The battle for interactive Big Data queries in the post-MapReduce era

Leonidas Barmpas

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements of the Degree of Master of Science at the University of Glasgow

September 7th, 2015

Abstract

Ever since its advent a decade ago, MapReduce dominated the scene of batch Big Data processing. However, modern workloads and applications require online/interactive query processing capabilities, where MapReduce fails miserably to deliver. YARN made the first step in decoupling the data storage from MapReduce's processing chores, evolving into what is now called a "data operating system". The next step then came in the form of Spark and Tez, new data processing infrastructures pushing the envelope of what is achievable with current clusters of commodity computers. They both implement in-memory data management and processing engines on top of YARN, allowing for the creation of arbitrary directed acyclic graphs of processing components to be combined to produce highly efficient execution plans. This project aims to benchmark these frameworks and research their design decisions and implementation details

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

<Please note that you are under no obligation to sign this declaration, but doing so would help future students.>

Name: Leonidas Barmpas Signature:

Acknowledgements

I would like to thank Dr. Nikos Ntarmos for the support and advice during this thesis process.

Contents

Chapter 1 Introduction

MapReduce[1] and its open source implementation Hadoop[2] have been since their emergence the dominant technologies for batch processing Big Data. Modern workloads though require interactive query capabilities, with which these systems cannot cope anymore. Several technologies have arisen lately in order to provide these capabilities. The aim of this project is to research and evaluate the major related technologies contesting in this post-MapReduce era.

1.1 Problem Formulation

Hadoop's main strength is in batch processing data, issuing queries in relatively static large datasets, long running extract/transform/load (ETL) jobs, where query execution time is not of great importance. Hive[3] is a data warehousing tool, initially implemented on top of Hadoop, designed to provide an SQL-like interface for ETL jobs to users, bypassing the need to write MapReduce code. Apache Pig[4], fills the same gap, providing a procedural language, similar to what an SQL parser would produce, in order to perform queries on Hadoop. Both of these tools transform user input into MapReduce jobs, to be executed on the underlying Hadoop infrastructure.

Modern workloads however demand even faster ways to analyze data stored in Hadoop, in an interactive/real-time way. Tools running on top of MapReduce are very slow, as each query is translated into multiple MapReduce jobs, leading to overall suboptimal execution plans. This need has led to the creation of various tools and technologies tackling the problem, providing interactive query capabilities on top of Hadoop, with fast results and the ease of on-the-fly redefining queries using SQL-compatible languages. Apache Tez[5] is an application framework aimed at processing data using directed-acyclic-graphs of tasks, whereas Apache Spark[6] is a fast and general framework for Big Data processing, using in-memory computing and an advanced directed-acyclic-graph engine, promising even faster query execution. On top of that, different storage formats optimized for performance have been proposed and implemented to replace plain HDFS[7] files.

The aim of this project is to evaluate these technologies and compare them in order to acquire a deeper understanding of their performance improvements over Hadoop batch tools.

Particularly, we are going to conduct the following experiment. At the bottom layer, our physical infrastructure will be a virtual machine, loaded with a Hortonworks HDP Sandbox[8] image for the operating system. On top of that, our storage layer will be HDFS, storing data in plain text files, as well as ORC[9] and Parquet[10] table files. Resource management is taken care of by YARN[11] alongside with Spark for the various frameworks we will benchmark. MapReduce, Tez and Spark will be our execution engines where our high level tools, particularly Hive, Pig and Spark SQL[12], will be parsing the user input and translating it into tasks to be shipped to the execution engine. For the benchmark of the frameworks, the dataset and queryset from the TPC-H benchmark[13] will be used, on various scale factors.

1.2 Motivation

The timing of this project is of great importance. Big Data analytics are constantly gaining attention and even more vendors publish tools to handle this kind of data. SQL-on-Hadoop technologies are emerging and constantly growing, adding features to cope with interactive query needs for accessing, analyzing and manipulating of really large scales of data. Gaining a deeper understanding of the design decisions and implementation details of various frameworks for accessing and processing large datasets on a distributed infrastructure will help us acquire valuable experience. Additionally, having a benchmark of the latest versions of major current contestants is something that is lacking and could potentially help make the differences between them clearer. SAS Inc[14], a company active in the field of Big Data processing, also took an interest in the project and came in touch with us -- an indication of the importance, timeliness and possible impact of this project.

1.3 Outline

In the following chapters, we are first going to discuss related works, comparing different Big Data query processing systems, and define our project’s contributions on top of that knowledge. Then, we will provide some background information on the various aspects of the technologies that will be compared, so as to get a clearer understanding of all the implementation details. After that, we will describe the various parts of our experimental setup, such as the hardware infrastructure used for the experiments, the dataset and the queryset, as well as the different metrics to perform our evaluations. We will then analyze the performance results of these experiments along different dimensions. Finally, we will try to extract some knowledge from these results and come up with overarching conclusions about the behavior of the various technologies. Also we will refer to the difficulties stumbled upon during the project and their solutions, as well as the knowledge gained throughout this project.

Chapter 2 Related Work

Several projects have taken place over time benchmarking different tools that sit on top of Hadoop, enabling it to run interactive queries for data analysis. A study conducted at Berkeley[15] measured the response time of a group of relational queries on five different systems: Redshift[16], Impala[17], Shark[18], Hive, and Hive on Tez. The study concludes to some interesting results about the technologies tested and the differences between them. In general, frameworks performing in-memory data processing such as Shark and Impala are significantly faster than others, and Tez as expected offers a major performance boost to Hive replacing MapReduce. The main drawback of the project is that it is using quite old versions of said software tools. In addition to that, data are stored in standard Hadoop sequence files, missing any kind of optimization and improvement in performance provided by columnar formats like ORC and Parquet.

IBM also conducted a benchmark[19] comparing the three leading frameworks at the time, in order to test whether SQL on Hadoop is ready to run complex workloads and support OLAP functionality. The TPC-DS[20] benchmark was used to compare IBM's Big SQL[21] with Hive and Impala over a 10TB workload. The main issue of the IBM's benchmark is that it strives to comply with the TPC rules, favoring Big SQL that runs queries with slight alterations but dropping queries on Hive and Impala because the changes required for them to run are not TPC compliant. The study is more geared towards seamless integration into customers' environments than actually comparing the best of each framework. Additionally, the data format used is not stated but is probably not a columnar one. Spark and Tez, which would be quite competitive participants, are also missing from the comparison.

Another benchmark from Radiant Advisors[22] is focused on measuring the speed and SQL capabilities on Hadoop of several tools such as Hive, Presto[23], InfiniDB[24] and Impala on various file formats such as ORC, Parquet, IDB[25] and plain sequence files. Multiple versions of each framework were tested, to form a straightforward comparison of SQL engines' performance on Hadoop. The results clearly show the inability of Hive on MapReduce to cope with the fast query response times of other tools designed with the aim of providing analytic SQL capabilities on top of Hadoop. It is also proved from their experiments that columnar file formats can provide a significant boost in response times when used from tools that fully support them. The benchmark also comes to the conclusion that no one SQL engine can meet all the needs but each has its advantages over others. Frameworks have since evolved and added many features that could improve performance since this benchmark uses old versions. In addition to that, Spark and Tez are not tested.

Apart from benchmarks, there have also been guides to improve performance on specific tools. A Hortonworks guide[26] suggests techniques on making Hive queries run faster, by enabling recently added features. [27] provides an in depth guide of Spark's execution model to help users write efficient Spark programs. [28] gives guidelines for using Pig joins in an optimal way according to the dataset. Last, performance can be increased by also tuning the file formats of the data as shown in a guide by Atlassian[29].

Big Data and especially frameworks enabling interactive query execution are constantly evolving, updating and adding features. Additionally, more frameworks and optimizations are constantly introduced, making comparisons between different tools quite hard. Last, said comparisons can become obsolete in a very short time and should be reconducted in order to be able to present truthful and meaningful results.

Chapter 3 Background

In this chapter, we will present the various dimensions of the infrastructure we used for the experiments. We will discuss the lower level of the benchmark, the data storage layer, then processing infrastructures and execution engines sitting on top of that, and finally the high level tools, the top layer enabling the user to easily interact with the frameworks.

3.1 Storage Layer

The file system used to store all of our data is the Hadoop Distributed File System (HDFS), a distributed file system. HDFS is the main file system used with Hadoop, and is designed to be a highly fault-tolerant file system, with the ability to be run on low cost hardware and highly optimized for high throughput on large data sets.

Initially, the generated data are stored in HDFS as plain files. Each table for every scale factor of the dataset used, is stored in a separate file, in csv-style format, with each record delimited by the character “|”. These files are used as External tables in order to load them in the formats used for the experiments. External tables is a way of easily using in Hive data already generated from another source, by pointing the location they are stored in and describing their schema.

One of the formats used in the performance evaluations is Optimized Row Columnar (ORC)[9] file format. ORC, added on Hive 0.11, acts as a replacement to the RCFile[30] format used for years in Hadoop. ORC is a self describing columnar file format optimized for streaming reads on large scale data. Data are stored in fixed size stripes in a columnar format as shown in Figure 1, retaining type information from the table definition, thus allowing type specific compression resulting in significantly smaller files. In addition to that, each stripe holds lightweight indexes for the data stored in it, enabling systems to skip whole stripes that do not satisfy the criteria of a particular query. ORC also supports projection, so queries that need specific subset of columns read only the bytes required.

The second format that will be tested in the experiments is Apache Parquet[10], the main competitor of the ORC format. Parquet was designed as an open source columnar format for Hadoop, that would work well independently of the framework or the serialization library used with. Data are stored in row groups, followed by metadata that contain information about the columns stored as shown in Figure 2, making it self described too. Similar to the ORC format, reading data requires first reading the metadata to find all the columns needed for a specific query.

Both ORC and Parquet file formats have the advantage that they store data in a columnar way. Instead of storing data row by row, they are stored in groups of the same column. This leads to performance improvements as non relevant columns can be skipped all together since they are stored consecutively. In addition to that, having the same type of data stored together can benefit from using specific type compression, drastically decreasing the final file size.

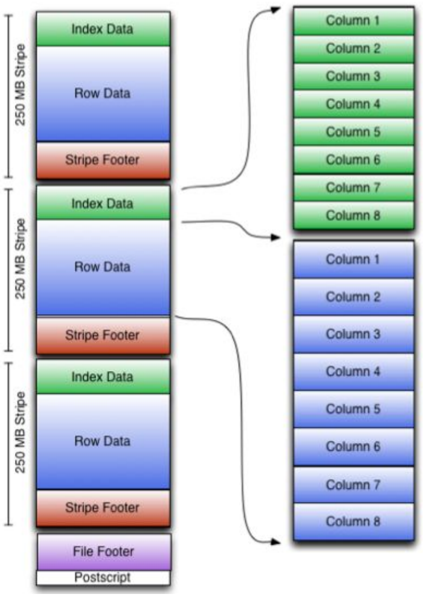

Figure 1 ORC file layout[1]

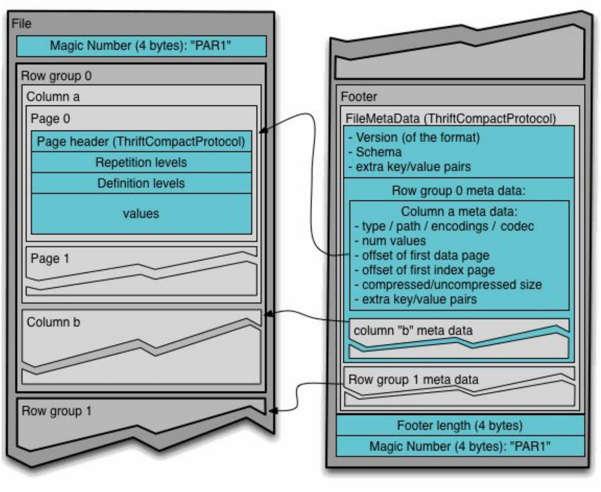
File
Magic Number (4 bytes): "PAR1"
Row group 0
Column a
Page 0
Page header (ThriftCompactProtocol)
Repetition levels
Definition levels
values
Page 1
Column b
Row group 1
Footer
FileMetaData (ThriftCompactProtocol)
- Version (of the format)
- Schema
- extra key/value pairs
Row group 0 meta data:
Column a meta data:
- type / path / encodings / codec
- num values
- offset of first data page
- offset of first index page
- compressed/uncompressed size
- extra key/value pairs
column "b" meta data
Row group 1 meta data
Footer length (4 bytes)
Magic Number (4 bytes): "PAR1"

Figure 2 Parquet file layout[2]

3.2 Distributed processing infrastructures

MapReduce[1] is a framework for parallel processing of large datasets making use of multiple nodes, working together in a distributed manner. Its major components are a JobTracker, which is essentially the master node that manages all the jobs and allocates the

[1] Image taken from https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC

[2] Image taken from https://parquet.apache.org/documentation/latest/

resources of the cluster and the TaskTrackers, which are agents deployed in each node whose job is to process the tasks that are assigned to them by the JobTracker, running either map or reduce tasks. By distributing independent input data splits and combining results in TaskTrackers, jobs on large datasets can be run effectively in parallel. MapReduce can take advantage of data locality to minimize transfers over the network. One of the major limitations of the MapReduce model is that in order for the reducers to start processing data, all mappers have to be finished processing their respective input splits, as reducers may need to get data from every mapper's output. Additionally, running chained MapReduce jobs imposes the need of the each job to finish before the next one starts if it uses the output of the former. This can bottleneck parallelization though and increase execution times for jobs.

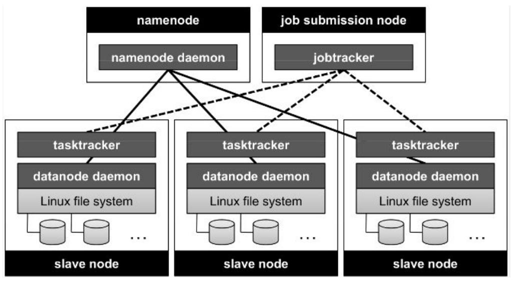

Figure 3 MapReduce Infrastructure[3]

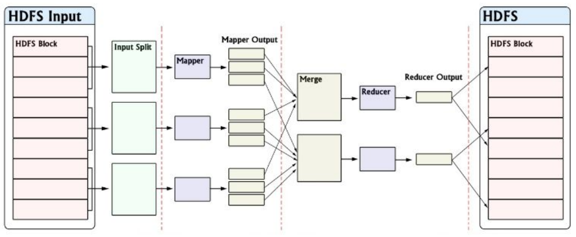

Figure 4 MapReduce data flow[4]

YARN[11] (also known as MapReduce v2) is a complete remake to update the MapReduce framework and deal with its limitations. The main components are now a Resource Manager, which is responsible for various things such as resource allocation through the Resource Scheduler, accepting jobs from clients and assigning them to nodes through the Application Manager, and the Node Managers. Node Manager are agents running in each node responsible for monitoring their resources, called containers, and reporting them to the Resource Manager. When jobs are accepted by the Resource Manager, it assigns a container as the Application Master for that specific job, which then negotiates with the Resource Scheduler for getting the appropriate containers to run the job, so each application runs with its own Application Master and containers. Apart from dealing with scalability, single points of

[3] Image taken from Big Data 4 class material

[4] Image taken from Big Data 4 class material

failure and resource utilization problems of MapReduce, YARN has also opened the way for running applications that do not exactly follow the MapReduce model, such as interactive queries and streaming applications, by decoupling the resource management and scheduling from the data processing components.

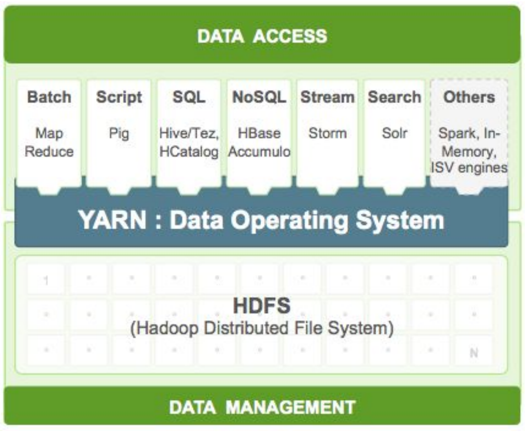

Figure 5 YARN stack[5]

Tez[5] is an application framework built on top of YARN that enables processing data in a directed acyclic graph of tasks where the vertices represent the application logic and the edges movement of data. Tez overcomes the limitations of MapReduce, allowing the processing of data in single Tez jobs that in normal MapReduce would require multiple jobs. In addition to that, Tez's architecture does not impose synchronization, as opposed to MapReduce, where all mappers must finish for the reducers to start and a job must wait for the previous to finish in order to start. The Tez project has set the standard for interactive workload execution on YARN environments. It is usually not used directly from end users, but higher level interfaces use it as an execution engine, taking advantage of its optimizations, which are mainly based on dynamically reconfiguring the graphs for proper resource utilization and performance by collecting runtime information.

[5] Image taken from Hortonworks http://hortonworks.com/

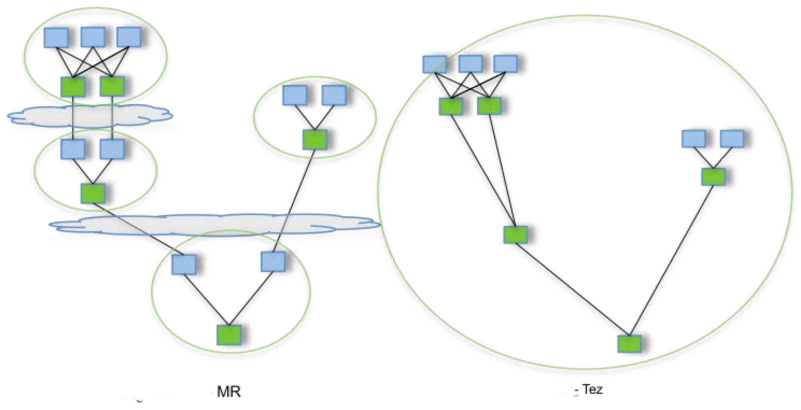

Figure 6 MapReduce (left) vs Tez execution (right)[6]

Spark[6] is a big data processing framework built with the aim to provide speed and ease of use for analytics over MapReduce. Spark's main advantage is the use of directed acyclic graph patterns combined with in-memory data processing and sharing between graphs. In-memory data storage makes shuffle stages, using intermediate results, or even querying whole datasets repeatedly, less expensive. Additionally, Spark attempts to hold as much data as possible in memory before spilling to disk resulting in major performance boosts. It can be deployed on a cluster of machines as a standalone service but can also be used on top of existing YARN installations. It can also use multiple distributed storage systems such as HDFS, Cassandra[31], HBase[32] and others.

3.3 Higher level interfaces

Hive[3] is a data warehouse software infrastructure built on top of Hadoop for managing, querying and analyzing large datasets stored in a distributed storage system. Hive was initially designed for use in batch jobs over large datasets of append only data but has since evolved into a tool capable of providing interactive query capabilities with the completion of the Stinger initiative[33], which greatly improved its speed, scalability and SQL semantics. It supports reading various file formats such as plain text files, ORC, Parquet and others. Hive provides a simple SQL-like query language, called HiveQL , which transparently transforms queries into a directed acyclic graph of MapReduce or Tez jobs which are submitted to the respective framework for execution. Support for Hive over Spark has been

[6] Image taken from https://tez.apache.org/

added lately but is not yet considered stable and is under active development. This means that Hive can work on top of all these frameworks, making it suitable for use by analysts not particularly familiar with the inner workings of said frameworks. On the other hand it still provides the flexibility to expert users of writing custom extensions to perform more sophisticated actions not supported by HiveQL.

Pig[4] is a high level procedural scripting language designed to be used on top of Hadoop for expressing analysis programs. Pig's compiler transforms the scripts provided by the user, written in Pig's own language named PigLatin, into sequences of MapReduce or Tez jobs. Its main advantages are the ease of use, as really complex tasks can be simplified into simpler easier to maintain data flow sequences, and the easily extendable nature of Pig, allowing it to crunch data from various types of sources such as files or streams and even invoke code from external programs written in different languages using User Defined Functions (UDF). In our benchmark, Pig over Tez was used, taking advantage of Tez's optimizations over MapReduce, in order to make Pig more competitive to other high level tools.

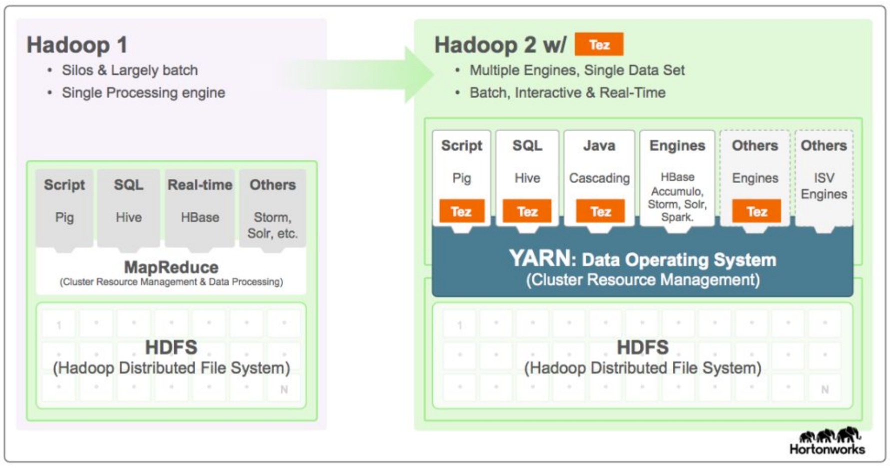
Hadoop 1
Silos & Largely batch
Single Processing engine
Script
Pig
SQL
Hive
Real-time
HBase
Others
Storm, Solr, etc.
MapReduce
(Cluster Resource Management & Data Processing)
HDFS
(Hadoop Distributed File System)
Hadoop 2 w/ Tez
Multiple Engines, Single Data Set
Batch, Interactive & Real-Time
Script
Pig
Tez
SQL
Hive
Tez
Java
Cascading
Tez
Engines
HBase Accumulo, Storm, Solr, Spark.
Others
Engines
Tez
Others
ISV Engines
YARN: Data Operating System
(Cluster Resource Management)
HDFS
(Hadoop Distributed File System)
Hortonworks

Figure 7 Tez sitting on top of YARN[7]

Spark SQL (originally known as Shark[18]) is a component of Spark[6] which provides support for manipulating and querying structured and semi-structured data in Spark programs, as well as accessing data that provide a schema such as JSON, ORC, Parquet and others. Spark SQL claims to provide high query performance, while trying to maintain

[7] Image taken from Hortonworks http://hortonworks.com/

complete compatibility with Hive, its User Defined Functions and the Hive metastore in order to run unmodified Hive queries on Spark.

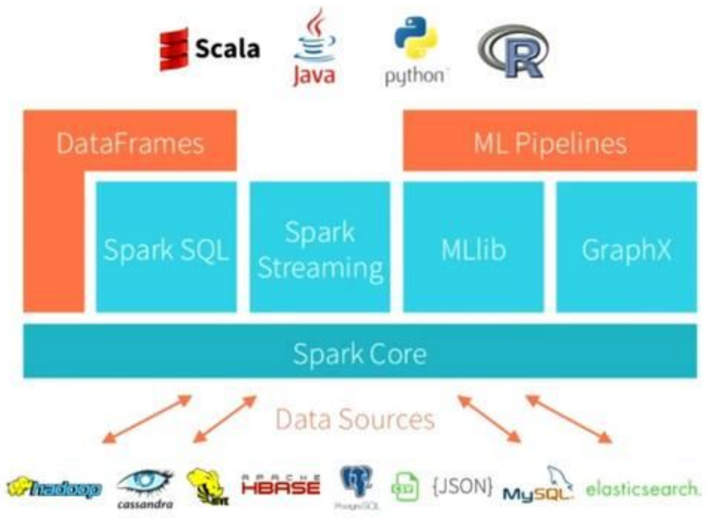

Figure 8 Spark framework[8]

8 Image taken from Apache Spark http://spark.apache.org/

Chapter 4 Experimental setup

4.1 Hardware infrastructure

In order to conduct the experiments, we had to use a variety of tools. We decided to use the Hortonworks HDP Sandbox[8], a preconfigured virtual machine image based on CentOS 6.7, that comes with all the systems needed installed. HDP Sandbox version 2.3 was used, which provides among other tools:

- Hadoop 2.7.1,
- Pig 0.15.0,
- Hive 1.2.1,
- Tez 0.7.0,
- Spark 1.3.1, and
- Ambari 2.1.0.

In order to take advantage of some new features such as window functions not supported by the Spark version coming with HDP 2.3, we further manually installed Spark 1.4.1. The Sandbox image was deployed in a Virtual Machine given

- 16 CPU virtual cores (backed by 2.3GHz Intel Xeon Sandy Bridge processors),
- 64GB of RAM, and
- 500GB of disk space.

4.2 Dataset

The dataset used for the experiments came from the TPC-H[13] Benchmark provided by the Transaction Processing Performance Council (TPC). TPC-H provides a command line tool, dbgen, in order to generate a synthetic dataset consisting of business data that modelling an industry decision support system. The schema of the database created by dbgen is the following.

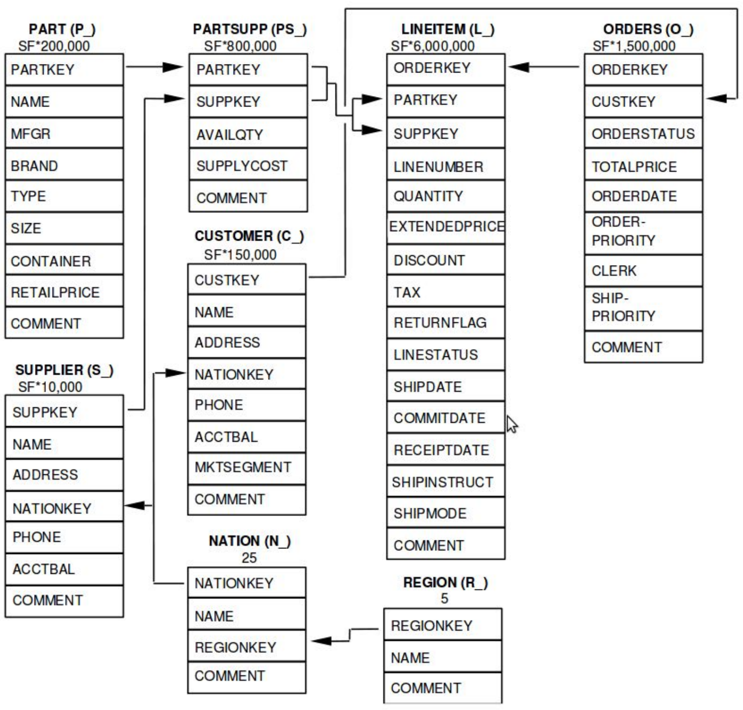

Figure 9 TPC-H database schema[9]

The dbgen tool uses a scale factor parameter in order to define the size of the dataset it is going to create. The default scale factor is 1, which creates a dataset of approximately 1GB in total for all 8 database tables. TPC-H was selected because it also provides a query set that accompanies the generated dataset, so we can have realistic experiments.

In the experiments, we are using 4 different scale datasets with scale factor 1, 25, 50 and 100. Initially, the generated tables are copied as plain files in HDFS, and then loaded to ORC and Parquet tables, to be used from the various tools. These files are further compressed using Snappy[34].

4.3 Query workload

The TPC-H benchmark also provides qgen, a command line tool that generates a series of business oriented queries in a realistic context, in order to evaluate the performance of systems running the benchmark. The tool generates a total of 22 queries that all differ from each other, have an ad-hoc nature and a high degree of complexity, cover a large percentage of the data available and can contain parameters that change in order to cope with different scale

[9] Image taken from TPC-H specification

factors. In our experiments we did not use all of the generated queries. After surveying the relevant literature, we hand-picked those used by most related works. The SQL queryset generated from qgen had to be made compatible with our tools, particularly converted to HiveQL for Hive and Spark, which would require some minor changes since both languages are close and HiveQL is constantly adding features to become compatible with SQL, and totally rewritten in Pig Latin for Pig. We used already transformed versions of the queries created by similar research projects[35][36] and only made changes in HiveQL versions wherever we thought it would be beneficial. Specifically we tried to reduce the number of queries that make use of temporary tables to calculate intermediate results, in an attempt to reduce the I/O that comes from writing intermediate results to disk between jobs. We also tried to make use of bleeding-edge features added to the latest versions of Hive and Spark, to further improve the query processing performance. Pig versions of the queries were left unchanged, except some reordering on some of the table joins where we put the largest tables first in order to increase performance; this increase in performance comes from Pig loading the right-side table in memory when using ‘replicated’ joins. Following is a short description of the queries used in our experiments.

4.3.1 Query 11

As described in the TPC-H specification, this query finds the most important parts of suppliers’ stock in a given nation. It scans the available stock for those parts that represent a significant percentage of the total value of all available parts.

Initially the query was the following as taken from project [35], requiring 2 additional temporary tables:

```
insert overwrite table q11_part_tmp
select ps_partkey, sum(ps_supplycost * ps_availqty) as part_value
from nation n join supplier s on s.s_nationkey = n.n_nationkey and
n.n_name = 'RUSSIA'
join partsupp ps on ps.ps_suppkey = s.s_suppkey
group by ps_partkey;

insert overwrite table q11_sum_tmp
select sum(part_value) as total_value
from q11_part_tmp;

insert overwrite table q11_important_stock
select ps_partkey, part_value as value
from   (   select   ps_partkey,   part_value,   total_value   from
q11_part_tmp join q11_sum_tmp ) a
where part_value > total_value * 0.0001
order by value desc;
```

We then transformed the query to the following, which required only one temporary table making use of HiveQL's support for subqueries:

```
insert into table q11_part_tmp
select ps_partkey, sum(ps_supplycost * ps_availqty) as part_value
from nation n join supplier s on s.s_nationkey = n.n_nationkey and
n.n_name = 'RUSSIA'
join partsupp ps on ps.ps_suppkey = s.s_suppkey
group by ps_partkey;

insert into table q11_important_stock
select ps_partkey, part_value as value
from  (select  sum(part_value)  as  total_value  from  q11_part_tmp)
sum_tmp
join q11_part_tmp
where part_value > total_value * 0.0001
order by value desc;
```

The final version that we transformed the query to, makes use of the window functions, introduced in Hive 0.11 and in Spark 1.4. No additional temporary tables are required for this version, which we expect to improve the query processing performance:

```
insert into table q11_important_stock
select ps_partkey, part_value as value
from (
select ps_partkey, sum(ps_supplycost * ps_availqty) as part_value,
sum(sum(ps_supplycost * ps_availqty)) over() as total_value
from nation n
join  supplier  s  on  s.s_nationkey  =  n.n_nationkey  and  n.n_name  =
'RUSSIA'
join partsupp ps on ps.ps_suppkey = s.s_suppkey
group by ps_partkey
) sum_tmp
where part_value > total_value * 0.0001
order by value desc;
```

4.3.2 Query 4

This query assesses how well the priority system of the business is working, by finding the orders that were received by the customer later than their committed date, counting them and ordering them based on priority order.

The query initially required one additional temporary table.

```
insert overwrite table q4_order_priority_tmp
select distinct l_orderkey
from lineitem
where l_commitdate < l_receiptdate;
```

```
nsert overwrite table q4_order_priority
select o_orderpriority, count(*) as order_count
from orders o join q4_order_priority_tmp t on o.o_orderkey =
t.o_orderkey and o.o_orderdate >= '1994-10-01' and o.o_orderdate <
'1995-01-01'
group by o_orderpriority
order by o_orderpriority;
```

The improved version is the following, requiring no temporary tables, by combining the two queries:

```
insert into table q4_order_priority
select o_orderpriority, count(distinct l_orderkey) as order_count
from orders o join lineitem l on o.o_orderkey = l.l_orderkey and
o.o_orderdate >= '1994-10-01' and o.o_orderdate < '1995-01-01' and
l.l_commitdate < l.l_receiptdate
group by o_orderpriority
order by o_orderpriority;
```

4.3.3 Query 6

This query finds the revenue increase a company would have by eliminating certain discounts on a particular time range. No changes were made to this query.

```
insert into table q6_forecast_revenue_change
select sum(l_extendedprice*l_discount) as revenue
from lineitem
where l_shipdate >= '1993-01-01' and l_shipdate < '1994-01-01' and
l_discount >= 0.03 and l_discount <= 0.05 and l_quantity < 24;
```

4.3.4 Query 7

This query finds the value of products shipped between certain nations in a particular time range. It required no change to the SQL generated by qgen.

```
insert into table q7_volume_shipping
select supp_nation, cust_nation, l_year, sum(volume) as revenue
from (
select n1.n_name as supp_nation, n2.n_name as cust_nation,
year(l_shipdate) as l_year, l_extendedprice * (1 - l_discount) as
volume
from supplier, lineitem, orders, customer, nation n1, nation n2
where s_suppkey = l_suppkey and o_orderkey = l_orderkey and
c_custkey = o_custkey and s_nationkey = n1.n_nationkey and
c_nationkey = n2.n_nationkey and ( (n1.n_name = 'INDIA' and
n2.n_name = 'ETHIOPIA') or (n1.n_name = 'ETHIOPIA' and n2.n_name =
'INDIA') ) and l_shipdate >= '1995-01-01' and l_shipdate <= '1996-
12-31'
) as shipping
group by supp_nation, cust_nation, l_year
order by supp_nation, cust_nation, l_year;
```

4.3.5 Query 13

This query finds relationships between customers and the size of their orders, creating a distribution of customers by the number of orders they have made. It also required no change to the SQL generated by qgen.

```
insert into table q13_customer_distribution
select c_count, count(*) as custdist
from (select c_custkey, count(o_orderkey) as c_count
from  customer  c  left  outer  join  orders  o  on  c.c_custkey  =
o.o_custkey and not o.o_comment like '%special%deposits%'
group by c_custkey
) c_orders
group by c_count
order by custdist desc, c_count desc;
```

4.3.6 Query 18

This query ranks and finds the top 100 customers based on their orders whose total quantity is above a certain level. This query requires an additional temporary table that we could not find a way to avoid by rewriting the query in a different way.

```
insert into table q18_tmp
select l_orderkey, sum(l_quantity) as t_sum_quantity
from lineitem
group by l_orderkey;

insert into table q18_large_volume_customer
select
c_name,c_custkey,o_orderkey,o_orderdate,o_totalprice,sum(l_quantit
y)
from customer c join orders o on c.c_custkey = o.o_custkey
join q18_tmp t on o.o_orderkey = t.l_orderkey and t.t_sum_quantity
> 315
join lineitem l on o.o_orderkey = l.l_orderkey
group by c_name,c_custkey,o_orderkey,o_orderdate,o_totalprice
order by o_totalprice desc,o_orderdate
limit 100;
```

4.4 Metrics

The performance of each query was measured as the total time taken to complete, as reported from each tool. The time taken to drop and create needed tables for each query was excluded from the calculations, but time taken to execute the intermediate queries for the temporary tables was still added to the result. This allowed us to compare the performance for the execution of each query over the different tools and also check how query performance scales across different scale factors. We report on the I/O cost for the execution of each query, as well as the scaling of the performance of the various tools over various dataset sizes.

In addition to that, the time taken to load the datasets from HDFS storage to their respective ORC and Parquet tables was measured, as well as the space each dataset occupied in said table formats compared to plain HDFS files.

Chapter 5 Performance results

5.1 Data loading

To perform the experiments, after generating the datasets for the different scale factors and copying the files to HDFS, the data were loaded using Hive on Tez to ORC and Parquet tables to be used from the queries. Following are the results for loading time and space taken for these tables after 3 successive runs.

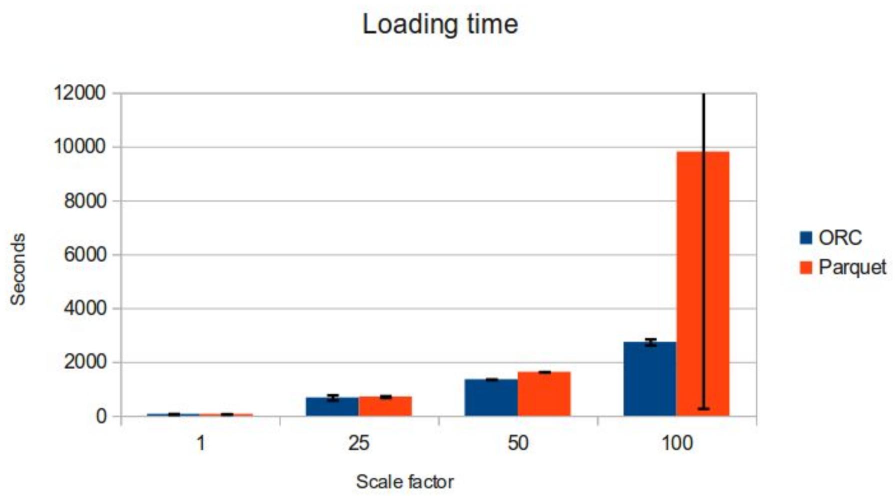

Figure 10 Loading time

As can be seen from the chart, the average loading time shows that the ORC format is almost on par with the Parquet format, with the exception of scale factor 100 where Hive needed on average more than twice the time to create the appropriate Parquet tables over ORC from plain HDFS files. During our 3 runs measuring data loading performance, loading of Parquet formatted tables took unexpectedly long, skewing our average time. By excluding the outlier we can tell that loading Parquet tables from HDFS files using Hive at the biggest scale factor is still a competitive solution, although slower than loading ORC tables.

Another comparison between ORC and Parquet file formats is the disk size the tables occupy after being loaded compared to each other and to plain HDFS files. The relevant results can be seen in the following figure.

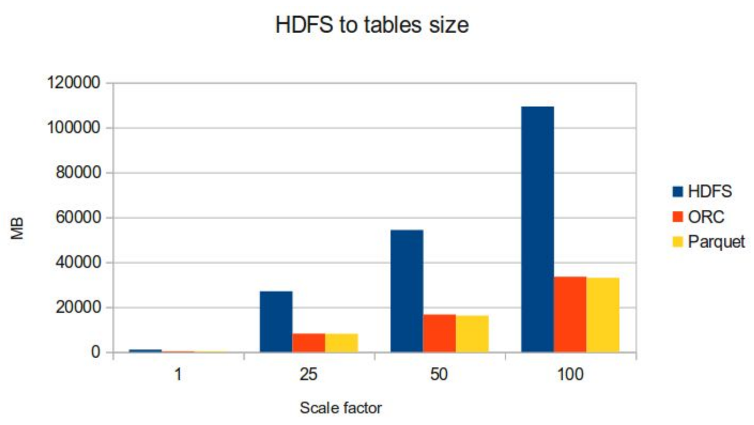

Figure 11 Size of datasets when stored in plain HDFS, ORC, and Parquet files

The level of size reduction on both formats is impressive, effectively reducing the disk space needed to 30% of the size of plain HDFS files. Both formats are really closely competing, with Parquet having a small advantage over ORC. This is because both formats are columnar, optimized to be more efficient in storage and compressed using Snappy. Testing Snappy independently on the original plain files resulted in a roughly 47% compression ratio, showing that the remaining reduction on total file size comes from the format itself. Having the data stored in such a compressed way results in a massive performance improvement in disk I/O.

5.2 Time

Following are the results for the performance of queries measuring response time on scale factor 100 across different frameworks.

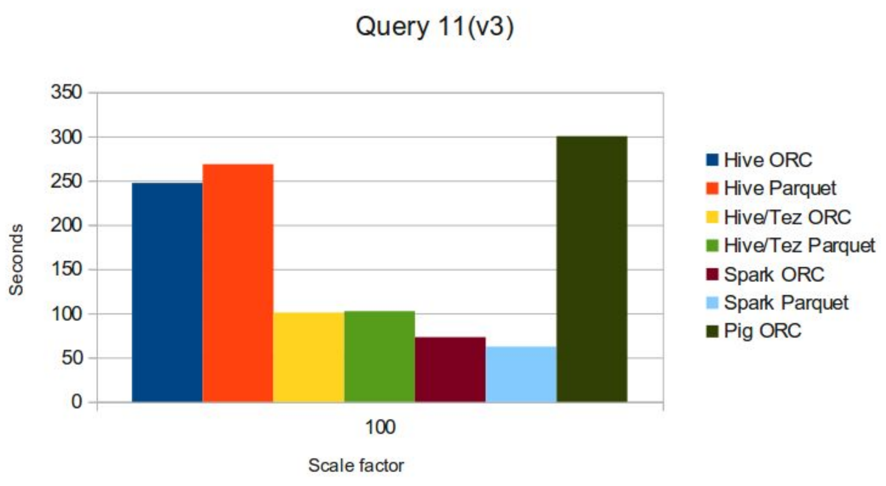

Figure 12 Query 11 response time

As can be seen from the graph in Figure 12, Spark is the fastest framework for query 11, with both file formats competing with each other in response time. Hive on Tez follows closely with similar results on both formats, while Hive over MapReduce and Pig respond way slower, over twice the time of others.

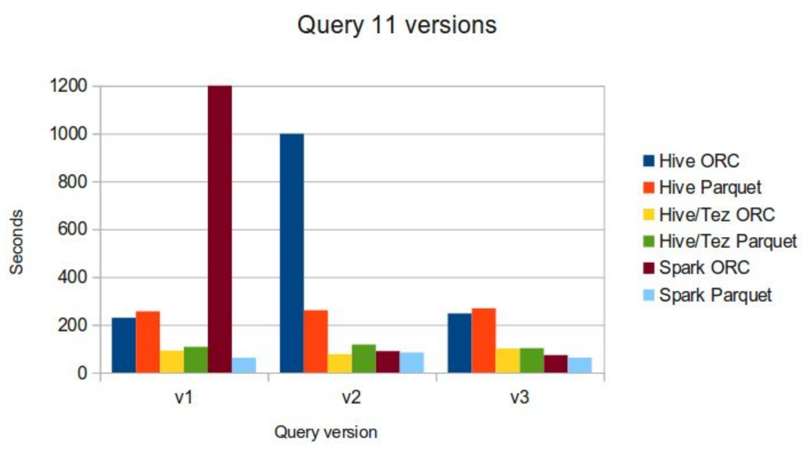

Figure 13 Query 11 versions response time

Taking a closer look at the different versions of query 11 at scale factor 100 in Figure 13, reveals that query rephrasing did not have a major impact on response time in general, with the exception of Spark using ORC, which oddly had an enormous response time of 4270 seconds in version 1.

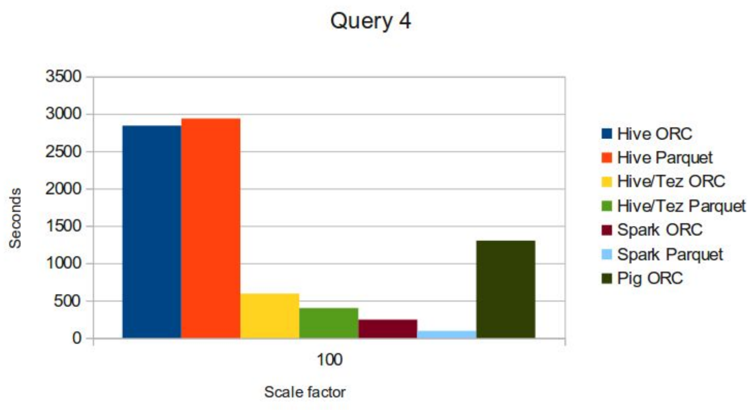

Figure 14 Query 4 response time

Figure 14, shows the response time for query 4, where again Spark is the fastest, followed by Hive on Tez. Spark with Parquet responds in less than half the time of Spark with ORC and in 25% of the time of Hive on Tez with Parquet. Pig and Hive on MapReduce are vastly slower, with Pig responding in 10 times the time of Spark and Hive on MapReduce almost 30 times slower on both file formats. This query uses the largest table of the dataset when joining, showing the bottleneck of MapReduce model and execution plan.

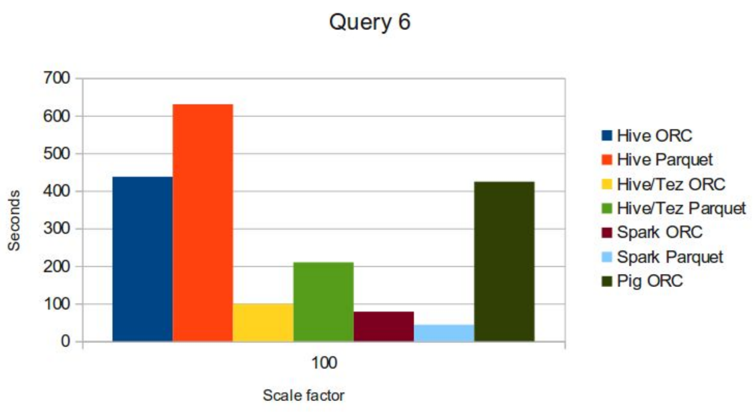

Figure 15 Query 6 response time

In query 6, as can be seen in Figure 15, Spark with Parquet is still the fastest framework. Spark with ORC and Hive on Tez with ORC are following and closely competing with each other while Hive on Tez with Parquet seems slower. Once again, Hive on MapReduce with both file formats and Pig are considerably slower.

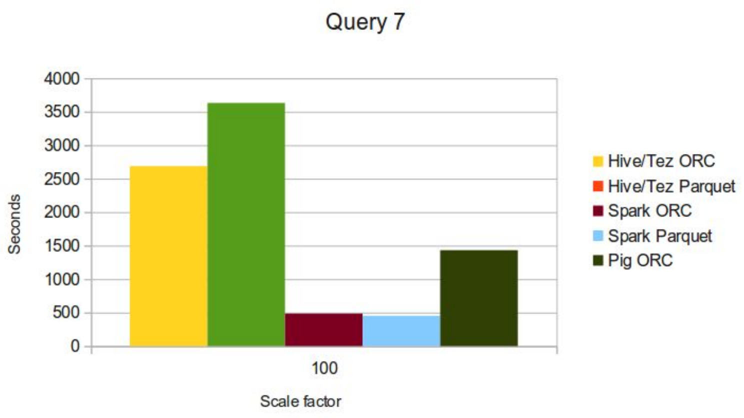

Figure 16 Query 7 response time

Figure 16, shows the response times for query 7, where Spark provides the fastest response time with both file formats. Pig is following with more than twice the time of Spark and Hive on Tez comes last with almost 5 times slower response time using ORC and 7 times slower using Parquet on scale factor 100. Unfortunately we could not get figures for Hive over MapReduce for query 7 because of the query constantly crashing, but we guess that they would not be competitive based on previous results.

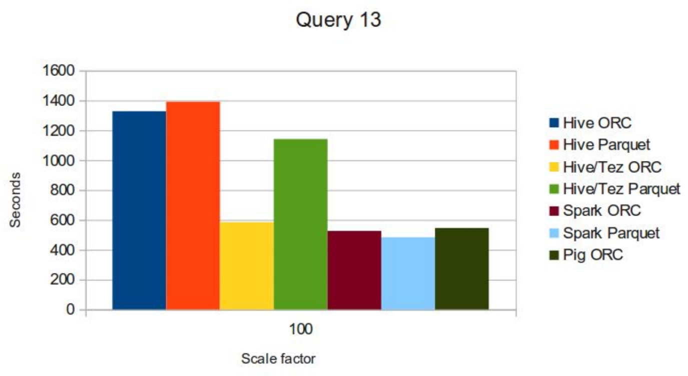

Figure 17 Query 13 response time

In query 13, as shown in Figure 17, Spark with Parquet is still the fastest framework. Spark with ORC follows closely, with Pig interestingly enough having similar response times and Hive on Tez with ORC following both without great difference. Hive on Tez with Parquet together with Hive on MapReduce respond in roughly twice the time of the other frameworks. These results might be because the query is not complex enough, being a simple aggregation over a join between two tables, not allowing the difference of framework query plan optimizations to become notable.

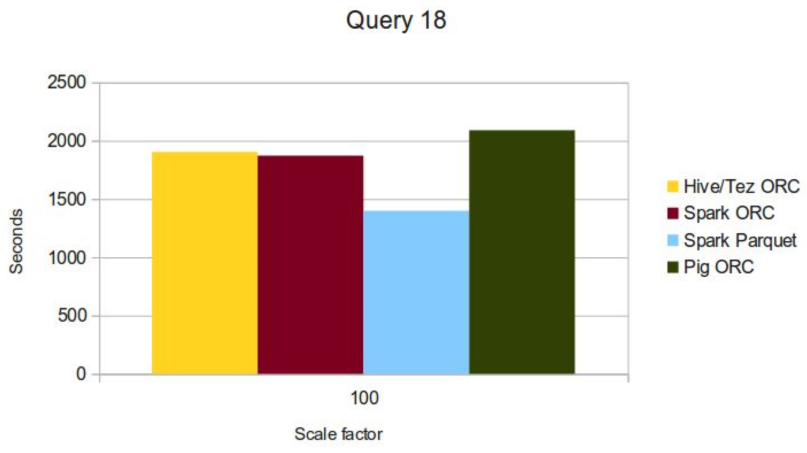

Figure 18 Query 18 response time

Figure 18, reveals some interesting results about query 18. Spark with Parquet is still the fastest in response time, but the Spark with ORC, Hive on Tez with ORC and Pig are following with not a huge difference, closely competing with each other. This performance, is possibly due to the low complexity of the query, similar to query 13. Queries for Hive over MapReduce with both file formats and Hive on Tez with Parquet were constantly crashing, so we could not get any results to compare.

5.3 I/O

Another measurement we took when running the experiments is the total I/O for each job on every framework. We define the total I/O as the sum of the bytes read from and written to HDFS and the bytes read from and written to plain files. Following are the results of our benchmarks.

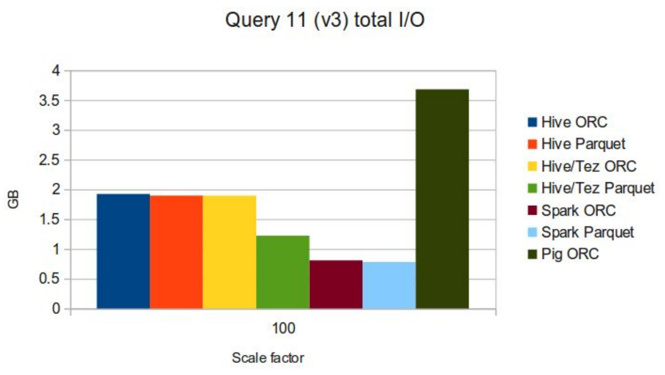

Figure 19 Query 11 I/O

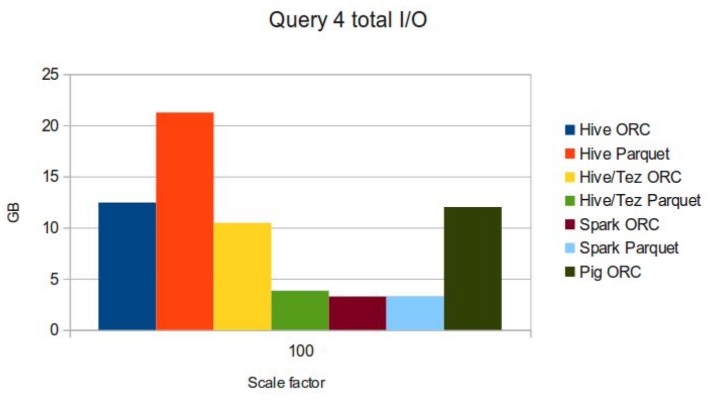

Figure 20 Query 4 I/O

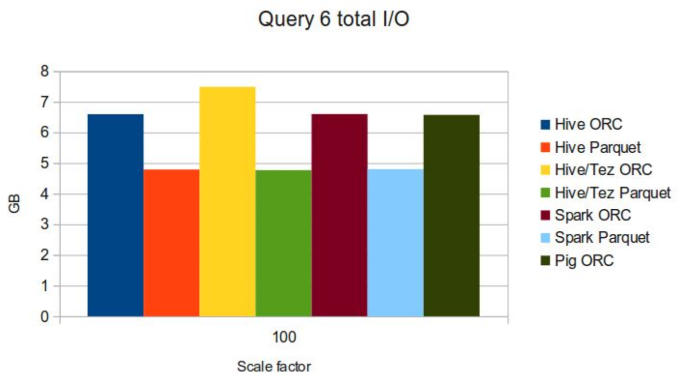

Figure 21 Query 6 I/O

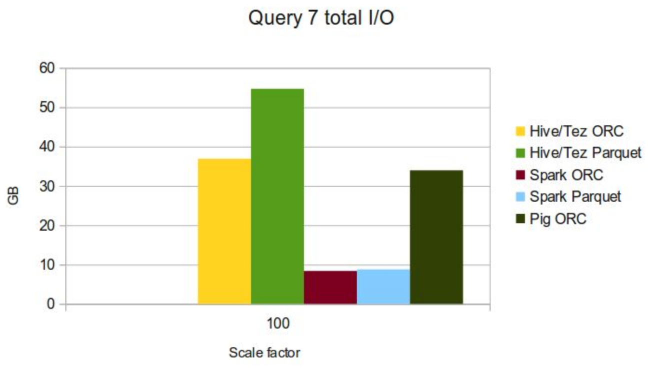

Figure 22 Query 7 I/O

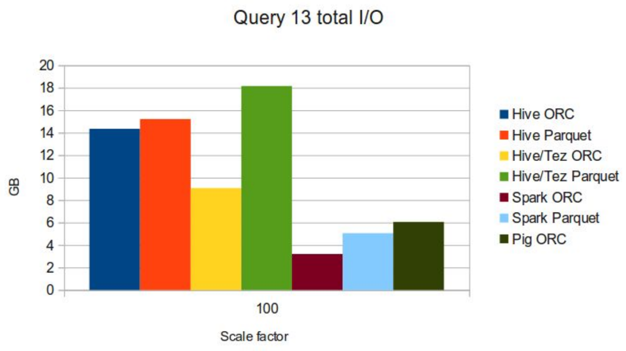

Figure 23 Query 13 I/O

In general we can see a trend where Spark has less data I/O than Hive and Pig. We can also notice, that I/O measurements are corresponding to the performance of the queries in last section, as data I/O adds overhead to query execution, resulting to longer response times.

5.4 Scaling

Following are the results of queries measuring response time across all the scale factors. Unfortunately we were not able to run vanilla Hive for all the queries in all scale factors. We included the results for it in scale factor 1 and 100 wherever possible to compare with other tools, although as evident from our earlier results we do not expect it to have a competitive performance.

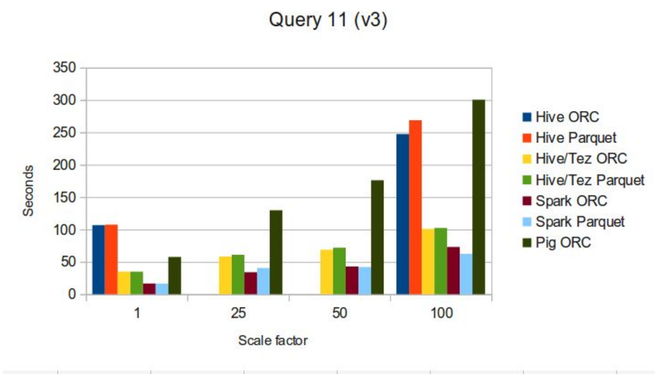

Figure 24 Query 11 scaling

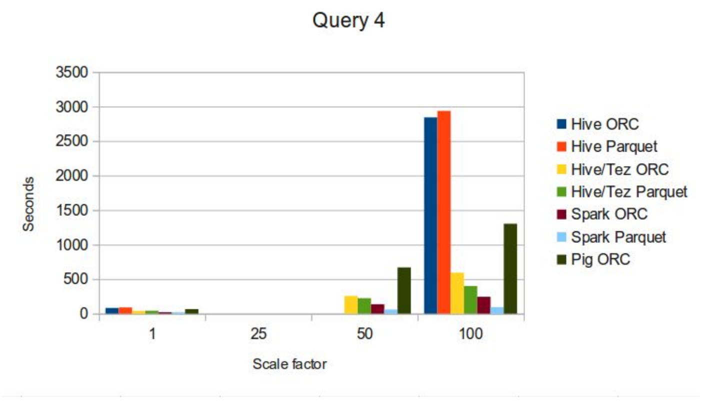

Figure 25 Query 4 scaling

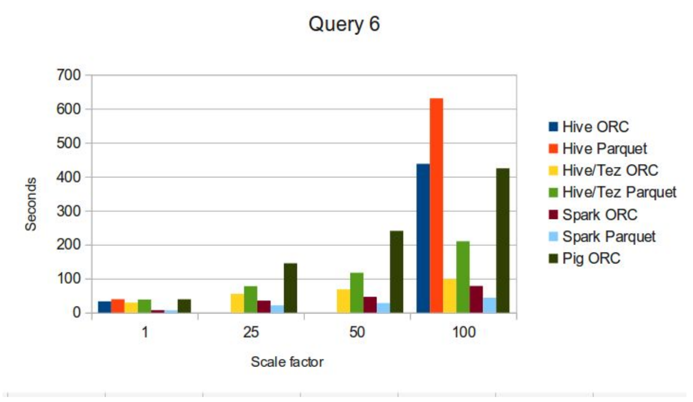

Figure 26 Query 6 scaling

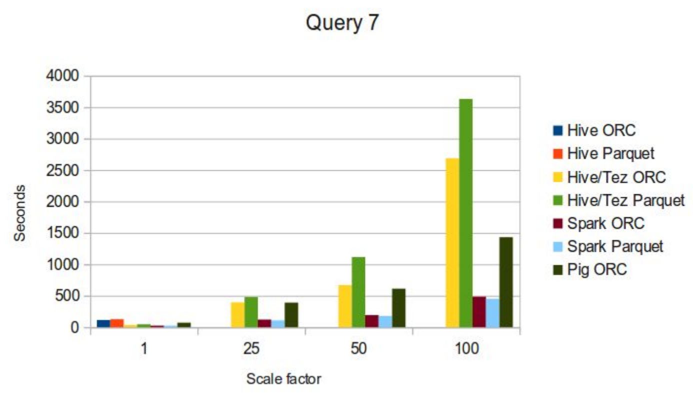

Figure 27 Query 7 scaling

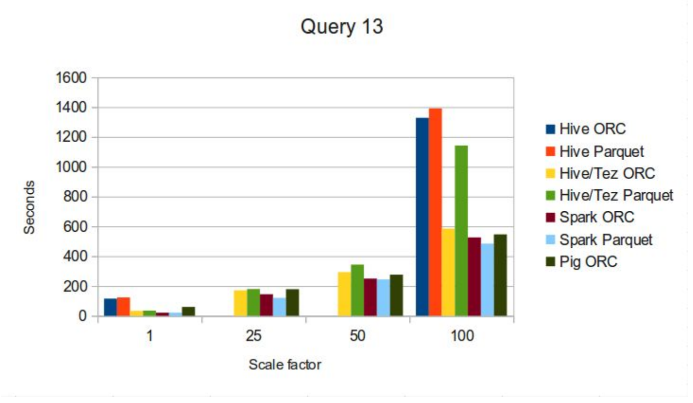

Figure 28 Query 13 scaling

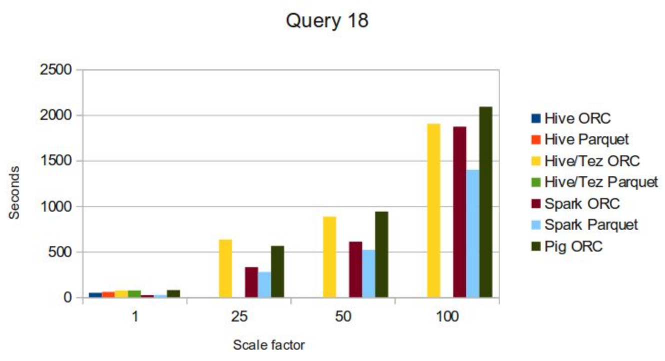

Figure 29 Query 18 scaling

When it comes to scaling, we can see from the performance results that Spark scales better than the other tools when measuring response time. In worst cases, response time is proportional to the change in scale factor. Hive on the other hand, does not provide a clear view of its scaling. In some queries it scales good enough when increasing the scale factor, similar to Spark, but in some cases the increase in response time when increasing scale factor is bigger than expected. Pig, while being among the last when it comes to response time, scales proportionally to the increase in scale factor in almost all the queries.

Chapter 6 Discussion

6.1 Performance

As can be seen from the performance results, the right choice of frameworks is a major decision when it comes to query response times, the most critical aspect of interactive queries. Spark is most of the times the fastest to use, with a big advantage over the other frameworks. This is mainly due to the design implementation of Spark, having an advanced DAG execution engine and the ability to optimize the core engine based on the type of workload and application. Spark can provide a significant boost in performance of various types of applications, such as SQL queries, streaming applications and machine learning alongside general Big Data tasks. In addition to that, Spark's well known in-memory capabilities, where data can be kept in memory instead of disk, provide fast speeds. Data for intermediate jobs can be held in memory, bypassing the disk I/O overheads, and thus data frequently used can be stored in memory to provide blazing fast access. In general, Spark tries to fit as much as it is possible in memory before spilling to disk, and this is its main advantage over other frameworks, which is clearly shown in the performance results. Spark is not working only in memory though, it can also work on disk, losing performance but still managing to be faster than the other SQL on Hadoop frameworks.

Hive over MapReduce, on the other hand, is clearly the slowest showing the limitations of the MapReduce model when it comes to interactive, real-time queries, having response times orders of magnitude greater than Spark. Being designed with long running batch jobs in mind, makes it clear that it cannot cope with modern interactive needs and workloads. Every query has to be modeled in mapreduce jobs, most of the times multiple of them, in order to calculate the results. This adds overhead, majorly impacting the response time. Tez solves these issues of Hive, by adding to it features that are trying to provide interactive query compatibility. Tez enables Hive to process data that would take multiple jobs to complete in MapReduce, in a single Tez job by using DAGs of tasks, in other words chaining multiple map and reduce tasks in a tree structure. Tez also dynamically reconfigures the execution plan and data flows at runtime to achieve better results. These additions among others, as seen from the performance results, greatly improve the capabilities of Hive at dealing with interactive query workloads. Hive on Tez response times are most of the times more than twice as fast than vanilla Hive and some times 80% faster. It is still slower than

Spark though and probably one of the main issues about that is the constant flushing to disk of the intermediate results adding overhead depending on the job.

Pig, although running on Tez, does not seem to have a competitive performance against the other frameworks. In most of the queries, its response time is on par with vanilla Hive, with the exception of some queries, like query 4 and query 7 where it responds in half the time of Hive but still slower than Hive on Tez and Spark. The best result of Pig is in query 13 where it is straightly competing with Spark, but overall its interactive query capabilities are not exactly satisfying. The main power of Pig though, resides in the ease of writing really complex queries into simple data flow sequences using its scripting language.

Apart from frameworks, choosing the appropriate file format to store the data in is a choice that has to be made. ORC, introduced by Hive, and Parquet, designed from the beginning to be framework agnostic, while being columnar formats implemented in similar ways, seem to have some visible differences when it comes to performance. From the experiment results we can see that Parquet is the fastest format when used from Spark in all queries with ORC following closely. On the other hand, Hive doesn't provide a clear view on which file format is faster to use. On some queries, ORC is on par with Parquet, on other queries ORC is almost twice as fast as Parquet and in some queries, like query 18, the opposite happens. Noticing the results of the data loading time and the size of the final data using both formats, where both are really close to each other, the final decision of which file format is better is not exactly clear. When using Spark we could say that Parquet is better but when using Hive it is quite arguable.

In addition to that, alterations on the query expression can affect the query execution plans, possibly increasing the performance of the query response time. In particular, when testing query 11 using 3 different query expressions in Spark, by looking at the produced execution plans we came up with the following results. In versions 1 and 2 of the query, where temporary tables are required to compute intermediate results, the query execution plan showed that after various filters, joins and aggregations, a cartesian product took place in order to join everything together and finally filter the results based on the condition of the part value in order to come up with the final result set. In version 3 of the query, making use of the window function, while every other step is similar, this cartesian product is replaced by a window function UDF. While in our case this did not end up in a significant increase in the performance of the query, it clearly shows that well written query expressions or even rephrased query expressions in order to make use of newly added features can produce an improved execution plan, which could possibly result into a boost in performance.

6.2 Problems faced

While setting up the infrastructure needed and conducting the experiments a few problems have arisen. Initially we had to decide on which dataset the experiments should be conducted on. An easy solution would be creating a synthetic dataset in whichever size we wanted to test the frameworks. This solution, though, would pose the problem of having to find a realistic queryset to use, testing several aspects of the frameworks. We also considered using real world, large, publicly available datasets such as the City of Oakland datasets[37] or the University of Maryland datasets[38], but the problem of finding realistic and exhaustive queries would still remain. The Yahoo Cloud Serving Benchmark (YCSB)[39] could also be used, but would require quite a bit of effort to make it compatible with all of our tools. Instead, we decided to use the dataset and queryset of the TPC-H benchmark. As TPC-H is an industry standard in benchmarking database performance, this would ensure that our experiments test real world scenarios, have several degrees of complexity depending on each query, would produce meaningful results for our comparison, and our results would have greater impact.

Another problem faced was deploying the tools needed for the experiments. We chose to use the Hortonworks HDP Sandbox and deploy it in a virtual machine, as it came with all the needed frameworks preinstalled. Initially we deployed version 2.3-beta, the latest at the time, but still not published as stable. Various problems and especially a bug of Ambari[40], the server management tool, that would not let us change the server settings, lead us to fall back to version 2.2 to develop our scripts and queries and get them to a working condition. In the meantime, version 2.3 reached stable status and was published by Hortonworks, so we installed it again and transferred everything there. This allowed us to test our queries against almost the latest version of each framework.

For query 11, we tested various versions of it by altering the query expression. In particular, in our first improved version we tried to combine the originally two intermediate queries into a single query, and in the third version we took advantage of the window function provided by Hive to combine all query subexpressions into just one query. The current version of Spark installed, though, did not support this kind of functionality yet. We proceeded installing the latest version of Spark manually, in order to have the ability to test all frameworks equally.

Several problems have also been faced when running the queries. In many occasions queries would stop running, throwing exceptions of out of memory errors or would simply hang without any output. This has been solved by manually tweaking the memory size of several settings such as Java heap size, the size provided to containers of Yarn and the driver used by Spark. Most of these errors had been solved but still we were not able to make everything run. We chose for the experiments the queries that were most commonly used in other similar benchmarks but still we had some queries that we could not get results for some frameworks. This whole ordeal proved to be a tedious process, tweaking every single query for each tool manually until properly working. Nevertheless, we strived to have complete measurements for the biggest scale factor, with the exception of Hive over MapReduce that we knew beforehand would not be quite competitive.

6.3 Knowledge gained

During the period of conducting the project I have gained knowledge and experience in several aspects of systems capable or running queries on top of Hadoop. I have acquired a deeper understanding in the various frameworks I was already introduced to in courses, such as Hive and Pig, and got to know new frameworks, such as Spark and Tez, gaining information about their implementation designs and differences, noticing how they perform against each other. I acquired significant hands-on experience on these bleeding edge tools, filling the constantly growing need of accessing and manipulating data on large scale datasets using a range of queries, from simple SPJ queries to even complex analytic queries. I also gained the experience of issuing optimal queries to maximize performance using HiveQL and Pig Latin through our experiments. Last, I got a deeper understanding in optimizations at the storage level, by using the ORC and Parquet file formats and researching their implementation decisions.

In addition to that, apart from using the tools, I learned how to configure the frameworks in order to make full use of the hardware resources and how to set up, manage and monitor a “cluster” with these tools, using Ambari. I performed various system administration tasks through the duration of the project, gaining valuable hands-on experience.

In our meeting with SAS, I got to see through our conversation with people working and using these technologies, how they are used in the industry and what kind of workloads are these technologies usually dealing with. Their feedback regarding our project's topic was more than encouraging, revealing the need of such a comparison of frameworks.

Chapter 7 Conclusions

Modern workloads, demanding interactive/real-time query capabilities on really large datasets have set the need for tools to be created that are able to handle those requirements. MapReduce is no longer a viable option on these type of workloads as its purpose is dealing with long running batch jobs. In this project we tried to evaluate the major competitors of the new post-MapReduce era, which provide this kind of capabilities on top of Hadoop. We have set up a benchmark where these frameworks were tested against the same workload, using the same resources, in order to compare them, mainly based on their query response time, one of the most important metrics when it comes to interactive queries. We also tested those tools using two different file formats for storing data that are focused on access speed and compression.

Our performance results showed that Spark, a framework making heavy use of memory in order to increase performance, is in most cases the winner in query response time, followed by Hive on Tez. Hive on MapReduce and Pig come last based on their response time, with a great difference, mainly because they are not designed to provide interactive query capabilities. When it comes to file formats, being almost similarly implemented, the differences where not huge. Both ORC and Parquet were competing closely in terms of performance, with Parquet having a slight advantage because of more stable performance across different queries. Although not having benchmarked the queries over plain HDFS files, we can assume that performance would be significantly worse, based on the I/O that would be needed for reading the uncompressed files and the lack of optimizations the two columnar formats provide. Our results are not dictating a definite final choice on which framework is the best for interactive queries on Big Data, since these frameworks are still being updated and are under heavy development, constantly adding new features and fixes pushing their performance even more. However, our results are showing the current state of frameworks supporting interactive/real-time query capabilities, which clearly shows a great future to be expected from these technologies.

Our experiments were conducted using almost the default settings of each framework with the exception of memory tweaks in order to maximize our hardware resources use. Further benchmarks can be conducted, with further optimizations of these tools using newly added features and techniques that increase their performance. In addition to that, optimizations can be made on job level, by tweaking different values of various settings to increase performance, such as the number of mappers/reducers, using cache and other for each query, or even trying to research and implement a tool that would automate the process, finding the optimal configurations for each framework and specific query. Moreover, performance can be pushed even more, by fine tuning the datasets and file formats for even faster access. ORC and Parquet can be partitioned based on the type and needs of different queries or even change their settings, such as stripe size for ORC and group size for Parquet, to achieve maximum performance. Another interesting benchmark, that we did not have the time to make, is the difference in performance of these frameworks when vertical scaling the system resources against horizontal scaling. Frameworks could react differently when adding resources to the underlying systems, making them more powerful than distributing the data processing across more, lower powered, machines. Testing the frameworks against more datasets would also produce interesting results, such as which kind of workloads perform better on specific tools.

Bibliography

1. MapReduce: Simplified Data Processing on Large Clusters http://research.google.com/archive/mapreduce.html
2. Apache Hadoop https://hadoop.apache.org/
3. Apache Hive https://hive.apache.org/
4. Apache Pig https://pig.apache.org/
5. Apache Tez https://tez.apache.org/
6. Apache Spark http://spark.apache.org/
7. Hadoop HDFS http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
8. Hortonworks HDP Sandbox http://hortonworks.com/products/hortonworks-sandbox/
9. Hive ORC https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC
10. Apache Parquet https://parquet.apache.org/
11. Hadoop YARN http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
12. Spark SQL http://spark.apache.org/sql/
13. TPC-H benchmark http://www.tpc.org/tpch/
14. SAS http://www.sas.com/en_us/home.html
15. Big Data Benchmark https://amplab.cs.berkeley.edu/benchmark/
16. Amazon Redshift https://aws.amazon.com/redshift/
17. Cloudera Impala http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html

18. Shark, Spark SQL, Hive on Spark , and the future of SQL on Spark https://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html

19. Which SQL engine leads the way? IBM benchmark http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=WH&infotype=SA&appname=SWGE_IM_EZ_USEN&htmlfid=IMW14799USEN&attachment=IMW14799USEN.PDF&cm_mc_uid=29291763661414362135447&cm_mc_sid_50200000=1441363916

20. TPC-DS benchmark http://www.tpc.org/tpcds/

21. IBM Big SQL http://www.ibm.com/developerworks/library/bd-bigsql/

22. Open-Source SQL-on-Hadoop Performance http://radiantadvisors.com/wp-content/uploads/2014/04/RadiantAdvisors_Benchmark_SQL-on-Hadoop_2014Q1.pdf

23. PrestoDB https://prestodb.io/

24. InfiniDB (dead project) https://en.wikipedia.org/wiki/InfiniDB

25. InfiniDB file format https://en.wikipedia.org/wiki/InfiniDB

26. 5 ways to make our hive queries run faster http://hortonworks.com/blog/5-ways-make-hive-queries-run-faster/

27. Tune our Apache Spark jobs http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/

28. Improving Pig Data Integration performance with join https://www.xplenty.com/blog/2014/05/improving-pig-data-integration-performance-with-join/

29. Optimizing ORC files for query performance https://streever.atlassian.net/wiki/display/HADOOP/Optimizing+ORC+Files+for+Query+Performance

30. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems http://web.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-11-4.pdf

31. Apache Cassandra http://cassandra.apache.org/

32. Apache Hbase http://hbase.apache.org/

33. The Stinger Initiative http://hortonworks.com/blog/100x-faster-hive/

34. Snappy compression http://google.github.io/snappy/

35. Running TPC-H queries on Hive https://issues.apache.org/jira/browse/HIVE-600

36. Running TPC-H queries on Pig https://issues.apache.org/jira/browse/PIG-2397

37. City of Oakland https://data.oaklandnet.com/browse

38. University of Maryland http://www.start.umd.edu/data-and-tools/start-datasets

39. YCSB https://labs.yahoo.com/news/yahoo-cloud-serving-benchmark

40. Apache Ambari https://ambari.apache.org/