Benchmarking Apache Spark over MySQL

(using TPC-DS)

Team Members:
Ankush Sharma
Haftamu Hailu
Kaoutar Chennaf
René Gomez

Supervised by:
Esteban Zimányi

Ecole Polytechnique de Bruxelles

Abstract

In a world of constantly prospering big data systems, benchmarking becomes more and more critical for achieving the desired outcomes with the desired performance and preciseness of queries execution, whether it be in corporate or research-oriented environments. When benchmarking big data systems, four requirements remain ubiquitous when considering the most appropriate one for the use-case: volume, velocity, variety and veracity. Spark was one of those revolutionizing big data toolboxes[1] that were designed to accommodate data scientists in their most meticulous data processing problems. Using the famous standardized TPC-DS benchmark, we analyzed the velocity aspect of queries execution on a dataset of 1Gb. This paper summarizes the process and results obtained from this benchmark.

Table of content

1. Executive Summary

Apache Spark is an open source data processing framework. It was born from a simple observation: MapReduce technology is very interesting but as complex (iterated) queries are needed and more real time comes into play, it reaches its limits. Hence the idea of creating a new framework using massive parallelization with in memory technology. Massive parallelization stands for distributing the calculations in a large number of processors or machines depending on the size of your infrastructure, while in memory means simply loading the data in memory. Apache Spark therefore represents an interesting framework that we wanted to test and evaluate in a simulated real-life business environment.

In this paper, we present an evaluation of Apache Spark on MySQL using the benchmark for Decision Support modeled by the Transaction Processing Performance Council (TPC-DS). Our goal is to assess Apache Spark's support of JDBC and evaluate the possibility of improving the performance of BI queries by using Spark to perform the computational part and relying on MySQL only as a data store .

We decided not to use Apache Spark in a distributed setting in order to focus our effort in understanding the data warehouses of the TPC-DS and how to properly use the partition schema of Apache Spark.

We chose this environment mainly because it will allow us to see if it's worthy to have a computation engine over a RDBMS and, at the same time, understand the use cases of Spark SQL. Moreover, since Spark is ubiquitous in the Big Data world, we wanted to evaluate the complexities and challenges that an engineer or analyst could face when implementing Spark in a real organisation. To that end, we decided to use the Scala API provided by Spark.

This document is divided as follows: chapter 2 gives an overview of the TPC-DS benchmark. Chapter 3 explores the abstractions defined by Apache Spark and some key concepts to understand our experiment. In chapter 4, we explain all the steps required to replicate the execution environment. Chapter 5 shows the results of the TPC-DS queries and their execution performance as well as the learnings we got from executing the different partition schemas in Apache Spark. Finally, we present some conclusions and future work.

2. About TPC-DS Benchmark

TPC-DS stands for the Decision-Support benchmark that was designed by the Transaction Processing Performance Council (TPC). In this paper, we refer the to the TPC-DS Standard Specification document (currently in version 2.10.0)[2] to get the definition, purpose and some other aspects defined by the Transaction Processing Performance Council for Decision Support systems.
For any other information regarding the model of the data warehouse, the queries groups or some more details the specification document must be consulted.

2.1 TPC Benchmarks

As defined in the TPC-DS Specification document[2], the purpose of TPC benchmarks is to provide relevant and objective performance measures to industry users. To achieve that purpose, TPC benchmark specifications require benchmark tests to be implemented with systems, products, technologies and pricing that:

I. Are generally available to users;
II. Are relevant to the market segment that the individual TPC benchmark models or represents (e.g., TPC-DS) models and represents complex, high volume data in decision support environments);
III. Would plausibly be implemented by a significant number of users in the market segment modelled or represented by the benchmark.

2.2 What is the TPC-DS?

The TPC Benchmark DS (TPC-DS) is a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance. (TPC-DS Specification).

2.3 Why TPC-DS?

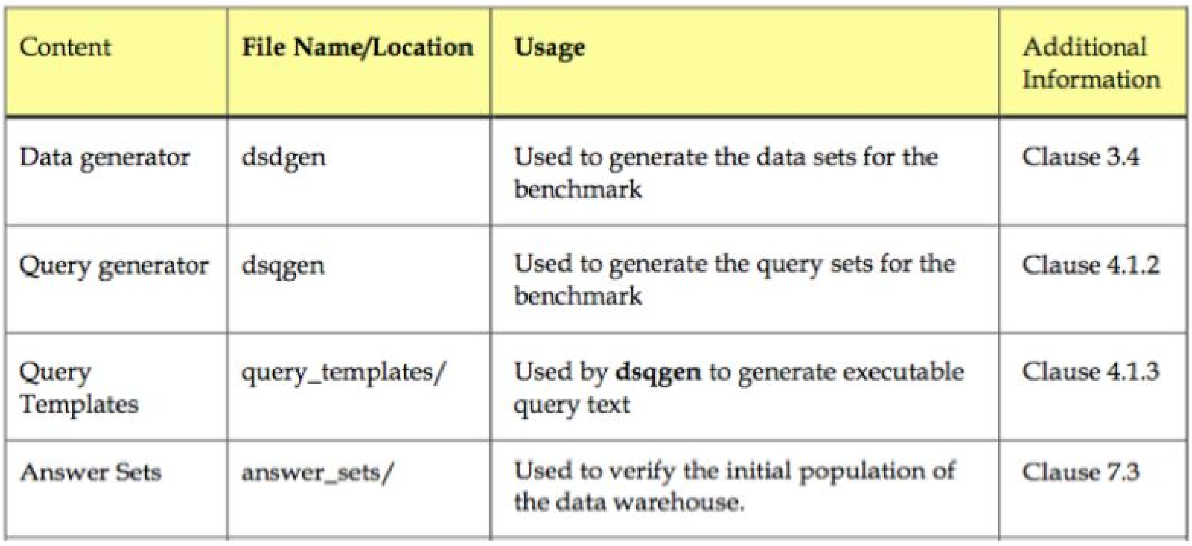
This benchmark illustrates decision support systems that:

- examine large volumes of data;
- give answers to real-world business questions;
- execute queries of various operational requirements and complexities (e.g., ad-hoc, reporting, iterative OLAP, data mining);
- are characterized by high CPU and IO load;
- are periodically synchronized with source OLTP databases through database maintenance functions.
- **run on "Big Data" solutions, such as RDBMS as well as Hadoop/Spark based systems.**

This last highlighted aspect is important for this experiment, where a commonly used RDBMS (MySQL) is evaluated using a Spark based system, Spark SQL to be specific.

2.4 TPC-DS Resources

For this project we used some of the digital resources provided by the TPC-DS.



Content	File Name/Location	Usage	Additional Information
Data generator	dsdgen	Used to generate the data sets for the benchmark	Clause 3.4
Query generator	dsqgen	Used to generate the query sets for the benchmark	Clause 4.1.2
Query Templates	query_templates/	Used by **dsqgen** to generate executable query text	Clause 4.1.3
Answer Sets	answer_sets/	Used to verify the initial population of the data warehouse.	Clause 7.3

Figure 1: Digital components from TPC-DS Standard Specification

In the benchmark setup chapter we will explain how to use these tools for the purpose of this evaluation.

2.5 Business Model

TPC-DS models any industry that must manage, sell and distribute products (e.g., food, electronics, furniture, music and toys etc.). It utilizes the business model of a large retail company having multiple stores located nationwide. Beyond its brick and mortar stores, the company also sells goods through catalogs and the Internet. Along with tables to model the associated sales and returns, it includes a simple inventory system and a promotion system.

2.6 Query Classes

TPC-DS has defined four broad classes of queries that characterize most decision support systems:

- Reporting queries
- Ad hoc queries
- Iterative OLAP queries
- Data mining queries

TPC-DS provides a wide variety of queries in the benchmark to emulate these diverse query classes.

3. Apache Spark

3.1 What is Apache Spark?

Apache Spark is an open-source distributed general-purpose cluster computing framework with (mostly) in-memory data processing engine that can do ETL, analytics, machine learning and graph processing on large volumes of data [3].
But most importantly for the purpose of this project, we can also describe Spark as a distributed, data processing engine for batch and streaming modes featuring SQL queries, graph processing, and machine learning [3][4].

Being an in memory computation engine, Spark can be connected different storage engines or data can be fed to it via streaming. For instance, we can configure Spark to stream data from message queues like Apache Kafka, or process it as a batch on top of external stores like Apache Cassandra, Apache HBase, HDFS, local filesystem, or a relational database.

3.2 Key Concepts

3.2.1 RDD

At the very core of Apache Spark is the concept of RDD, which stands for Resilient Distributed Datasets. RDD's can be thought of as a collection of elements that are partitioned across several machines so that they can be operated upon in parallel. Consider the following code sample:

```
object RDDsSample {

  val cores: Int = Runtime.getRuntime.availableProcessors()
  val numOfPartitions: Int = cores / 2

  def main(args: Array[String]): Unit = {
     val seq = Seq(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
     for(value <- seq) {
       println(value)
     }

    val spark = SparkSession.builder.master("local").getOrCreate
    val sparkContent = spark.sparkContext
    val rddSeq = sparkContent.parallelize(seq, numOfPartitions)
    rddSeq.foreachPartition( partition => {
      for(partition <- partition) {
        println(partition)
      }
    })
    spark.close()
  }
}
```

A Sequence in Scala is just an array of elements. Whereas a Sequence is computed over a single JVM node, RDD's can partition a Sequence based on the the value of numOfPartitions. For instance, if numOfPartitions = 4, then Spark will slice the Sequence into 4 distinct sequences, and distribute it across the cluster.

3.2.2 RDD Operations and Lazy Evaluation

RDD supports two types of operations :

1. Transformations : A transformation can be thought of as a mathematical function, that takes a dataset as input and transforms it into a new dataset. Strictly speaking, a transformation will take in an RDD as input, compute the function on each element of the RDD, and return a new RDD as output. Examples of transformations include map(), filter(), flatMap() etc.
2. Actions[5] : RDD Transformations create new RDDs from an existing RDD, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation; rather actions are RDD operations returns final results of RDD computations that are non-RDD values. The values of action are stored to drivers or to the external storage system. Action exaction in RDD is based on the lineage graph. It brings laziness of RDD into motion. first(), take(), reduce(), collect(), and count() are some of the actions in spark

3.2.3 DataSets and DataFrames

Spark provides two interfaces to work with data :

1. DataSet : A DataSet is an abstraction for data that is distributed across the cluster. It can be instantiated using JVM objects and then manipulated using functional transformations such as map(), filter(), flatMap() etc. The DataSet API is available only in Scala and Java. The API is not supported for Python and R.
2. DataFrame : A Dataframe in essence is a DataSet, but with named columns. They can be thought of as the equivalent of a Table in relational databases. When we load a table from MySQL using JDBC, the table is represented as a DataFrame object by Spark.

Let's try to understand by way of an example:

```
object DataFrameSample {

  def main(args: Array[String]): Unit = {

    val sparkConf = new SparkConf().setAppName("DB Warehouse Project SQL")
    val spark = SparkSession.builder.config(sparkConf).master("local").getOrCreate

    val mysqlConnProperties = new Properties()
    mysqlConnProperties.setProperty("user", "root")
    mysqlConnProperties.setProperty("password", "root")

    val db = "tpc-ds-db-warehouse-project"
    val url = "jdbc:mysql://localhost:3306/" + db
    val catalogPage = "catalog_page"
    val catalogPageTable = spark.read.jdbc(url, catalogPage, mysqlConnProperties)

    catalogPageTable.foreach( row => {
      val csCatalogPageSk = row.getAs[Int]("cp_catalog_page_sk")
      val cpDesc = row.getAs[String]("cp_description")
      println(csCatalogPageSk)
      println(cpDesc)
    })

    logger.info("Stopping Apache Spark")
    spark.close()
  }
}
```

We initialized Spark, and made a connection to MySQL through JDBC, which is a low level API on the JVM to access relational databases like MySQL. Under the hood, JDBC maintains a thread pool of its own, where each thread corresponds to a connection to MySQL. For instance, a thread pool of 100 JDBC threads means that the application has 100 connections with MySQL.

Every table in MySQL corresponds to a DataFrame in Spark. The jdbc() function accepts the url of the database, the name of the table, and additional properties to connect with MySQL. Every element in a DataFrame can be thought of as a “Tuple” or a “Row” belonging to a table in MySQL. Moreover, every element in a Row can be thought of as a “Column” belonging to that row. In our example, we know beforehand that there is a table called “catalog_page” and this table has columns "cp_catalog_page_sk" and "cp_description". We can see that DataFrame API provides methods to access “columns” in a “row”.

3.2.4 Spark SQL

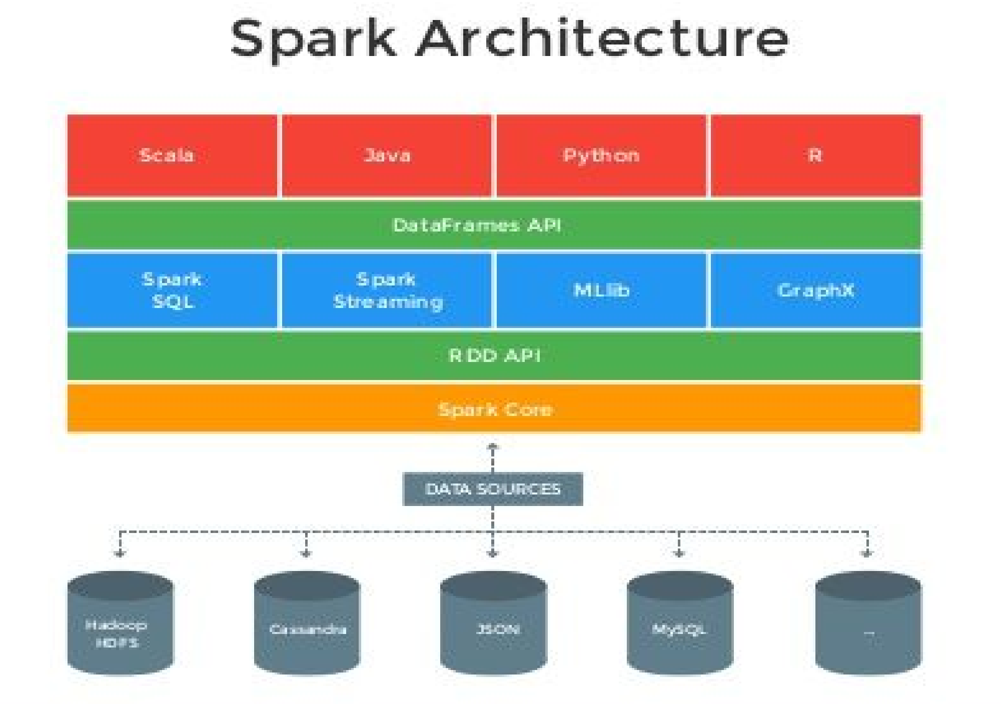


Figure 2: Spark Architecture. Source.

Spark SQL is a library on top of Spark Core and RDD API[6]. It provides SQL interfaces for Data Warehousing applications. There are two ways to work with Spark SQL. One is through spark-shell, which is a console based application to analyze data interactively. The other is to work directly with JDBC using one out of Scala, Java, Python or R API's.

Especifically for this project we wrote the code in Scala to connect to MySQL using JDBC, retrieve the data stored in the data warehouse and then, using Spark SQL, execute one by one the 99 queries of the TPC-DS benchmark. Furthermore, we could extend this solution to other RDBMS that supports JDBC connection, just changing the parametrization for database connection. We will give the details in the next chapter.

3.2.6 Partitions and Partitioning

Recall that DataFrame is a distributed collection of records spread across a cluster, upon which we can perform certain transformations and actions. In order to partition a DataFrame, Spark SQL mandates that the following parameters be provided :

Property name	Description
url	The jdbc url to connect to. For MySQL running on localhost and port 3306, the url is `jdbc:mysql://localhost:3306/db`
dbtable	The name of the table in MySQL that will be represented

	as a DataFrame in Spark
connectionProperties	Key value pairs required by JDBC to connect to MySQL. The mandatory ones are : "user" and "password"
partitionColumn, lowerBound, upperBound	These are perhaps the most important parameters. partitionColumn is the column that will be used to partition the table. This must be a numeric column. lowerBound and upperBound are used to calculate a partitionStride, which we will see below
numPartitions	The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections

The parameters partitionColumn, lowerBound, upperBound and numPartitions are all used to calculate the partition stride. The formula is :

`partitionStride` = (`upperBound - lowerBound) / numPartitions [7]`

For example, assume that :

- lowerBound : 0
- upperBound : 10000
- numPartitions : 10

Then, partitionStride = (10000 - 0) / 10 = 1000

One of the interesting things that Spark SQL does is that it can translates certain parts of a query and can "push down" certain conditions to MySQL. A SQL query contains WHERE clauses followed by multiple AND, OR, IN conditions, followed by GROUP BY, ORDER BY, and other aggregations etc. When Spark is given an instruction to compute an SQL Query against MySQL, it will only send the conditions in the WHERE clause to MySQL and compute other aggregates like GROUP BY, ORDER BY etc **in memory**. In other words, it will query MySQL for the data records only, complex calculations will be executed inside the Spark cluster itself.

Building up on the idea presented above, we will now try to model a very simple query. Our query, when executed in MySQL goes something like this : `SELECT * FROM table GROUP BY column1`

Using the parameters defined above, Spark will compute partitionStrides, and then each partition will correspond to following queries :

```
SELECT * FROM table WHERE partitionColumn BETWEEN 0 AND 1000
SELECT * FROM table WHERE partitionColumn BETWEEN 1000 AND 2000
SELECT * FROM table WHERE partitionColumn BETWEEN 2000 AND 3000
.
.
```

```
.
SELECT * FROM table WHERE partitionColumn BETWEEN 9000 AND 10000
SELECT * FROM table WHERE partitionColumn > 10000
```

Notice that the GROUP BY clause was excluded by Spark. In this example, Spark will load data into 10 partitions, and then perform the GROUP BY aggregation.

4. Benchmark setup

4.1 Downloading TPC-DS Tools

We need the tools to create the data at different scale factors. This step is also required to get the answer set and the structure of the database.

You can follow the instructions provided in the TPC-DS How To Guide.

If you are using MacOS, we recommend to use this version of the TPC-DS tools, which allows you to compile without errors.

You can also download the official tools from the TPC website.

4.2 Generating the database

- Generating 1GB database

For generating 1GB data use the following command:

```
$./dsdgen -scale 1 -dir ../../data1GB/
```

- Generating 5GB database

For generating 5GB data use the following command:

```
$./dsdgen -scale 5 -dir ../../data5GB/
```

4.3 Install and run MySQL

Now, it's necessary to create the database to store the data. Install MySQL in your machine.
For MacOS open Terminal and execute the following command to install MySQL using Homebrew:

```
$brew install mysql
```

Execute the following command to set the root password:

```
$mysqladmin -u root password 'yourpassword'
```

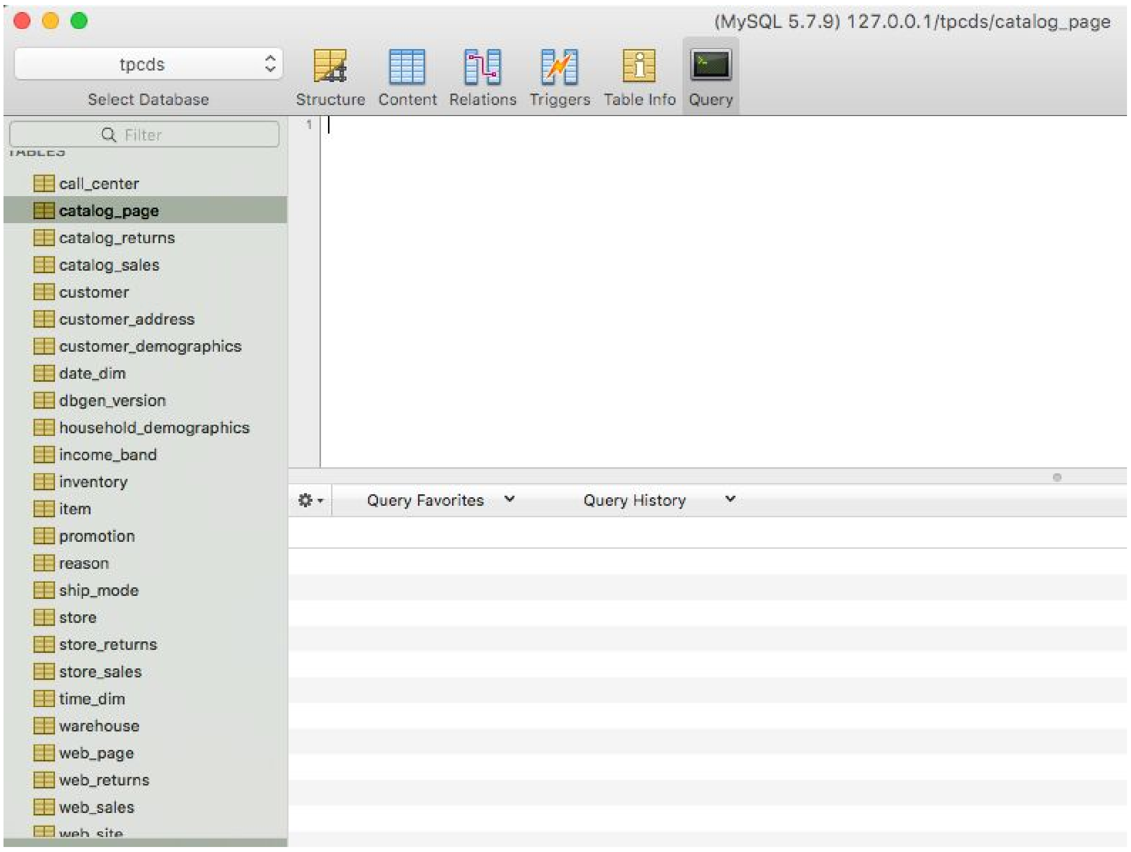
Execute the following command to start the server:

```
$mysql.server start
```

To manage the databases we recommend using Sequel Pro, a MySQL management tool designed for macOS. You can use other management tools according to your operative system.

Create the databases, for instance you can use the names tpcds and tpcds5gb for 1GB and 5GB datasets. In each database create the data warehouse tables using the file *tpcds.sql* provided in the TPC DS tools.

For this step, you can use the management tool for MySQL. Open Sequel Pro, copy & paste the content of *tpcds.sql* in a query under the selected database, run the query to create the structure of the data warehouse. You should be able to see all the tables.



call_center
catalog_page
catalog_returns
catalog_sales
customer
customer_address
customer_demographics
date_dim
dbgen_version
household_demographics
income_band
inventory
item
promotion
reason
ship_mode
store
store_returns
store_sales
time_dim
warehouse
web_page
web_returns
web_sales
web_site

Figure 3: Data warehouse structure in MySQL

4.4 Loading the data into MySQL

For doing so, run the loader provided with your DBMS (Sequel Pro in this case) and load the *dsdgen* generated data files into the data warehouse tables. Keep in mind the following notes:

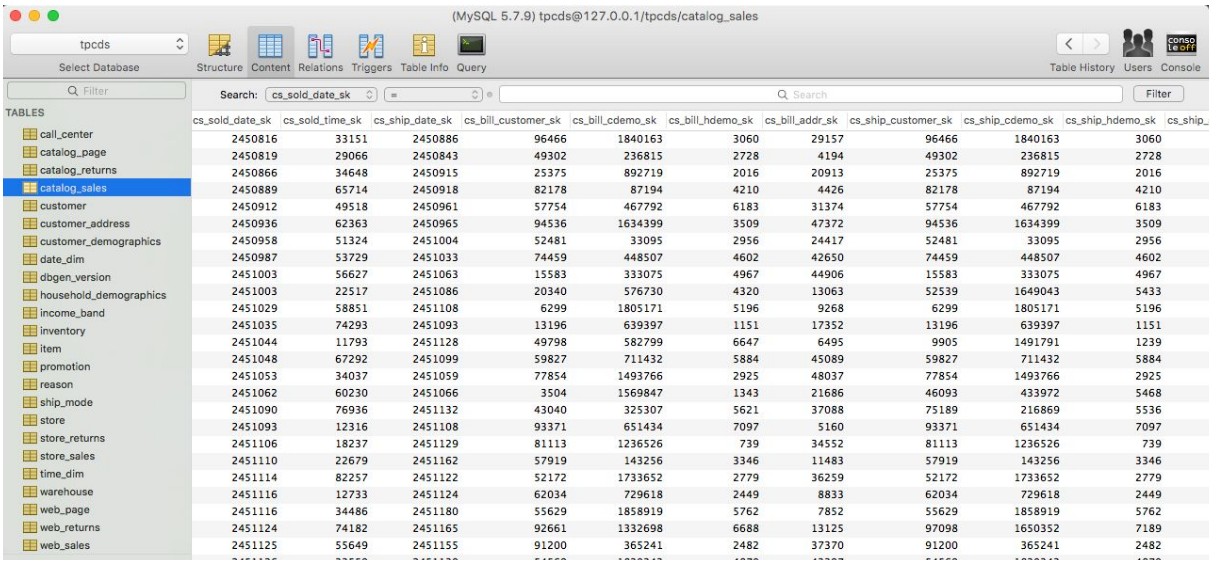
- Note that the default delimiter is '|' so you may need to specify a different delimiter with dsdgen or the loader if the defaults don't match.
- Also, the default "null" value is "||" so if your loader expects (for example), "|NULL|", then you will need to override the loader's value for nulls.

An alternative is to use a script to load all the data. We followed this instructions to load the data with the following script. Modify <datafolder>,<database>, <user> and <password> to run it in your machine. If you are in a Unix-based system you can use the same script.

This script is also provided in the files attached to this document.

```
DIR=./<datafolder>
ls $DIR/*.dat | while read file; do
    pipe=$file.pipe
    mkfifo $pipe
    table=`basename $file .dat | sed -e 's/_[0-9]_[0-9]//'`
    echo $file $table
    LANG=C && sed -e 's_^|_\\N|_g' -e 's_||_|\\N|_g' -e 's_||_|\\N|_g' $file > $pipe & \
    mysql <database> -u<user> -p<password> --local-infile -Dtpcds -e \
          "load data local infile '$pipe' replace into table $table character set latin1
fields terminated by '|'"
    rm -f $pipe
done
```

If everything is setup correctly, you should be able to browse the content of the data warehouse. Here is an example of the 1GB dataset.



cs_sold_date_sk	cs_sold_time_sk	cs_ship_date_sk	cs_bill_customer_sk	cs_bill_cdemo_sk	cs_bill_hdemo_sk	cs_bill_addr_sk	cs_ship_customer_sk	cs_ship_cdemo_sk	cs_ship_hdemo_sk	cs_ship_
2450816	33151	2450886	96466	1840163	3060	29157	96466	1840163	3060	
2450819	29066	2450843	49302	236815	2728	4194	49302	236815	2728	
2450866	34648	2450915	25375	892719	2016	20913	25375	892719	2016	
2450889	65714	2450918	82178	87194	4210	4426	82178	87194	4210	
2450912	49518	2450961	57754	467792	6183	31374	57754	467792	6183	
2450936	62363	2450965	94536	1634399	3509	47372	94536	1634399	3509	
2450958	51324	2451004	52481	33095	2956	24417	52481	33095	2956	
2450987	53729	2451033	74459	448507	4602	42650	74459	448507	4602	
2451003	56627	2451063	15583	333075	4967	44906	15583	333075	4967	
2451003	22517	2451086	20340	576730	4320	13063	52539	1649043	5433	
2451029	58851	2451108	6299	1805171	5196	9268	6299	1805171	5196	
2451035	74293	2451093	13196	639397	1151	17352	13196	639397	1151	
2451044	11793	2451128	49798	582799	6647	6495	9905	1491791	1239	
2451048	67292	2451099	59827	711432	5884	45089	59827	711432	5884	
2451053	34037	2451059	77854	1493766	2925	48037	77854	1493766	2925	
2451062	60230	2451066	3504	1569847	1343	21686	46093	433972	5468	
2451090	76936	2451132	43040	325307	5621	37088	75189	216869	5536	
2451093	12316	2451108	93371	651434	7097	5160	93371	651434	7097	
2451106	18237	2451129	81113	1236526	739	34552	81113	1236526	739	
2451110	22679	2451162	57919	143256	3346	11483	57919	143256	3346	
2451114	82257	2451122	52172	1733652	2779	36259	52172	1733652	2779	
2451116	12733	2451124	62034	729618	2449	8833	62034	729618	2449	
2451116	34486	2451180	55629	1858919	5762	7852	55629	1858919	5762	
2451124	74182	2451165	92661	1332698	6688	13125	97098	1650352	7189	
2451125	55649	2451155	91200	365241	2482	37370	91200	365241	2482	

Figure 4: Data warehouse content loaded in MySQL

4.5 Generating the queries

We used the queries provided here by IBM open source to benchmark SparkSQL. Again, you can use the same queries to benchmark any underlying database supported by JDBC, MySQL in this case. These queries are included in the project provided with this document, you don't need to download them again.

If for any other reason you need to create the queries using other dialect then you can use the *dsqgen* utility to generate all the queries with the following command:

```
$./dsqgen -VERBOSE Y -DIALECT netezza -input
../../mysqlQueries/templates.lst -DIRECTORY ../../mysqlQueries/ -QUALIFY
-SCALE 1 -OUTPUT_DIR ../../queries 1GB/
```

From the *TPC DS - How To Guide*:
The "dsqgen" utility is used to transform the query templates into executable SQL for your target DBMS. The unmodified templates are not executable.

The following "dialect templates" are supported: db2.tpl, netezza.tpl, oracle.tpl, sqlserver.tpl.

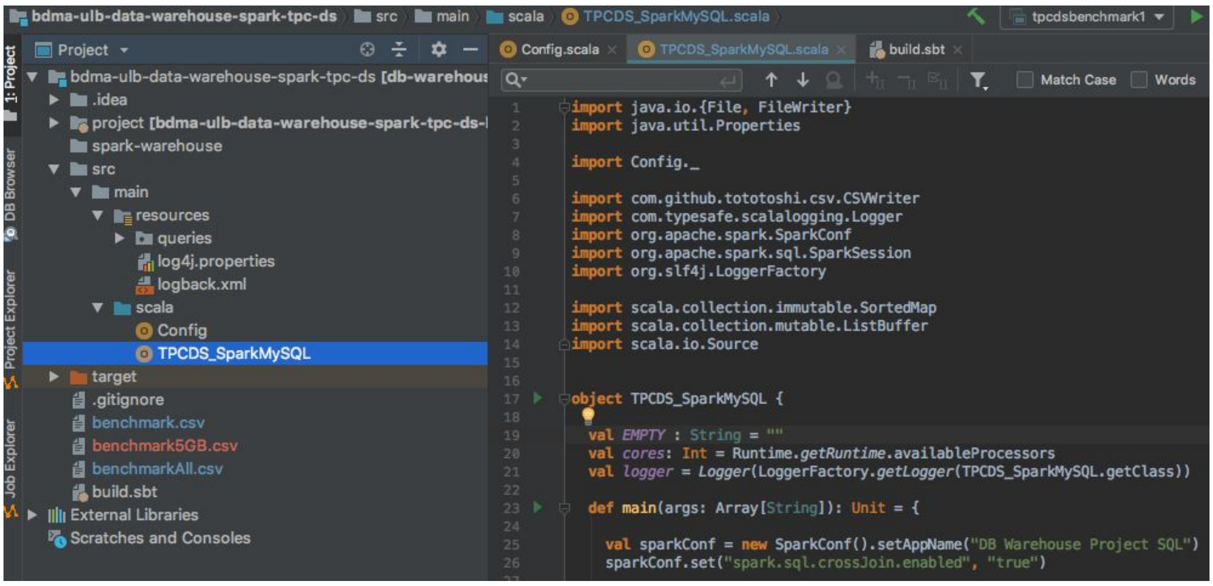
4.6 Scala project

First, we recommend to install IntelliJ IDEA in your operative system. [You can follow the instructions provided here]().

Now, you are ready to use the project we provide with this document. You can also clone the repository from [GitHub, click here.]()

Open IntelliJ, go to File-> Open, look for the location of the code and select the root folder *bdma-ulb-data-warehouse-spark-tpc-ds.* IntelliJ is going to take some minutes to setup all the environment.

Once it's done, you are ready to run the project. Here is an overview.



```
bdma-ulb-data-warehouse-spark-tpc-ds [db-warehous
  .idea
  project [bdma-ulb-data-warehouse-spark-tpc-ds-
  spark-warehouse
  src
    main
      resources
        queries
        log4j.properties
        logback.xml
      scala
        Config
        TPCDS_SparkMySQL
  target
  .gitignore
  benchmark.csv
  benchmark5GB.csv
  benchmarkAll.csv
  build.sbt
External Libraries
Scratches and Consoles
```

```scala
import java.io.{File, FileWriter}
import java.util.Properties

import Config._

import com.github.tototoshi.csv.CSVWriter
import com.typesafe.scalalogging.Logger
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
import org.slf4j.LoggerFactory

import scala.collection.immutable.SortedMap
import scala.collection.mutable.ListBuffer
import scala.io.Source


object TPCDS_SparkMySQL {

  val EMPTY : String = ""
  val cores: Int = Runtime.getRuntime.availableProcessors
  val logger = Logger(LoggerFactory.getLogger(TPCDS_SparkMySQL.getClass))

  def main(args: Array[String]): Unit = {

    val sparkConf = new SparkConf().setAppName("DB Warehouse Project SQL")
    sparkConf.set("spark.sql.crossJoin.enabled", "true")
```

<u>Figure 5:</u> Overview Scala Project

We shall now talk about the code that we used to parse SQL queries and execute them in Spark. We first place all our configurations and table names etc in one Scala file.

```scala
object Config {

    //Global Config
    val USERNAME : String = "tpcds"
    val PASSWORD : String = "TPCds2018"
    val DB : String = "tpcds5gb"
    val URL : String = "jdbc:mysql://localhost:3306/" + DB

    //Table names
    val CALL_CENTER : String = "call_center"
    val CATALOG_PAGE : String = "catalog_page"
    val CATALOG_RETURNS : String = "catalog_returns"
    val CATALOG_SALES : String = "catalog_sales"
    val CUSTOMER : String = "customer"
    val CUSTOMER_ADDRESS : String = "customer_address"
    val CUSTOMER_DEMOGRAPHICS : String = "customer_demographics"
    val DATE_DIM : String = "date_dim"
    val DBGEN_VERSION : String = "dbgen_version"
    val HOUSEHOLD_DEMOGRAPHICS : String = "household_demographics"
    val INCOME_BAND : String = "income_band"
    val INVENTORY : String = "inventory"
    val ITEM : String = "item"
    val PROMOTION : String = "promotion"
    val REASON : String = "reason"
    val SHIP_MODE : String = "ship_mode"
    val STORE : String = "store"
    val STORE_RETURNS : String = "store_returns"
    val STORE_SALES : String = "store_sales"
    val TIME_DIM : String = "time_dim"
    val WAREHOUSE : String = "warehouse"
    val WEB_PAGE : String = "web_page"
    val WEB_RETURNS : String = "web_returns"
    val WEB_SALES : String = "web_sales"
    val WEB_SITE : String = "web_site"

    //Partitioning Keys
    val CATALOG_SALES_PARTITIONING_KEY = "cs_item_sk"
    val CUSTOMER_PARTITIONING_KEY = "c_customer_sk"
    val CUSTOMER_DEMOGRAPHICS_PARTITIONING_KEY = "cd_demo_sk"
    val CUSTOMER_ADDRESSES_PARTITIONING_KEY = "ca_address_sk"
    val STORE_RETURNS_PARTITIONING_KEY = "sr_item_sk"
    val STORES_SALES_PARTITIONING_KEY = "ss_item_sk"
    val INVENTORY_PARTITIONING_KEY = "inv_item_sk"
    val WEB_SALES_PARTITIONING_KEY = "ws_item_sk"
    val WEB_RETURNS_PARTITIONING_KEY = "wr_item_sk"
    val DATE_DIM_PARTITIONING_KEY = "d_date_sk"
    val ITEM_PARTITIONING_KEY = "i_item_sk"
    val TIME_PARTITIONING_KEY = "t_time_sk"

}
```

```
import Config._

object TpcDsBenchmark {

  val EMPTY : String = ""
  val cores: Int = Runtime.getRuntime.availableProcessors

  val logger = Logger(LoggerFactory.getLogger(TpcDsBenchmark.getClass))
  val parallelismLevel = cores * 50

  def main(args: Array[String]): Unit = {

    val sparkConf = new SparkConf().setAppName("DB Warehouse Project SQL")
    sparkConf.set("spark.sql.crossJoin.enabled", "true")

    val spark = SparkSession.builder.config(sparkConf).master("local").getOrCreate // -----> 1

    logger.info("Attempting to Start Apache Spark")
    val mysqlConnProperties = new Properties()
    mysqlConnProperties.setProperty("user", USERNAME)
    mysqlConnProperties.setProperty("password", PASSWORD)

    //We use partitioning only for big tables
    val callCenterTable = spark.read.jdbc(URL, CALL_CENTER, mysqlConnProperties)
    val catalogPageTable = spark.read.jdbc(URL, CATALOG_PAGE, mysqlConnProperties)
    val catalogReturnsTable = spark.read.jdbc(URL, CATALOG_RETURNS, mysqlConnProperties)
    val catalogSalesTable = spark.read.jdbc(URL, CATALOG_SALES, CATALOG_SALES_PARTITIONING_KEY,
                                             10000, 2000000, parallelismLevel ,
                                             mysqlConnProperties) // ------> 2

    val customerTable = spark.read.jdbc(URL, CUSTOMER, mysqlConnProperties)
    val dateTimTable = spark.read.jdbc(URL, DATE_DIM, mysqlConnProperties)

    val customAddressTable = spark.read.jdbc(URL, CUSTOMER_ADDRESS, mysqlConnProperties)
    val customerDemographicsTable = spark.read.jdbc(URL,
                                 CUSTOMER_DEMOGRAPHICS, CUSTOMER_DEMOGRAPHICS_PARTITIONING_KEY,
                                 10000, 100000, parallelismLevel,
                                 mysqlConnProperties)

    val dbGenVersionTable = spark.read.jdbc(URL, DBGEN_VERSION, mysqlConnProperties)
    val houseHoldDemographicsTable = spark.read.jdbc(URL, HOUSEHOLD_DEMOGRAPHICS,
                                                     mysqlConnProperties)

    val incomeBandTable = spark.read.jdbc(URL, INCOME_BAND, mysqlConnProperties)
    val inventoryTable = spark.read.jdbc(URL,
                                  INVENTORY, INVENTORY_PARTITIONING_KEY,
                                  10000, 200000, parallelismLevel,
                                  mysqlConnProperties)

    val itemTable = spark.read.jdbc(URL, ITEM, mysqlConnProperties)
    val promotionTable = spark.read.jdbc(URL, PROMOTION, mysqlConnProperties)
    val reasonsTable = spark.read.jdbc(URL, REASON, mysqlConnProperties)
    val shipModeTable = spark.read.jdbc(URL, SHIP_MODE, mysqlConnProperties)
    val storeTable = spark.read.jdbc(URL, STORE, mysqlConnProperties)
    val storeReturnsTable = spark.read.jdbc(URL, STORE_RETURNS, mysqlConnProperties)
    val storesSalesTable = spark.read.jdbc(URL, STORE_SALES, STORES_SALES_PARTITIONING_KEY,
                                            10000, 2000000, parallelismLevel ,
                                            mysqlConnProperties)
    val timeDimTable = spark.read.jdbc(URL, TIME_DIM, TIME_PARTITIONING_KEY,  10000, 2000000,
 parallelismLevel , mysqlConnProperties)
    val webPageTable = spark.read.jdbc(URL, WEB_PAGE, mysqlConnProperties)
    val webReturnsTable = spark.read.jdbc(URL, WEB_RETURNS, mysqlConnProperties)
    val warehouseTable = spark.read.jdbc(URL, WAREHOUSE, mysqlConnProperties)
    val webSalesTable = spark.read.jdbc(URL, WEB_SALES, WEB_SALES_PARTITIONING_KEY, 10000,
2000000, parallelismLevel, mysqlConnProperties)
    val webSiteTable = spark.read.jdbc(URL, WEB_SITE, mysqlConnProperties)

    callCenterTable.createOrReplaceTempView(CALL_CENTER) // ------> 3
    catalogPageTable.createOrReplaceTempView(CATALOG_PAGE)
    catalogReturnsTable.createOrReplaceTempView(CATALOG_RETURNS)
    catalogSalesTable.createOrReplaceTempView(CATALOG_SALES)
    customerTable.createOrReplaceTempView(CUSTOMER)
    customAddressTable.createOrReplaceTempView(CUSTOMER_ADDRESS)
    customerDemographicsTable.createOrReplaceTempView(CUSTOMER_DEMOGRAPHICS)
    dateTimTable.createOrReplaceTempView(DATE_DIM)
    dbGenVersionTable.createOrReplaceTempView(DBGEN_VERSION)
    houseHoldDemographicsTable.createOrReplaceTempView(HOUSEHOLD_DEMOGRAPHICS)
    incomeBandTable.createOrReplaceTempView(INCOME_BAND)
    inventoryTable.createOrReplaceTempView(INVENTORY)
    itemTable.createOrReplaceTempView(ITEM)
    promotionTable.createOrReplaceTempView(PROMOTION)
    reasonsTable.createOrReplaceTempView(REASON)
    shipModeTable.createOrReplaceTempView(SHIP_MODE)
    storeTable.createOrReplaceTempView(STORE)
    storeReturnsTable.createOrReplaceTempView(STORE_RETURNS)
    storesSalesTable.createOrReplaceTempView(STORE_SALES)
    timeDimTable.createOrReplaceTempView(TIME_DIM)
    webPageTable.createOrReplaceTempView(WEB_PAGE)
    webReturnsTable.createOrReplaceTempView(WEB_RETURNS)
    warehouseTable.createOrReplaceTempView(WAREHOUSE)
    webSalesTable.createOrReplaceTempView(WEB_SALES)
    webSiteTable.createOrReplaceTempView(WEB_SITE)

    val sparkSqlContext = spark.sqlContext

    val sqlQueriesMap = SortedMap(getQueriesMap.toSeq.sortBy(_._1):_*) // ------> 4
    val benchmarkStatistics =  ListBuffer[List[String]]()

    for((index, sqlQueries) <- sqlQueriesMap) { // --------> 5
       val queryNo = index + 1
       logger.info ("Executing Query {}", queryNo)
       val start = System.currentTimeMillis()

        for(sqlQuery <- sqlQueries) {
          val dataFrame = sparkSqlContext.sql(sqlQuery)
          dataFrame.show(1000000)
        }

        val stop = System.currentTimeMillis()
        val timeTakenMs = stop-start
        val timeTakenSeconds = toSeconds(timeTakenMs)
        benchmarkStatistics += List(queryNo.toString, parallelismLevel.toString,
                                   timeTakenSeconds.toString)
        appendToCsv(queryNo + ",  " + cores + ",  " + parallelismLevel + ", " +
                                   timeTakenSeconds.toString()+ "\n")
        logger.info ("Time taken for Query {} : Milliseconds : {}, Seconds : {}", queryNo,
                                   timeTakenMs, timeTakenSeconds )
    }
    writeToCsv(benchmarkStatistics)
    logger.info("Stopping Apache Spark")
    spark.stop() // ------> 6
  }


  private def getQueriesMap : Map[Int, Seq[String]] = {
    val queriesDirectory = new File(getClass.getResource("/queries").getFile)
    queriesDirectory.listFiles()
        .map(file => Source.fromFile(file).getLines.toList)
        .map(lines => {
          val stringBuilder = new StringBuilder
          //Queries with comments were causing problems. Therefore, all comments are ignored
when building the sql query
          for(line <- lines if !isSqlComment(line)) {
            stringBuilder.append(line).append("\n")
          }
          //The last new line character was also creating a problem, therefore we strip the sql
query of a newline character at the very end
          stringBuilder.toString.stripSuffix("\n")
        })
        //We have few cases where a single file has multiple sql queries. If we have any string
that is not whitespace after a ';', then we confirm that we have multiple queries
        .map(query =>  if(query.contains(";")) query.split(";").toSeq else Seq(query))
        //We also need to strip every sql query of a ';'. This is done because Spark throws an
error if there is a ';' present at end of query
        .map(queries => {
          queries.map(query => if(query.contains(";")) query.replaceAll(";", EMPTY).trim else
query.trim)
        })
       .zipWithIndex
       .map({
          case (k, v) => (v, k)
       })
      .toMap
  }

  private def writeToCsv(records : ListBuffer[List[String]]) : Unit = {
    val file = new File("./benchmarkAll.csv")
    if (!file.exists) {
      file.createNewFile
    }
    val writer = CSVWriter.open(file)
    writer.writeAll(records.toList)
  }
  private def appendToCsv(records : String) : Unit = {


    val fw = new FileWriter("./benchmark.csv", true)
    try {
      fw.write(records)
    }
    finally fw.close()
  }

  private def isSqlComment(line : String) : Boolean = {
    if(line.startsWith("--")) true else false
  }

  private def toSeconds(ms : Long) : Int = {
    (ms / 1000).toInt
  }

}
```

We start by placing all of our queries in */queries* directory. All the 99 queries to be executed are in that directory, labelled as *query01.sql, query02.sql* and so on. The code blocks of interest have been labelled in the comments, and we will now go over them one by one :

1. The Spark environment is initialized. During the course of this experiment, we discovered that Spark refuses to compile queries that use a Cartesian product. We had to explicitly configure Spark to enable cross joins, by so :
   `sparkConf.set("spark.sql.crossJoin.enabled", "true")`

2. We have already discussed how a DataFrame relates to a table in a relational database. We also spoke about lowerBound, upperBound, and partitionStrides. For instance, let's take the case when we partition the table "catalog_sales". Since we set our our partitionColumn = "*cs_item_sk*", lowerBound = 10000, upperBound = 200000, and numOfPartitions = cores * 50 = 8 * 50 = 400, our partition strides become :

```
SELECT * FROM catalog_sales WHERE cs_item_sk IS NOT NULL AND (
cs_item_sk <  10000)
SELECT * FROM catalog_sales WHERE cs_item_sk IS NOT NULL AND (
cs_item_sk >=  10000 AND cs_item_sk < 209975)
.
.
SELECT * FROM catalog_sales WHERE cs_item_sk IS NOT NULL AND (
cs_item_sk > 200000)
```

3. For every table, we create a temporary view
4. We read the directory */queries* for our 99 queries. We extract the content of each file, and did the following modifications (go to *getQueriesMap* function):
   - The queries were stripped of the trailing semi-colon ';' . This was done because Spark throws an exception if the SQL string has a semi colon
   - For the same reason stated above, the queries were also stripped of all comments

- In some cases, a single file had two SQL queries separated by a semi-colon. For this reason, our *getQueriesMap* function returns a Map of Int keys and Sequence values, where each Sequence contains the multiple queries. If there is only one SQL query within the file, then our sequence has only one string

5. We then loop over the queries, and instruct Spark to run the queries. Our *sparkSqlContext.sql(sqlQuery)* function returns a DataFrame. Note, that in order to actually run the query, we need to perform an *"action"* on it. This, we accomplish by calling the *"show"* function, which basically will print the values returned upto the number passed as argument. As we are now looping over each query one by one, this gives us the opportunity to compute the time taken for Spark to run the query and write the result in **benchmark.csv**.

6. Finally, we exit Spark.

5. Benchmark Results

5.1 System Under Test (SUT)

According to the TPC Standard Specification[2] the SUT consists of:

a) The host system(s) or server(s), including hardware and software supporting access to the database employed in the performance test and whose cost and performance are described by the benchmark metrics

b) Any client processing units (e.g., front-end processors, workstations, etc.) used to execute the queries

c) The hardware and software components needed to communicate with user interfacedevices

d) The hardware and software components of all networks required to connect and support the SUT components

For this project, we used a SUT with the following characteristics:

- MacBook Pro (13-inch, Late 2011) [upgraded]
- Memory: 16 GB 1333 MHz DDR3
- Storage: Samsung SSD 850 EVO 250GB
- Processor Name: Intel Core i5
- Processor Speed: 2.4 GHz
- Number of Processors: 1
- Total Number of Cores: 2
- L2 Cache (per Core): 256 KB
- L3 Cache: 3 MB

5.2 Results comparison

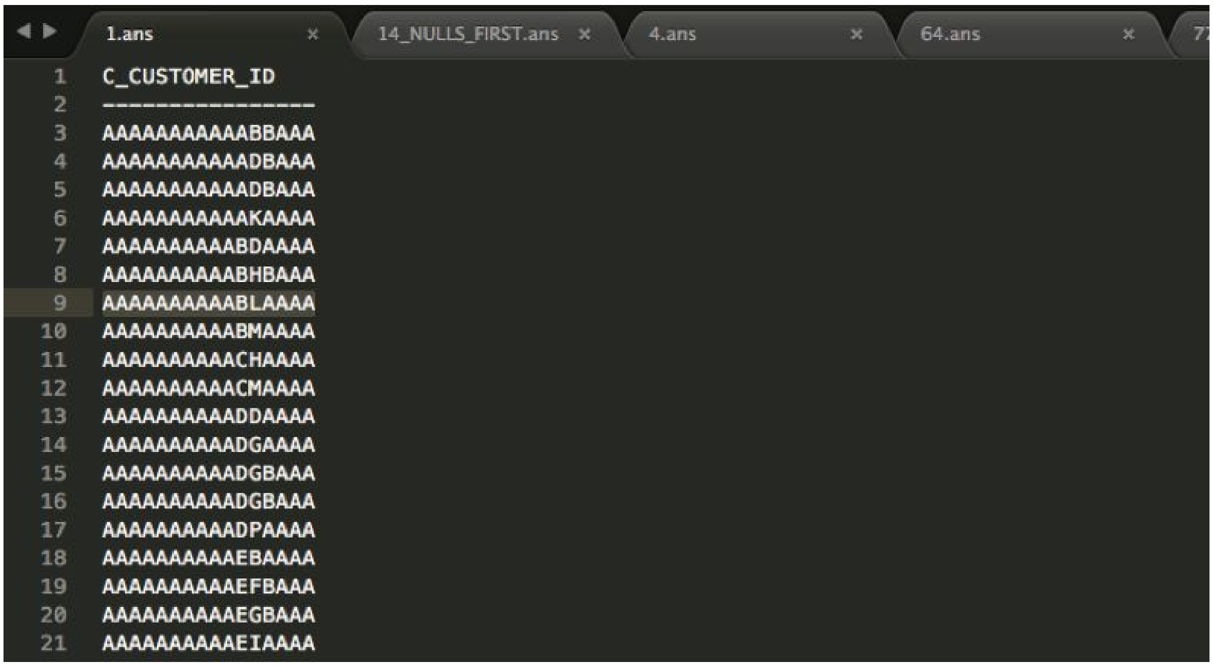
Up to this point, everything that is required to execute the experiment has been reviewed. We now compare the results of the queries. For doing so, we compared the results of all queries in our experiment with the answer set provided by the TPC-DS tools. We present the result comparison for some queries in the following sections, highlighting one random record to facilitate the reading.

5.1.1 Query 1.

Find customers who have returned items more than 20% more often than the average customer returns for a store in a given state for a given year.

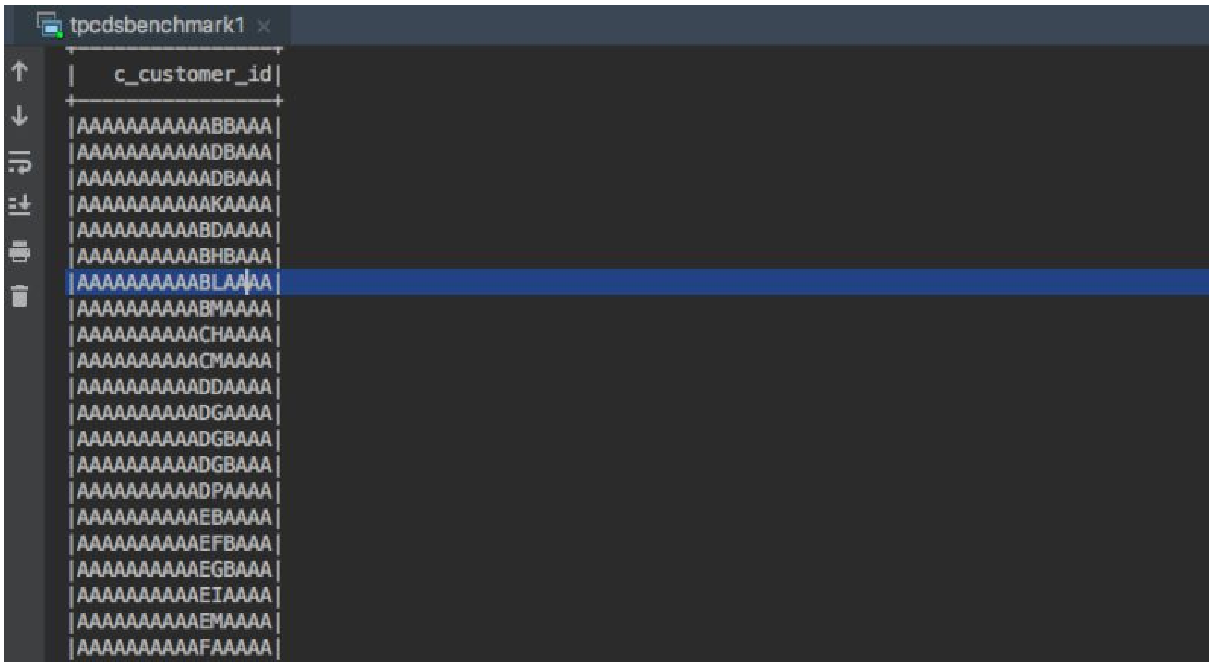
Qualification Substitution Parameters:

- YEAR.01=2000
- STATE.01=TN
- AGG_FIELD.01 = SR_RETURN_AMT



```
C_CUSTOMER_ID
----------------
AAAAAAAAAAABBAAA
AAAAAAAAAAADBAAA
AAAAAAAAAAADBAAA
AAAAAAAAAAAKAAAA
AAAAAAAAAABDAAAA
AAAAAAAAAABHBAAA
AAAAAAAAAABLAAAA
AAAAAAAAAABMAAAA
AAAAAAAAAACHAAAA
AAAAAAAAAACMAAAA
AAAAAAAAAADDAAAA
AAAAAAAAAADGAAAA
AAAAAAAAAADGBAAA
AAAAAAAAAADGBAAA
AAAAAAAAAADPAAAA
AAAAAAAAAAEBAAAA
AAAAAAAAAAEFBAAA
AAAAAAAAAAEGBAAA
AAAAAAAAAAEIAAAA
```

Figure 6: Query 1 answer with TPC-DS



```
+----------------+
|   c_customer_id|
+----------------+
|AAAAAAAAAAABBAAA|
|AAAAAAAAAAADBAAA|
|AAAAAAAAAAADBAAA|
|AAAAAAAAAAAKAAAA|
|AAAAAAAAAABDAAAA|
|AAAAAAAAAABHBAAA|
|AAAAAAAAAABLAAAA|
|AAAAAAAAAABMAAAA|
|AAAAAAAAAACHAAAA|
|AAAAAAAAAACMAAAA|
|AAAAAAAAAADDAAAA|
|AAAAAAAAAADGAAAA|
|AAAAAAAAAADGBAAA|
|AAAAAAAAAADGBAAA|
|AAAAAAAAAADPAAAA|
|AAAAAAAAAAEBAAAA|
|AAAAAAAAAAEFBAAA|
|AAAAAAAAAAEGBAAA|
|AAAAAAAAAAEIAAAA|
|AAAAAAAAAAEMAAAA|
|AAAAAAAAAAFAAAAA|
```

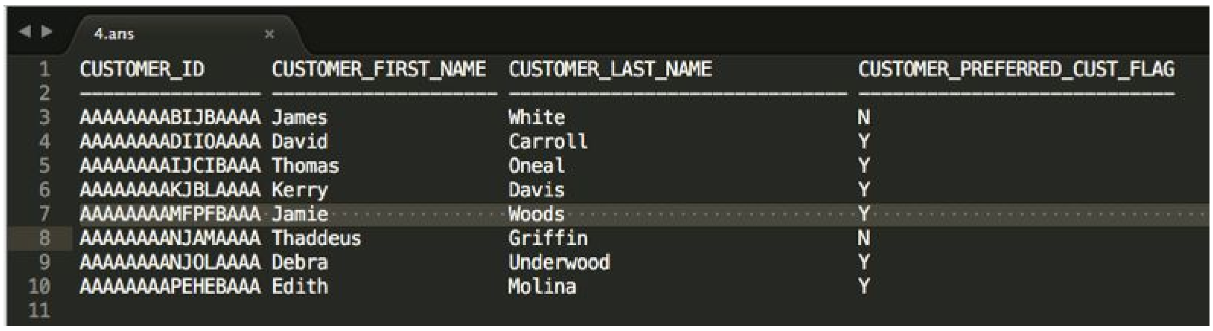
Figure 7: Query 1 answer - Spark over MySQL

5.1.2 Query 4.

Find customers who spend more money via catalog than in stores. Identify preferred customers and their country of origin. Qualification Substitution Parameters:

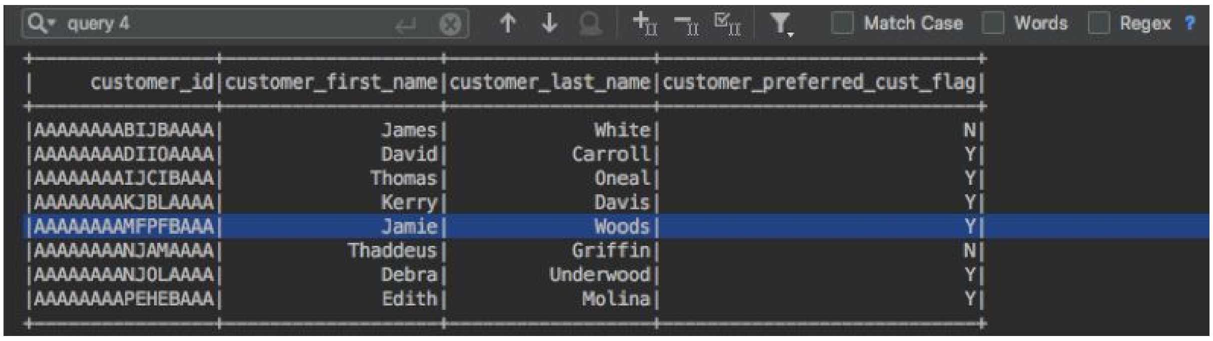
- YEAR.01=2001

· SELECTONE.01= t_s_secyear.customer_preferred_cust_flag



```
CUSTOMER_ID       CUSTOMER_FIRST_NAME  CUSTOMER_LAST_NAME                            CUSTOMER_PREFERRED_CUST_FLAG
----------------- -------------------- --------------------------------------------- ----------------------------
AAAAAAAABIJBAAAA James                White                                          N
AAAAAAAADIIOAAAA David                Carroll                                        Y
AAAAAAAAIJCIBAAA Thomas               Oneal                                          Y
AAAAAAAAKJBLAAAA Kerry                Davis                                          Y
AAAAAAAAMFPFBAAA Jamie                Woods                                          Y
AAAAAAAANJAMAAAA Thaddeus             Griffin                                        N
AAAAAAAANJOLAAAA Debra                Underwood                                      Y
AAAAAAAAPEHEBAAA Edith                Molina                                         Y
```

Figure 8: Query 4 answer of TPC-DS



```
+----------------+-------------------+------------------+----------------------------+
|     customer_id|customer_first_name|customer_last_name|customer_preferred_cust_flag|
+----------------+-------------------+------------------+----------------------------+
|AAAAAAAABIJBAAAA|              James|             White|                           N|
|AAAAAAAADIIOAAAA|              David|           Carroll|                           Y|
|AAAAAAAAIJCIBAAA|             Thomas|             Oneal|                           Y|
|AAAAAAAAKJBLAAAA|              Kerry|             Davis|                           Y|
|AAAAAAAAMFPFBAAA|              Jamie|             Woods|                           Y|
|AAAAAAAANJAMAAAA|           Thaddeus|           Griffin|                           N|
|AAAAAAAANJOLAAAA|              Debra|         Underwood|                           Y|
|AAAAAAAAPEHEBAAA|              Edith|            Molina|                           Y|
+----------------+-------------------+------------------+----------------------------+
```

Figure 9: Query 4 answer with Spark over Mysql

5.1.3 Query 14.

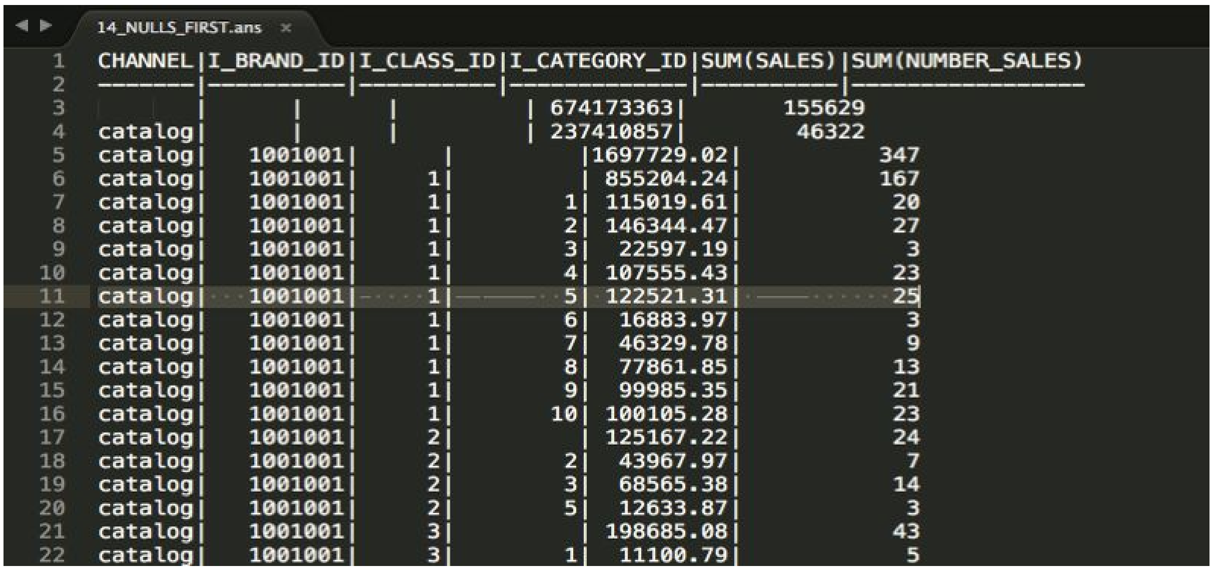
This query contains multiple iterations:

Iteration 1: First identify items in the same brand, class and category that are sold in all three sales channels in two consecutive years. Then compute the average sales (quantity*list price) across all sales of all three sales channels in the same three years (average sales). Finally, compute the total sales and the total number of sales rolled up for each channel, brand, class and category. Only consider sales of cross channel sales that had sales larger than the average sale.

Iteration 2: Based on the previous query compare December store sales. Qualification Substitution Parameters:

· DAY.01 = 11

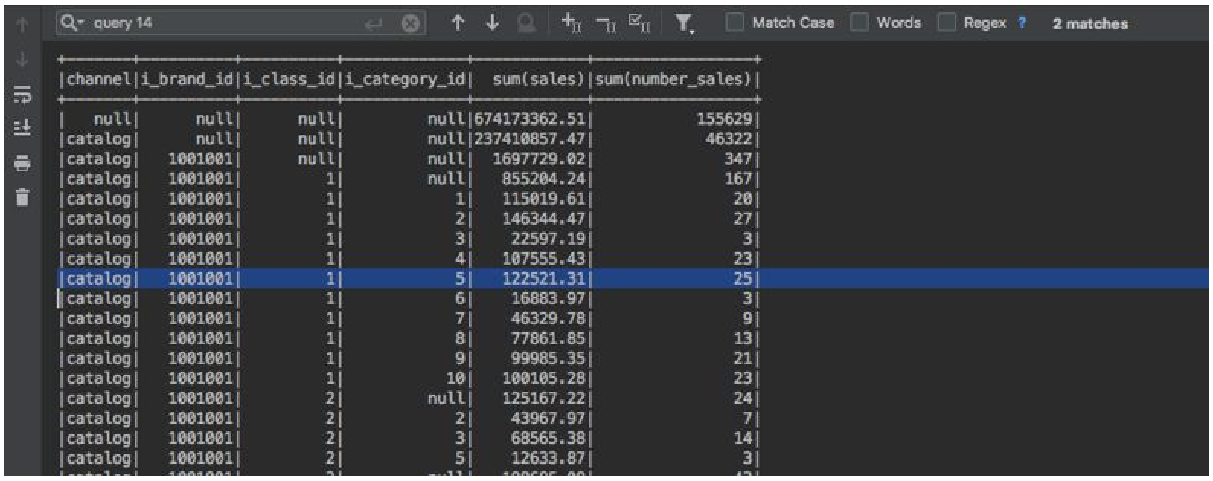
· YEAR.01 = 1999



14_NULLS_FIRST.ans

```
CHANNEL|I_BRAND_ID|I_CLASS_ID|I_CATEGORY_ID|SUM(SALES)|SUM(NUMBER_SALES)
-------|----------|----------|-------------|----------|-----------------
       |          |          |             | 674173363|        155629
catalog|          |          |             | 237410857|         46322
catalog|   1001001|          |             |1697729.02|               347
catalog|   1001001|         1|             | 855204.24|               167
catalog|   1001001|         1|            1| 115019.61|                20
catalog|   1001001|         1|            2| 146344.47|                27
catalog|   1001001|         1|            3|  22597.19|                 3
catalog|   1001001|         1|            4| 107555.43|                23
catalog|   1001001|         1|            5| 122521.31|                25
catalog|   1001001|         1|            6|  16883.97|                 3
catalog|   1001001|         1|            7|  46329.78|                 9
catalog|   1001001|         1|            8|  77861.85|                13
catalog|   1001001|         1|            9|  99985.35|                21
catalog|   1001001|         1|           10| 100105.28|                23
catalog|   1001001|         2|             | 125167.22|                24
catalog|   1001001|         2|            2|  43967.97|                 7
catalog|   1001001|         2|            3|  68565.38|                14
catalog|   1001001|         2|            5|  12633.87|                 3
catalog|   1001001|         3|             | 198685.08|                43
catalog|   1001001|         3|            1|  11100.79|                 5
```

Figure 10: Query 14 answer from TPC-DS



```
+-------+----------+----------+-------------+------------+-----------------+
|channel|i_brand_id|i_class_id|i_category_id|  sum(sales)|sum(number_sales)|
+-------+----------+----------+-------------+------------+-----------------+
|   null|      null|      null|         null|674173362.51|           155629|
|catalog|      null|      null|         null|237410857.47|            46322|
|catalog|   1001001|      null|         null|  1697729.02|              347|
|catalog|   1001001|         1|         null|   855204.24|              167|
|catalog|   1001001|         1|            1|   115019.61|               20|
|catalog|   1001001|         1|            2|   146344.47|               27|
|catalog|   1001001|         1|            3|    22597.19|                3|
|catalog|   1001001|         1|            4|   107555.43|               23|
|catalog|   1001001|         1|            5|   122521.31|               25|
|catalog|   1001001|         1|            6|    16883.97|                3|
|catalog|   1001001|         1|            7|    46329.78|                9|
|catalog|   1001001|         1|            8|    77861.85|               13|
|catalog|   1001001|         1|            9|    99985.35|               21|
|catalog|   1001001|         1|           10|   100105.28|               23|
|catalog|   1001001|         2|         null|   125167.22|               24|
|catalog|   1001001|         2|            2|    43967.97|                7|
|catalog|   1001001|         2|            3|    68565.38|               14|
|catalog|   1001001|         2|            5|    12633.87|                3|
```

Figure 11: Query 14 answer with Spark over Mysql

5.1.1 Query 64.

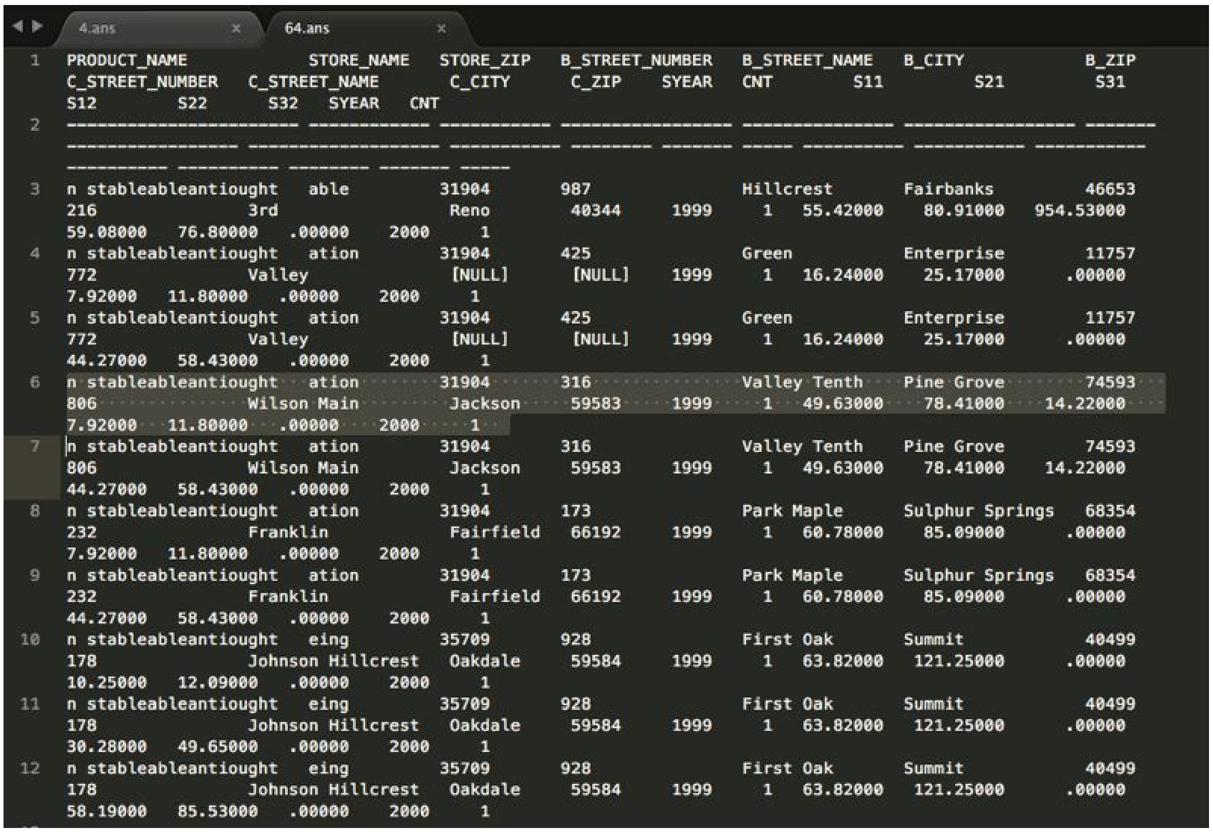
Find those stores that sold more cross-sales items from one year to another. Cross-sale items are items that are sold over the Internet, by catalog and in store.

Qualification Substitution Parameters:

- YEAR.01 = 1999
- PRICE.01 = 64
- COLOR.01 = purple
- COLOR.02 = burlywood
- COLOR.03 = indian
- COLOR.04 = spring

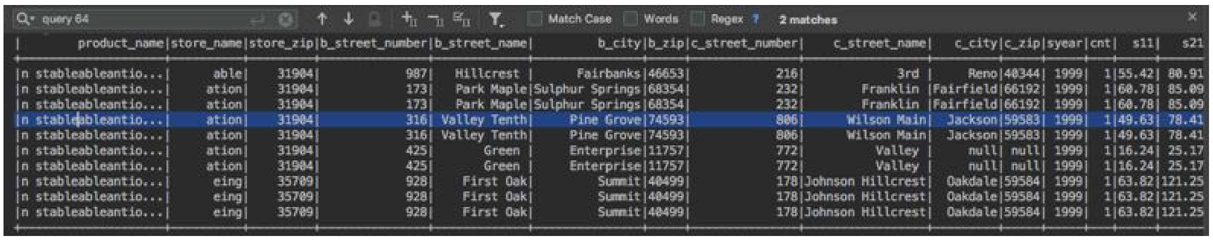
· COLOR.05 = floral

· COLOR.06 = medium



```
PRODUCT_NAME              STORE_NAME   STORE_ZIP   B_STREET_NUMBER  B_STREET_NAME   B_CITY            B_ZIP
C_STREET_NUMBER   C_STREET_NAME        C_CITY      C_ZIP     SYEAR   CNT        S11         S21         S31
S12        S22        S32    SYEAR    CNT
---------------------- ------------ ----------- ---------------- --------------- ----------------- -------
---------------- ------------------- ----------- --------- -------- ----- ---------- ----------- -----------
---------- ---------- --------- -------- -----
n stableableantiought   able         31904       987              Hillcrest       Fairbanks         46653
216               3rd                  Reno        40344     1999    1   55.42000    80.91000   954.53000
59.08000   76.80000    .00000    2000    1
n stableableantiought   ation        31904       425              Green           Enterprise        11757
772               Valley               [NULL]      [NULL]    1999    1   16.24000    25.17000      .00000
7.92000   11.80000    .00000    2000    1
n stableableantiought   ation        31904       425              Green           Enterprise        11757
772               Valley               [NULL]      [NULL]    1999    1   16.24000    25.17000      .00000
44.27000   58.43000    .00000    2000    1
n stableableantiought   ation        31904       316              Valley Tenth    Pine Grove        74593
806               Wilson Main          Jackson     59583     1999    1   49.63000    78.41000    14.22000
7.92000   11.80000    .00000    2000    1
n stableableantiought   ation        31904       316              Valley Tenth    Pine Grove        74593
806               Wilson Main          Jackson     59583     1999    1   49.63000    78.41000    14.22000
44.27000   58.43000    .00000    2000    1
n stableableantiought   ation        31904       173              Park Maple      Sulphur Springs   68354
232               Franklin             Fairfield   66192     1999    1   60.78000    85.09000      .00000
7.92000   11.80000    .00000    2000    1
n stableableantiought   ation        31904       173              Park Maple      Sulphur Springs   68354
232               Franklin             Fairfield   66192     1999    1   60.78000    85.09000      .00000
44.27000   58.43000    .00000    2000    1
n stableableantiought   eing         35709       928              First Oak       Summit            40499
178               Johnson Hillcrest    Oakdale     59584     1999    1   63.82000   121.25000      .00000
10.25000   12.09000    .00000    2000    1
n stableableantiought   eing         35709       928              First Oak       Summit            40499
178               Johnson Hillcrest    Oakdale     59584     1999    1   63.82000   121.25000      .00000
30.28000   49.65000    .00000    2000    1
n stableableantiought   eing         35709       928              First Oak       Summit            40499
178               Johnson Hillcrest    Oakdale     59584     1999    1   63.82000   121.25000      .00000
58.19000   85.53000    .00000    2000    1
```

Figure 12: Query 64 answer of TPC-DS



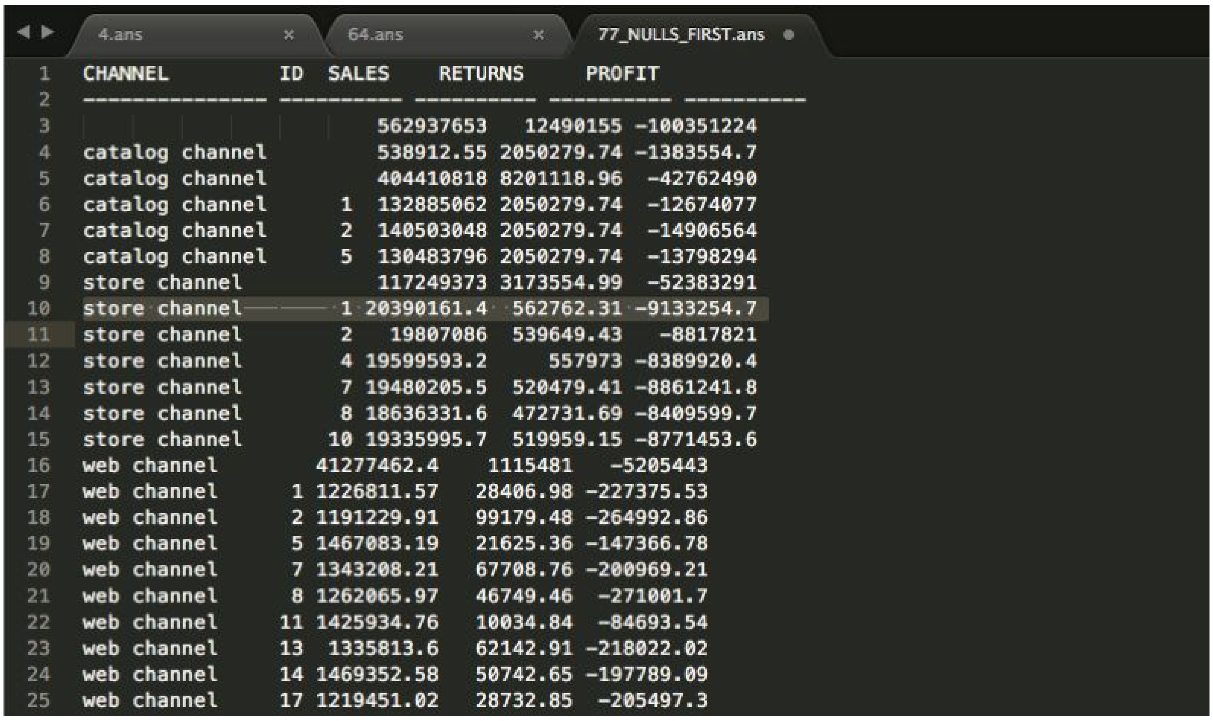
```
|        product_name|store_name|store_zip|b_street_number|b_street_name|         b_city|b_zip|c_street_number|     c_street_name|   c_city|c_zip|syear|cnt|  s11|   s21
+--------------------+----------+---------+---------------+-------------+---------------+-----+---------------+------------------+---------+-----+-----+---+-----+------
|n stableableantio...|      able|    31904|            987|   Hillcrest |      Fairbanks|46653|            216|               3rd |     Reno|40344| 1999|  1|55.42| 80.91
|n stableableantio...|     ation|    31904|            173|   Park Maple|Sulphur Springs|68354|            232|          Franklin |Fairfield|66192| 1999|  1|60.78| 85.09
|n stableableantio...|     ation|    31904|            173|   Park Maple|Sulphur Springs|68354|            232|          Franklin |Fairfield|66192| 1999|  1|60.78| 85.09
|n stableableantio...|     ation|    31904|            316| Valley Tenth|     Pine Grove|74593|            806|       Wilson Main|  Jackson|59583| 1999|  1|49.63| 78.41
|n stableableantio...|     ation|    31904|            316| Valley Tenth|     Pine Grove|74593|            806|       Wilson Main|  Jackson|59583| 1999|  1|49.63| 78.41
|n stableableantio...|     ation|    31904|            425|       Green |     Enterprise|11757|            772|            Valley |     null| null| 1999|  1|16.24| 25.17
|n stableableantio...|     ation|    31904|            425|       Green |     Enterprise|11757|            772|            Valley |     null| null| 1999|  1|16.24| 25.17
|n stableableantio...|      eing|    35709|            928|    First Oak|         Summit|40499|            178| Johnson Hillcrest|  Oakdale|59584| 1999|  1|63.82|121.25
|n stableableantio...|      eing|    35709|            928|    First Oak|         Summit|40499|            178| Johnson Hillcrest|  Oakdale|59584| 1999|  1|63.82|121.25
|n stableableantio...|      eing|    35709|            928|    First Oak|         Summit|40499|            178| Johnson Hillcrest|  Oakdale|59584| 1999|  1|63.82|121.25
+--------------------+----------+---------+---------------+-------------+---------------+-----+---------------+------------------+---------+-----+-----+---+-----+------
```

Figure 13: Query 64 answer using Spark over Mysql

5.1.2 Query 77.

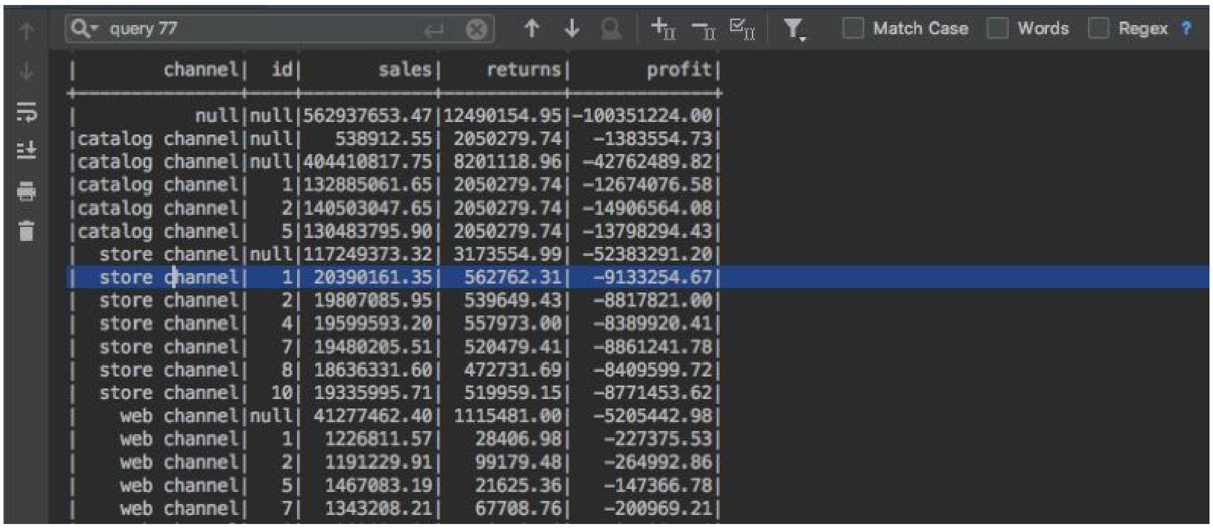
Report the total sales, returns and profit for all three sales channels for a given 30 day period. results by channel and a unique channel location identifier. Qualification Substitution Parameters:

· SALES_DATE.01 = 2000-08-23



```
CHANNEL         ID  SALES      RETURNS     PROFIT
--------------- ---------- ---------- ---------- ----------
                        562937653   12490155 -100351224
catalog channel         538912.55 2050279.74 -1383554.7
catalog channel         404410818 8201118.96  -42762490
catalog channel       1 132885062 2050279.74  -12674077
catalog channel       2 140503048 2050279.74  -14906564
catalog channel       5 130483796 2050279.74  -13798294
store channel           117249373 3173554.99  -52383291
store channel         1 20390161.4  562762.31 -9133254.7
store channel         2  19807086  539649.43   -8817821
store channel         4 19599593.2     557973 -8389920.4
store channel         7 19480205.5  520479.41 -8861241.8
store channel         8 18636331.6  472731.69 -8409599.7
store channel        10 19335995.7  519959.15 -8771453.6
web channel        41277462.4    1115481    -5205443
web channel       1 1226811.57   28406.98 -227375.53
web channel       2 1191229.91   99179.48 -264992.86
web channel       5 1467083.19   21625.36 -147366.78
web channel       7 1343208.21   67708.76 -200969.21
web channel       8 1262065.97   46749.46   -271001.7
web channel      11 1425934.76   10034.84   -84693.54
web channel      13  1335813.6   62142.91 -218022.02
web channel      14 1469352.58   50742.65 -197789.09
web channel      17 1219451.02   28732.85   -205497.3
```

Figure 14: Query 77 answer using Spark over Mysql



```
|         channel|  id|        sales|    returns|         profit|
+----------------+----+-------------+-----------+---------------+
|            null|null|562937653.47|12490154.95|-100351224.00|
|catalog channel|null|   538912.55| 2050279.74|  -1383554.73|
|catalog channel|null|404410817.75| 8201118.96| -42762489.82|
|catalog channel|   1|132885061.65| 2050279.74| -12674076.58|
|catalog channel|   2|140503047.65| 2050279.74| -14906564.08|
|catalog channel|   5|130483795.90| 2050279.74| -13798294.43|
|  store channel|null|117249373.32| 3173554.99| -52383291.20|
|  store channel|   1| 20390161.35|  562762.31|  -9133254.67|
|  store channel|   2| 19807085.95|  539649.43|  -8817821.00|
|  store channel|   4| 19599593.20|  557973.00|  -8389920.41|
|  store channel|   7| 19480205.51|  520479.41|  -8861241.78|
|  store channel|   8| 18636331.60|  472731.69|  -8409599.72|
|  store channel|  10| 19335995.71|  519959.15|  -8771453.62|
|    web channel|null| 41277462.40| 1115481.00|  -5205442.98|
|    web channel|   1|  1226811.57|   28406.98|   -227375.53|
|    web channel|   2|  1191229.91|   99179.48|   -264992.86|
|    web channel|   5|  1467083.19|   21625.36|   -147366.78|
|    web channel|   7|  1343208.21|   67708.76|   -200969.21|
```

Figure 15: Query 77 answer using Spark over Mysql

5.3 Evaluated Scenarios (partitioning)

5.3.1 Scenario 1 - 4 Partitions.

Since we are using the 1GB scale factor, in the four first scenarios, partitioning was used just for the tables with a considerable number of the records.

Since we are running in one single node with four cores, we decided to use 4 partitions in the first scenario.

All the queries took 8268 seconds (**2h18min**) to execute. We consider this is a good time considering the SUT is just one node running not only the queries but some others applications.

5.3.2 Scenario 2 - 8 Partitions.

In this scenario, we partitioned the same tables, but we double the number of partitions to see how this could affect the performance of the system.

After running all the queries, the time was 8231 seconds or **2h17min**. The results were not conclusive at this point, we didn't know until which point increasing the number of partitions could start to affect the performance.

5.3.2 Scenario 3 - 12 Partitions.

We decided to increase to 12 and 16 partitions in the following escenarios.

For 12 partitions, the time required to schedule all the tasks in different cores and partitions starts to affect the overall performance of the system.

Under this scenario, the queries last 9703 seconds, it is **2h42m**.

5.3.2 Scenario 4 - 16 Partitions.

With the intention to corroborate the tendency, we finally increased the number of partitions to 16. It took 9993 seconds (**2h47m**). It's already a difference of 30 minutes with the first scenario.

In the following figures we show the execution time of the queries. Let's start with an overview.

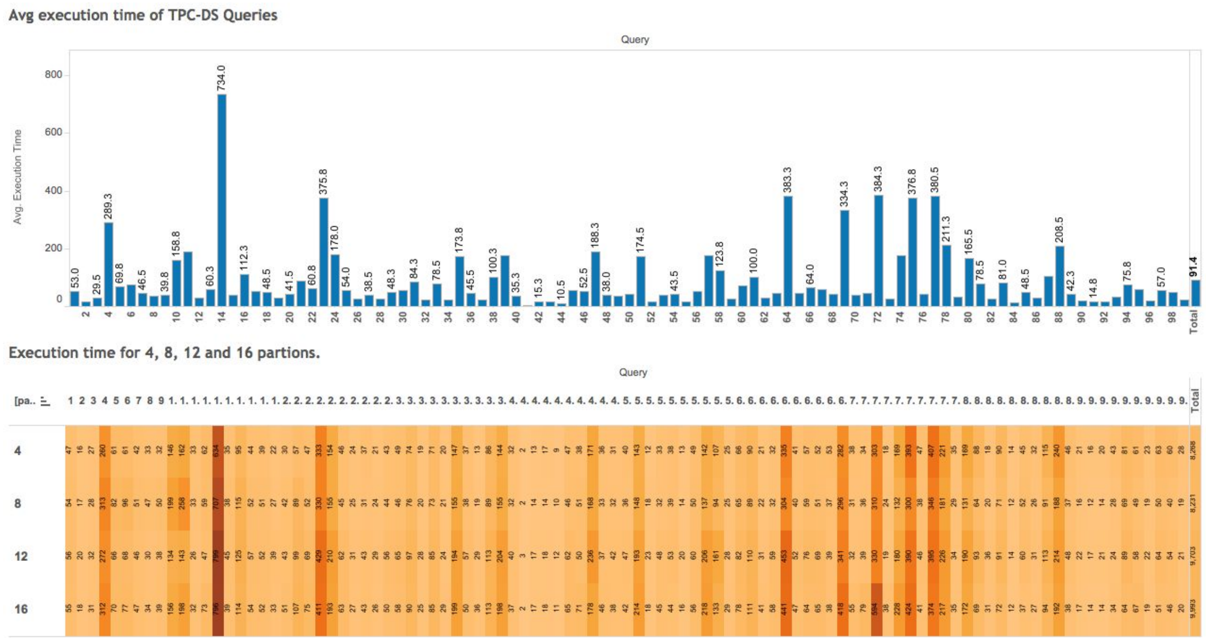
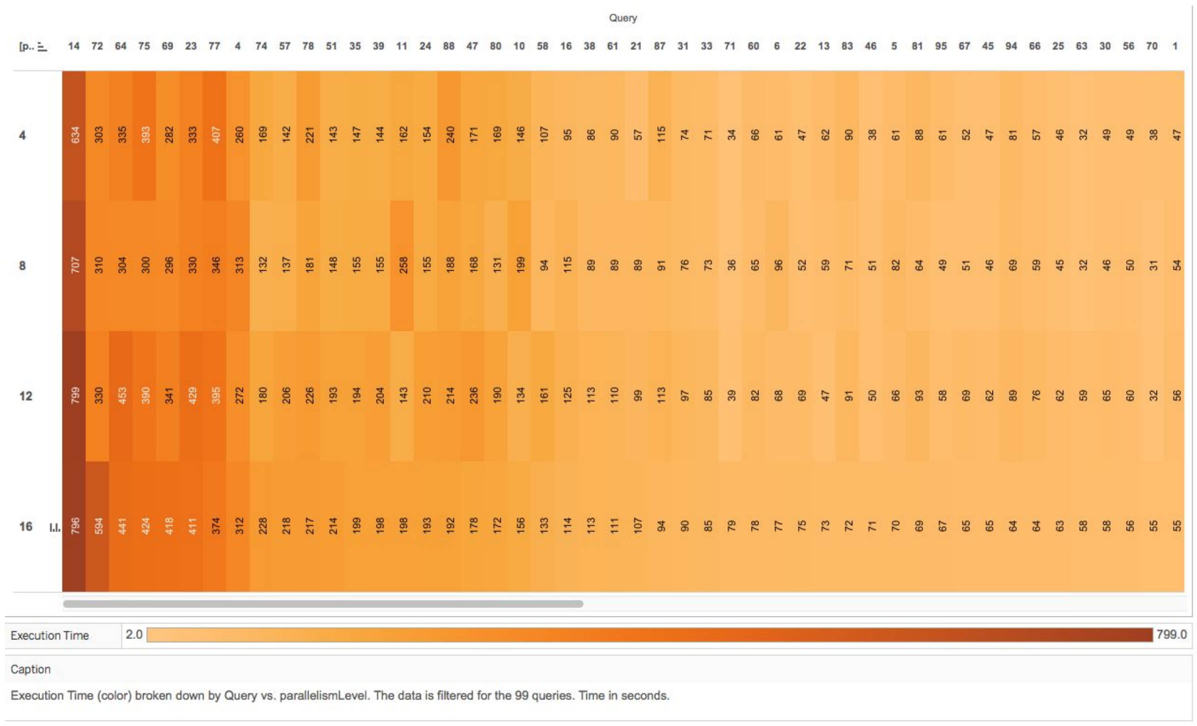


Figure 16: Execution time for each query. Scenarios 1 to 4

There are some queries that took a lot of time in comparison with the others. For instance query 14, which consists of two iterations as we described in the previous section.

In the next figure we present the queries which took longer to execute sorted by the 4th scenario. It starts with query **14** as we previously mentioned and followed by queries **72, 64, 75, 69, 23, 77, 4, 74** and **57** to complete the top 10.



[p..	14	72	64	75	69	23	77	4	74	57	78	51	35	39	11	24	88	47	80	10	58	16	38	61
4	634	303	335	393	282	333	407	260	169	142	221	143	147	144	162	154	240	171	169	146	107	95	86	90
8	707	310	304	300	296	330	346	313	132	137	181	148	155	155	258	155	188	168	131	199	94	115	89	89
12	799	330	453	390	341	429	395	272	180	206	226	193	194	204	143	210	214	236	190	134	161	125	113	110
16	796	594	441	424	418	411	374	312	228	218	217	214	199	198	198	193	192	178	172	156	133	114	113	111

[p..	21	87	31	33	71	60	6	22	13	83	46	5	81	95	67	45	94	66	25	63	30	56	70	1
4	57	115	74	71	34	66	61	47	62	90	38	61	88	61	52	47	81	57	46	32	49	49	38	47
8	89	91	76	73	36	65	96	52	59	71	51	82	64	49	51	46	69	59	45	32	46	50	31	54
12	99	113	97	85	39	82	68	69	47	91	50	66	93	58	69	62	89	76	62	59	65	60	32	56
16	107	94	90	85	79	78	77	75	73	72	71	70	69	67	65	65	64	64	63	58	58	56	55	55

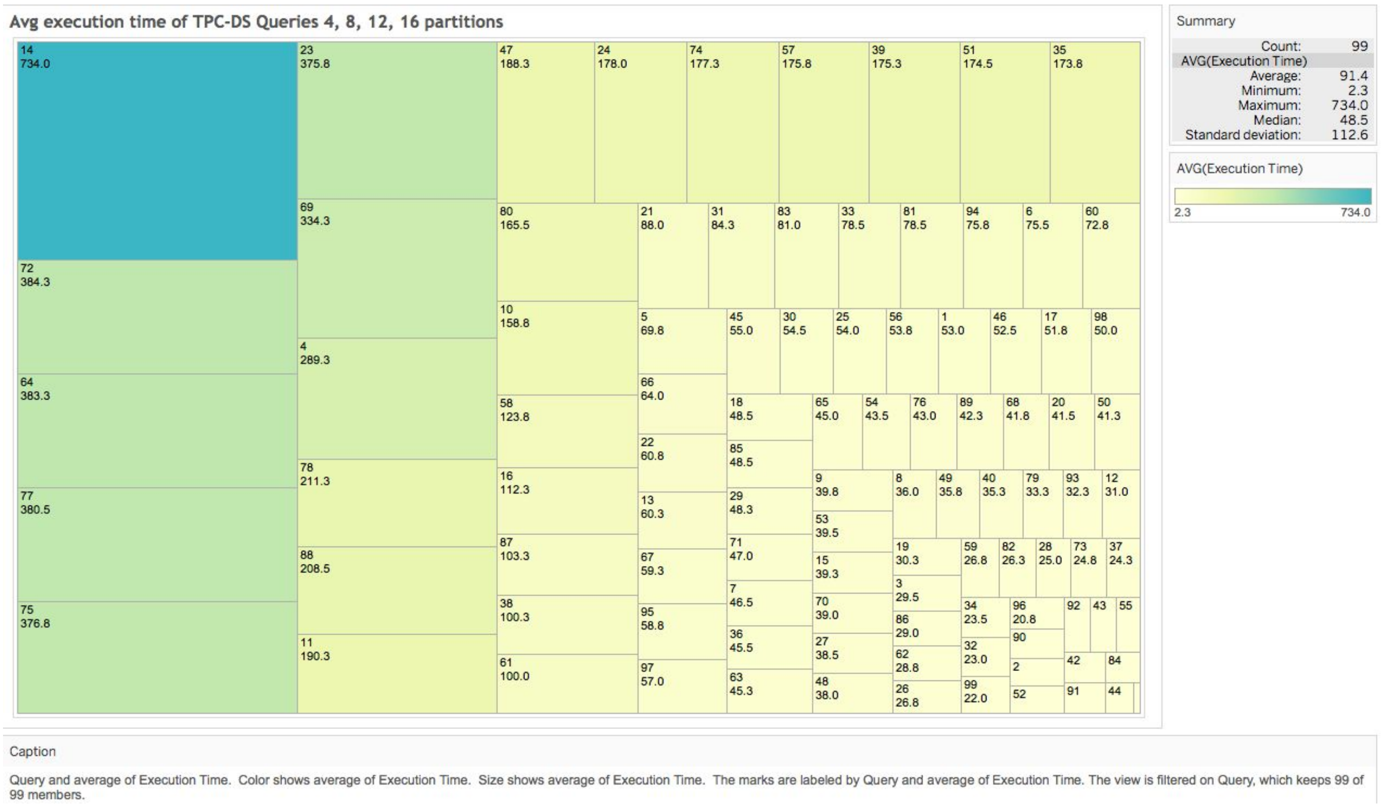
Execution Time	2.0	799.0

Caption

Execution Time (color) broken down by Query vs. parallelismLevel. The data is filtered for the 99 queries. Time in seconds.

Figure 17: Longest queries to execute sorted by the 4th scenario

Here are the details of the average execution time:



Summary

Count:	99
AVG(Execution Time)	
Average:	91.4
Minimum:	2.3
Maximum:	734.0
Median:	48.5
Standard deviation:	112.6

Caption

Query and average of Execution Time. Color shows average of Execution Time. Size shows average of Execution Time. The marks are labeled by Query and average of Execution Time. The view is filtered on Query, which keeps 99 of 99 members.

<u>Figure 18:</u> Avg execution time of TPC-DS queries for 4, 8, 12, 16 partitions

5.3.3 Scenario 5 - Partitioning for 5GB dataset

We also wanted to check if we could run the queries against the 5GB dataset. We tried first using the same partition scheme we had configured so far with the 1 GB dataset. We got a lot of errors in different queries regarding timeout in the response of the threads, Out of Memory errors and Garbage Collection errors.

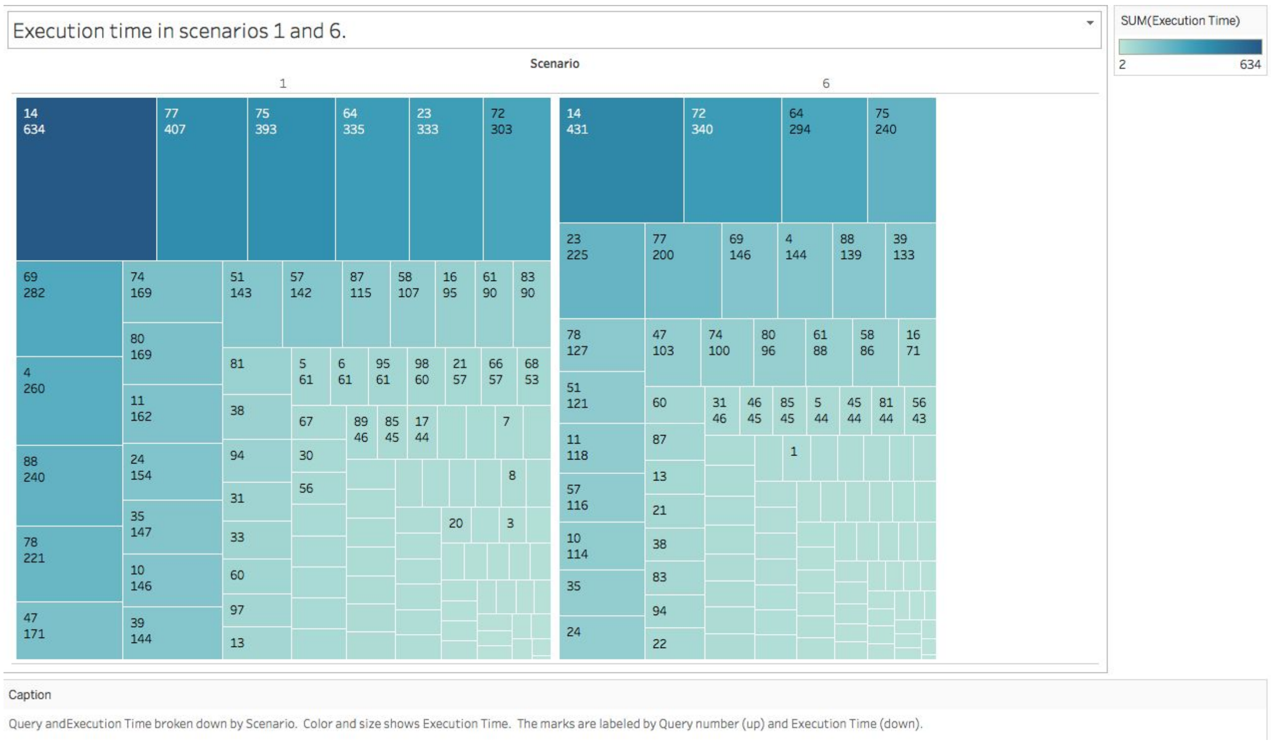
We noticed we would have to partition the tables in a different way. If you are not careful, the quantity of records in each table is too big for managing it in one single node. We had to include more tables in the partition schema and find the proper ranges according to the partition key we were using. We tested different configurations.

Finally, we found a configuration that executed 71 queries using 200 partitions, it was able to compute the queries but it was taking too much time and we stopped the process.

5.3.4 Scenario 6 - Testing the best partition scheme in 1 GB dataset.

The previous test allowed us to understand a little bit more about Spark SQL and the partition schema. We used the same technique for the partition scheme that we obtained while solving the errors we got with the 5GB data set to see if it could improve the performance for 1 GB database. For this scenario we used 4 partitions.

In the next figure we compare the results with the first scenario. Indeed, this last configuration has a better performance over the whole process. It took **1h38m** to execute all the queries, while the best of the other configurations took **2h17m**. That is an improvement of **39 minutes**.



Caption

Query andExecution Time broken down by Scenario. Color and size shows Execution Time. The marks are labeled by Query number (up) and Execution Time (down).

Figure 19: Execution time in scenarios 1 and 6

6. Conclusions

We looked at how Apache Spark and MySQL complement each other and bring forward a better overall solution for Data Warehousing using the TPC-DS benchmark, which allowed us to execute queries that answer real-world business questions with different operational requirements.

Specifically in Spark, we learned how the partitioning works. With different experiments we got to understand the nature of the data we were trying to process and according to that, we found a better solution in scenario number 6, improving considerably the execution time of all the queries.

We took the advantage of the powerful Scala API provided by Spark, and got a taste of what it would mean to implement production level systems using Scala, MySQL and Spark. Moreover, the solution we wrote could be could be extended to other RDBMS that supports JDBC.

As future work, we need to deploy the solution over a cluster, evaluating the different partition schemas in multiple nodes with bigger scale factors.

7. References

[1] Rui Han, et al. *Benchmarking Big Data Systems: State-of-the-Art and Future Directions*. Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 2016.

[2] Transaction Processing Performance Council. "TPC DS Standard Specification", v. 2.10.0. Documentation page. http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp

[3] Jacek Laskowski. "Mastering Apache Spark", https://legacy.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details.

[4] Armbrust, Michael, et al. "Spark SQL: Relational Data Processing in Spark." *ACM*, 4 May 2015, people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf.

[5] Foundation, Apache Software. "Spark Programming Guide." *Spark Programming Guide - Spark 1.6.0 Documentation*, https://spark.apache.org/docs/1.6.0/programming-guide.html#rdd-operations.

[6] "Architecture of Spark SQL?" https://www.packtpub.com/mapt/book/big_data_and_business_intelligence/9781785884696/4/ch04lvl1sec26/architecture-of-spark-sql

[7] "Whats the Meaning of PartitionColumn, LowerBound, UpperBound, NumPartitions Parameters?" *Stack Overflow*, 1 year 10 months, stackoverflow.com/questions/41085238/whats-meaning-of-partitioncolumn-lowerbound-upperbound-numpartitions-paramete#.