Security Intelligence* (n.d.): n. pag. Web.

[4]: "Hadoop." *Wikipedia*. Wikimedia Foundation, n.d. Web. 08 Oct. 2015.

[5]: "Spark." *Wikipedia*. Wikimedia Foundation, n.d. Web. 08 Oct. 2015.

[6]: “Hadoop Architecture”. Wikimedia Foundation, n.d.Web.08 Oct. 2015.