Evaluating Presto as an SQL on Hadoop Solution

A Case at Truecaller

Sahir Ahmed
2016

Bachelor of Arts
Systems Science

Luleå University of Technology
Department of Computer science, Electrical and Space engineering
Evaluating Presto as an SQL on Hadoop solution

*a case at truecaller*

Sahir Ahmed – VT2016
Abstract

Truecaller is a mobile application with over 200 million unique users worldwide. Every day truecaller stores over 1 billion rows of data that they use to analyse for improving their product. The data is stored in Hadoop, which is a framework for storing and analysing large amounts of data on a distributed file system. In order to be able to analyse these large amounts of data the analytics team needs a new solution for more lightweight, ad-hoc analysis. This thesis evaluates the performance of the query engine Presto to see if it meets the requirements to help the data analytics team at truecaller gain efficiency. By using a design-science methodology, Presto’s pros and cons are presented. Presto is recommended as a solution to be used together with the tools today for specific lightweight use cases for users that are familiar with the data sets used by the analytics team. Other solutions for future evaluation are also recommended before taking a final decision.

Keywords: Hadoop, Big Data, Presto, Hive, SQL on Hadoop
Table of contents

Abstract .................................................................................................................................... 2
Table of contents ......................................................................................................................... 3
Table of figures ............................................................................................................................. 5
Table of tables ............................................................................................................................... 6
Glossary ......................................................................................................................................... 7

1. Introduction ................................................................................................................................. 1
   1.1 Problem statement .................................................................................................................. 1
   1.1.1 Problem description ......................................................................................................... 1
   1.1.2 Research questions ......................................................................................................... 1
   1.1.3 Purpose ........................................................................................................................... 1
   1.2 Motivations for research ...................................................................................................... 2
   1.3 Delimitations ....................................................................................................................... 2

2. Background ................................................................................................................................ 3
   2.1 Truecaller ............................................................................................................................. 3
   2.2 Big Data ................................................................................................................................ 3
      2.2.1 What is Big Data? ........................................................................................................... 3
      2.2.2 Data Analytics ............................................................................................................. 5
      2.2.3 Data Warehouse .......................................................................................................... 5
   2.3 Hadoop .................................................................................................................................. 7
      2.3.1 What is Hadoop ............................................................................................................. 7
      2.3.2 Truecaller and Hadoop ............................................................................................... 7
      2.3.3 MapReduce .................................................................................................................. 8
      2.3.4 HDFS .......................................................................................................................... 10
      2.3.5 YARN .......................................................................................................................... 11
         2.3.5.1 Slider ..................................................................................................................... 13
      2.3.6 SQL on Hadoop ............................................................................................................ 13
         2.3.6.1 Hive ....................................................................................................................... 13
         2.3.6.2 Presto ...................................................................................................................... 15

3. Literature review ......................................................................................................................... 17
   3.1 Purpose of literature review ................................................................................................. 17
   3.2 Benchmarks .......................................................................................................................... 17
      3.2.1 Fast-data-hackathon ..................................................................................................... 17
      3.2.2 Renmin University ........................................................................................................ 18
      3.2.3 Commercial benchmarks ............................................................................................ 19
         3.2.3.1 Pivotal (HAWQ) ................................................................................................... 19
         3.2.3.2 Cloudera (Impala) ................................................................................................. 20
   3.3 Hive not meant for low-latency querying ............................................................................. 21

4. Research methodology ................................................................................................................. 22
   4.1 Identify problems & Motivate ............................................................................................... 22
   4.2 Define Objectives for solution ............................................................................................. 23
   4.3 Design & Development ......................................................................................................... 23
Table of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig 1.</td>
<td>Exponential growth of data (Fishman, 2014, July 14)</td>
<td>4</td>
</tr>
<tr>
<td>Fig 2.</td>
<td>The Big data process and sub processes (Gandomi &amp; Haider, 2015)</td>
<td>5</td>
</tr>
<tr>
<td>Fig 3.</td>
<td>Star schema (Connolly &amp; Begg, 2005)</td>
<td>6</td>
</tr>
<tr>
<td>Fig 4.</td>
<td>Simplified MapReduce dataflow (White, 2010)</td>
<td>7</td>
</tr>
<tr>
<td>Fig 5.</td>
<td>MapReduce flow with one reducer (White, 2010)</td>
<td>9</td>
</tr>
<tr>
<td>Fig 6.</td>
<td>MapReduce data flow with multiple reducers (White, 2010)</td>
<td>10</td>
</tr>
<tr>
<td>Fig 7.</td>
<td>HDFS Architecture (Srinivasa &amp; Mupalla, 2015)</td>
<td>11</td>
</tr>
<tr>
<td>Fig 8.</td>
<td>Applications running on YARN (White, 2010)</td>
<td>12</td>
</tr>
<tr>
<td>Fig 9.</td>
<td>YARN components (Apache Hadoop, 2016)</td>
<td>13</td>
</tr>
<tr>
<td>Fig 10.</td>
<td>Hive Architecture (Thusoo et al., 2009)</td>
<td>15</td>
</tr>
<tr>
<td>Fig 11.</td>
<td>Presto Architecture (Teradata, 2016)</td>
<td>16</td>
</tr>
<tr>
<td>Fig 12.</td>
<td>fast-data-hackathon results (Treasure Data Blog, 2015, March 20)</td>
<td>18</td>
</tr>
<tr>
<td>Fig 13.</td>
<td>Renmin University results for 100-node cluster (Chen et al., 2014)</td>
<td>19</td>
</tr>
<tr>
<td>Fig 14.</td>
<td>Hawq’s speed up ratio vs. Impala (Soliman et al., 2014)</td>
<td>19</td>
</tr>
<tr>
<td>Fig 15.</td>
<td>Cloudera’s Impala benchmark (Cloudera, 2016, February 11)</td>
<td>20</td>
</tr>
<tr>
<td>Fig 16.</td>
<td>Design Science Research Methodology Peffers, Tuunanen, Rothenberger &amp; Chatterjee, 2007.</td>
<td>22</td>
</tr>
<tr>
<td>Fig 17.</td>
<td>Query execution time in seconds</td>
<td>27</td>
</tr>
</tbody>
</table>
Table of tables

Table 1. Glossary........................................................................................................................................7
Table 2. What is big data? (Cervone, 2015)............................................................................................4
Table 3. RDBMS compared to MapReduce (White, 2010)..........................................................................8
Table 4. Requirements ............................................................................................................................23
Table 5. Cluster details.............................................................................................................................25
# Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>Open source Java framework for distributed data storage</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop distributed file system</td>
</tr>
<tr>
<td>YARN</td>
<td>(Yet another resource negotiator) resource manager for Hadoop</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational database management system</td>
</tr>
<tr>
<td>Node</td>
<td>Commodity computer (hardware) used by HDFS</td>
</tr>
<tr>
<td>Cluster</td>
<td>A group of connected nodes used by HDFS</td>
</tr>
<tr>
<td>Hive</td>
<td>SQL data warehouse for Hadoop</td>
</tr>
<tr>
<td>MapReduce</td>
<td>Programming model for processing data on Hadoop</td>
</tr>
<tr>
<td>Presto</td>
<td>SQL on top of Hadoop solution</td>
</tr>
<tr>
<td>Slider</td>
<td>Application to run non-Yarn applications on YARN</td>
</tr>
</tbody>
</table>

Table 1. Glossary
1. Introduction

This section introduces the reader to truecaller, the company where this study is conducted and gives some background information about the current situation regarding the data that is stored and analysed. Following that information a background regarding the data warehouse and tools used for data analytics within the analytics team of truecaller is explained. Hadoop is one of the main tools used and described on a fairly high-level.

Truecaller is a mobile app used by over 200 million users worldwide. At truecaller today, the daily incoming data (about half a billion rows) is stored in a system distributed file system using the open source Java framework Hadoop. The collected data is used by all the departments of the company for improving the product and taking new decisions. In order to analyse the large amounts of data, the Data-Analytics team is looking for a new low latency solution that can be used as a complement to their standard Hadoop tools, which in some cases are considered too slow. The goal of this research is to identify, test and evaluate a number of tools that fits the team’s requirements and could potentially be implemented.

1.1 Problem statement

“Running ad-hoc queries on Hive is too slow for data discovery” – Björn Brinne, Head of Data-Analytics at Truecaller

1.1.1 Problem description

The problem within the data-analytics organisation at truecaller is that the use of Hive is not fast enough for ad-hoc querying and on-the-fly data discovery. The users run their queries in Hive and longer queries with multiple joins and aggregations can take up to several hours to execute. After the query has run, the user might not be satisfied with the outcome and will have to adjust and rerun the query, sometimes in multiple iterations. This makes the whole process ineffective and holds back the data scientists and other users from using their creativity to add more value to their job and company because of poor performance of the tools. It also limits them to just follow what is prioritized at that time and leads to some work not being done or pushed really far back in priority and. This can also make the analytics organisation less innovative.

1.1.2 Research questions

1. How much faster than Hive is Presto on truecaller’s Hadoop cluster?
2. Which type of use cases is Presto suitable/not suitable for?

1.1.3 Purpose

The purpose of this project is to identify, test and implement a solution to support data-discovery and ad-hoc querying of data in the truecaller data warehouse. The solution is aimed to complement Hadoop and will hopefully be implemented and used by the analytics-team at truecaller. There are several commercial and open source products are supposed to provide that functionality to the users. Those products often vary in performance and depend on the environment that they are set up in. Therefor the aim and purpose of this research is only meant to solve the issues and use-cases at truecaller and is not intended to be used as a general recommendation or solution for other companies to rely upon. However, the outcomes can provide the readers with an overview that can
be compared to other articles on the subject. The main goal is to advice truecaller on which of the identified solutions to implement and use as an analytical tool.

1.2 Motivations for research
The main motivation is to test and evaluate newer techniques within the growing Big Data field and to compare them with the industry standard tool Hive. A secondary motivation is to provide support and help to the data-analytics team at truecaller, which is undergoing rapid growth, and wishes to find a more suitable tool for faster querying and analysis of their daily incoming data.

1.3 Delimitations
The project will take place halftime under 4 months and will be based on pre-specified use cases for truecaller. One solution is to be tested. First on a test environment like e.g. a virtual one node cluster and as a second stage in the production environment. The solution is free open source to keep the costs down. This project will mostly focus on how fast the data is presented and will not go in depth on how the techniques work behind the scenes, due to the subject of Information Systems Science. The data sets will be predefined and the aim so to finally use the data warehouse and daily incoming data.
2. Background

2.1 Truecaller

Truecaller is a Swedish mobile application company based in Stockholm. Their flagship application; truecaller is developed to provide users with information about wanted and unwanted calls from unknown numbers. By doing that truecaller helps the users identifying spam calls and messages, giving them possibilities to block those numbers and also share this information with the rest of the user base (Truecaller, 2016).

Truecaller has today over 200 million users and is steadily growing. The largest user group is located in India where there are more than 100 million users. In India alone, truecaller helps identify about 900 million calls and 120 million spam calls every month. The company stores over 2 billion names and numbers in its data warehouse where it is later analysed for further actions. Working together with several telecom providers helps truecaller to reach out to new users in different countries (Choudury, 2015, October 12).

In India where truecaller is very popular, the number of unwanted/spam calls is a big problem. The government has special regulations concerning spam calls (Rukmini, 2014, December 13) and this can be one factor to the popularity of the truecaller application.

Truecaller stores over 1 billion daily events in their data warehouse, where this data is later used for analysis. These analysis can be based on user data, application performance, A/B testing etc. and are used for further development of the application.

2.2 Big Data

“There were 5 Exabyte’s of information created between the dawn of civilization through 2003, but that much information is now created every two days.” - Eric Schmidt, Software Engineer and Executive Chairman of Google (Walker, 2015)

2.2.1 What is Big Data?

The definition of Big Data is often discussed and is a popular and recurring subject within the IT world. Some choose to define “What” Big Data is while others choose to look at what it “Does” (Gandomi & Haider, 2015). The definition is therefore not crystal clear and use of “the three V: s (Volume, Variety and Velocity)” has emerged and is being used to describe Big Data. Volume has to do with the size and is often the first thing that comes in mind when talking about Big Data see fig. 2. When talking about sizes in Big Data, sizes over multiple Petabytes often comes to mind (Walker, 2015) Keep in mind that these sizes increase all the time. It increases even faster than Moore’s Law that means it would double in size every two years. The increase of data actually doubles in size about every 18 months (National Institute of Standards and Technology, 2015). Variety concerning the varying types of data, which shows that the majority (over 90%) of the data is unstructured due to the different types of data that is uploaded all the time through different channels such as, photos, videos, tweets etc. Structured data is stored in relational databases in tables; this is not the case for data stored in Hadoop. Velocity refers to the speed of which the data is produced, stored and analysed see fig. 1. Over the latest years where smart phones have become widely used, the rate of data velocity has exploded and the needs for real-time analytics are in high-demand. Using data from end users from a smart phones vendors can analyse data collected from the hand held devices
such as geospatial location, phone numbers, demographics etc. (Gandomi & Haider, 2015). These are the three “main” V: s but over time additional V: s such as Variability, Value, Validity, Veracity etc. have made it in to discussion in defining Big Data and data analytics (National Institute of Standards and Technology, 2015).

Fig 1. Exponential growth of data (Fishman, 2014, July 14).

Table 2. What is big data? (Cervone, 2015).
Here is an example of one of many definitions of Big Data:
“Big Data consists of extensive datasets - primarily in the characteristics of volume, variety, velocity,
and/or variability - that require a scalable architecture for efficient storage, manipulation, and analysis.” (National Institute of Standards and Technology, 2015)

So what “Does” Big Data do, when is it used and by who? We take a look at it in next paragraph.

2.2.2 Data Analytics
Big Data is useless if it is just stored and never used. The main reason for storing all the data is to analyse it and somehow make use of it. With the help of data analysis, companies can make important decisions for their processes (Walker, 2015). The ways of analysing data is not so different from the past, using statistical models and formulas. The difference is mostly about the size of the data sets and the speed of analysing larger amounts of data. Also, with the help of tools such as Hadoop, companies can focus more on causation because of the availability of data that can be provided. The use of causation helps us to determine why something is happening which leads to making better (National Institute of Standards and Technology, 2015). According to Lambrinis and Jagadish (2012)(Walker, 2015), the process for extracting insights of data is divided into five stages that are part of the two sub processes, data-management and analytics see fig.3. The first three stages are within the more backend process data-management and covers acquiring and storing the data, cleaning and preparing it and then presenting/delivering it to the analysis process where the data is actually analysed, interpreted and visualized (Gandomi & Haider, 2015).

Fig 2. The Big data process and sub processes (Gandomi & Haider, 2015).

2.2.3 Data Warehouse
A data warehouse is a data set/database data collected from one or multiple data sources used to analyse historical data to help organisations with decision-making. In 1990 Bill Inmon introduced the term data warehouse, and in 1993 he stated that data warehouse is defined as:

- Subject-oriented: Meaning that the data focuses on specific subjects regarding the business e.g. customers, products etc.
- Integrated: Because the data often comes from multiple sources the data has to be re-structured to hold a consistent pattern of the data warehouse.
- Time variant: The data is fetched in different times and dates it has to be shown so that it can make sense for different time intervals and time related matters.
- Non-volatile: The data grows incrementally and does not replace the old data but grows steadily over time (Connolly & Begg, 2005).

Data warehouses are often connected to OLAP (Online Analytical Processing Tools) tools for analysis, mining and machine learning. The process when a data warehouse retrieves new data from its data sources is called ETL (extract transform load). Data warehouses are often modelled using a star-schema, which differs from a normal relational database model (Jukić, Sharma, Nestorov & Jukić, 2015). A star-schema is a multi dimensional model see fig.4 using fact tables that represents a business object e.g. a purchase. The fact tables are connected to multiple dimension tables with specific grains that together make up the facts in the fact tables e.g. a product dimension, time dimension etc. Together they can show what product was purchased at what time and date etc. Data warehouses can be modelled with different schemas too e.g. snowflake-schema, OLAP cube etc. (Connolly & Begg, 2005).

![Star schema diagram](Connolly & Begg, 2005).
2.3 Hadoop

2.3.1 What is Hadoop

Hadoop was created by Doug Cutting between 2003 and 2006. Cutting was working on an open source search engine called Nutch where he was inspired by Google’s technologies GFS (Google File System) and MapReduce that together made up a platform for processing a data efficiently on a large scale (White, 2010). From those technologies Cutting started developing Hadoop that quickly became a prioritized project within Apache open source foundation. Yahoo who soon hired Cutting strongly supported the project with help and resources. The name Hadoop comes from Cuttings son’s yellow toy elephant and is also why the logo for Hadoop looks the way it does. By 2008, Hadoop was being used by several large companies like Amazon, Facebook etc. (Turkington, 2013).

Hadoop is a Java based framework for distributed data computation and storing. It’s linear scalability and cheap infrastructure makes it easy to maintain and upgrade, hence makes it the main actor for Big Data processing. Hadoop is made up of two main technologies, HDFS and MapReduce, which are completely different but complement each other. By using Hadoop, large sets of data can be computed in parallel on all the nodes of the cluster. This way all computing power and resources can be utilized efficiently (White, 2010).

2.3.2 Truecaller and Hadoop

Truecaller stores all its data in Hadoop. Hadoop processes the data by MapReduce jobs running in parallel over a cluster of distributed nodes using HDFS (Hadoop Distributed File System). MapReduce is a programming model that splits the data processes in a query where the Map stage sorts and pairs the data in key-value format and then the data is summarized and presented in the Reduce phase see fig. 5.

Fig 4. Simplified MapReduce dataflow (White, 2010).

The main advantage of using Hadoop instead of a relational database for storing and analysing Big Data is because of cost and performance. Hadoop is relatively cheap because the nodes are normal commodity computers that interact in HDFS. It is also linearly scalable which means that when it is needed to add more capacity or space, adding new nodes to the cluster can easily solve the issue (White, 2010). When using a RDMS the data has to be stored on disk and in memory, which can be problematic when it comes to large amounts of data e.g. Petabytes. Therefore, spreading the workload on multiple nodes helps solve this issue (Chen et al., 2014). However, there are some other differences where Hadoop loses against the RDBMS see table. 1. The types of queries that are mainly supposed to be executed in Hadoop are MapReduce batch jobs. Hadoop was not meant to be used
for interactive on-the-fly ad-hoc queries or data discovery. RDBMS are better for ad-hoc queries and provide a much more interactive user interface than Hadoop was intended to do.

<table>
<thead>
<tr>
<th>Data size</th>
<th>Traditional RDBMS</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Interactive and batch</td>
<td>Batch</td>
</tr>
<tr>
<td>Updates</td>
<td>Read and write many times</td>
<td>Write once, read many times</td>
</tr>
<tr>
<td>Transactions</td>
<td>ACID</td>
<td>None</td>
</tr>
<tr>
<td>Structure</td>
<td>Schema-on-write</td>
<td>Schema-on-read</td>
</tr>
<tr>
<td>Integrity</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Scaling</td>
<td>Nonlinear</td>
<td>Linear</td>
</tr>
</tbody>
</table>

Table 3. RDBMS compared to MapReduce (White, 2010).

In recent years many new SQL on Hadoop engines that lets the user have an SQL interface and language to structure and query the data while using MapReduce, have been developed to support and provide ad-hoc querying. Hive is now more or less an industry standard tool for SQL on Hadoop (White, 2010)(Chen et al., 2014). Hive is used as the main data warehouse tool at truecaller.

2.3.3 MapReduce

MapReduce is a programming model used for processing big data distributed and in parallel across the Hadoop cluster. MapReduce is one of the two main parts of Hadoop and can be written in many different programming languages. As noted earlier, the MapReduce model consists of two different phases the;

a. Map, which takes the input data and,
b. Reduce, which outputs the final result.

In Map stage the input data is divided into splits, which have one map task for each split. The map function then runs for each record in the split, and YARN, which is Hadoop’s resource manager schedules the tasks and divides them among the nodes in the cluster. If a job fails YARN sees to it that it is re-run on another machine. Having the right size of splits helps the performance of the MapReduce job, because when the size of the splits are too big the job will be divided into fewer machines and will take longer to process, while if there are many nodes and the splits are small the job can be divided among many machines which can process them in parallel (Maitrey & Jha, 2015). This is something the user has to setup right for the used environment. Normally, a split will be 128 MB, like the size of a HDFS block, which is the default size. A block and a split is not the same thing. A block is a hard division of the data, which is pre-set. Because a record of data does not always fit in one block or can overlap to another block, the MapReduce job uses splits keep track of the different records to process. The number of splits for a MapReduce job can therefore not be predefined but is entirely dependent of the size of the data to be processed in the job (White, 2010).
For best results, Hadoop tries to run the map phase on the same node as where the input data is stored on HDFS. This way, network bandwidth is not wasted. If that node is busy running other map tasks Hadoop chooses another node in the cluster with the replicated data. The name for that process is called data locality optimization.

As mentioned, for every input split there will be one mapper and the mapper outputs a key-value pair as a result. These key-value outputs are written to the local disk on the node and not on HDFS. This is because the result of the map task is only for temporary use, and once the final result of the reduce task is completed, that data will be deleted from the local disk. The reduce phase does not have the data locality optimization because a reducer receives data from multiple mappers which have data from different nodes. After the map phase the key-value pair for every record is grouped by the key with a list with the values before filtered to the right output in the reduce phase(Turkington, 2013). This is the procedure for a MapReduce job with only one reducer see fig. 6. If dealing with a large set of data, it is advisable to use more than one reducer to avoid performance issues and bottlenecks.

The phase between map and reduce is called the shuffle phase. The shuffle phase is very important because it is where the key-value pairs are sorted and merged together, creating the key with the list of values that are used in the reduce phase before the final output (White, 2010). It comes into focus when the jobs are using more than one reducer see fig. 7.

Fig 5. MapReduce flow with one reducer (White, 2010).
When using more than one reducer, the mappers partitions the data, one partition for every reducer. This way, in the shuffle phase when data is sorted and merged the right key-value output goes to the right reducer. Without the partitioning the shuffle phase would not know which reducer the data should be sent to, which would result in having same keys spread out on different reducers which would output a result for each reducer. This would result in an output that will be processed ineffectively, and would quite likely be wrong (Maitrey & Jha, 2015).

![MapReduce data flow with multiple reducers](White, 2010).

### 2.3.4 HDFS

The Hadoop distributed file system or HDFS is the second and other important part of Hadoop, together with MapReduce. HDFS is the file system used on a Hadoop cluster and is used for storing large sets of data on a cluster of distributed nodes. One of the key features of HDFS is that it is run on commodity hardware nodes, which means that is normal priced hardware that can be bought in multiple vendors, and not specially designed for Hadoop. HDFS is designed to keep running if one node fails because that is something that happens frequently and must be accounted for.

The data processing is stream processed which means it is write-once, read-many-times. This concept is meant that the data sets used are written once and then used for processing over time and there for the importance of reading the data sets is bigger than low latency for the output results. The high throughput of the data comes with the cost of latency. The limit for the number of files stored in HDFS is dependant of the memory size of the master node(s), also known as the name node. The name node stores the Meta data of the file system in its memory (location, type etc.) and is used to locate the files on the different worker nodes (White, 2010).
The file system works by using namenodes and datanodes (master/slave) see fig. 8. Initially there used to be one namenode in HDFS, but in later versions of Hadoop, the use of multiple namenodes have been introduced for stability and off-load purposes. Today normally, there are primary and secondary namenode (which are in standby mode). The namenode manages the file system tree and metadata and keeps track of all the files and directories on the HDFS, which are stored on the rest of the nodes, which are the datanodes. The limitation for storing files in HDFS is the memory on the namenode, which holds the reference to all the files in the system. As long as the memory on the namenode is not full, nodes and disk space can be added to the cluster, which makes it scalable. The datanodes are the workers in the HDFS system and all the data on HDFS is stored in blocks on the datanodes. As soon as a change occurs to data on a datanode that is found in the namespace, it is recorded on the namenode (Maitrey & Jha, 2015). The use of storing data in blocks has the advantages that files larger than the disks on the nodes does not become a problem, since it can be distributed among multiple blocks in HDFS. The blocks are replicated and distributed on the cluster for backup and failure concerns. The default replication factor is 3, which means that a block will be replicated and spread three times on the cluster. This makes HDFS very reliable when a node fails or goes down (Apache, 2016) (Srinivasa & Mupalla, 2015).

![HDFS Architecture](Fig 7. HDFS Architecture (Srinivasa & Mupalla, 2015))

2.3.5 YARN
Apache YARN or (Yet Another Resource Negotiator) is Hadoop’s resource management system and was included with Hadoop 2.0. YARN is used for both resource management across the cluster and also for node management, managing the resources for every single node by using containers. The
Containers are used for running applications on top of YARN and have their own allocated resources such as CPU and memory. Applications can be built on YARN using different APIs, and YARN is often not something that the end user has to be concerned about. Applications that use YARN as an intermediate layer between them and the storage layer are called YARN applications see fig. 9. This way YARN makes it easier to run applications on the cluster and sees to it that the resources are allocated right (White, 2010).

Fig 8. Applications running on YARN (White, 2010)

YARN consists of:

a. The resource manager
b. Node manager, and
c. The application master.

The resource manager looks after the clusters resources while the node manager handles the nodes and containers. The application master (one per application) negotiates resources from the resource manager so and together with the node managers it helps the application run and compute tasks.

The resource manager itself consists of:

a. The scheduler, and
b. The ApplicationManager.

The scheduler is the component that schedules and allocates all the resources needed for running applications based on what resources are needed and which can be found in the containers. ApplicationManager ensures that the application gets the right containers from the scheduler so that the application can run. While running it is also responsible for restarting when failures occur, and to track and monitor the status of the application see fig. 10 (Apache Hadoop, 2016).
2.3.5.1 Slider
Slider is a YARN application developed to run non-YARN applications on YARN and to simplify this process. With slider, different users, using different instances of the application, can run an application on YARN. Applications can also be stopped, killed and restarted etc. An example of a non-yarn application that can be run with slider is the Hadoop query engine Presto (Apache Slider Incubator, 2016)(White, 2010).

2.3.6 SQL on Hadoop
SQL (Structured Query Language) has been used for analysing data in relational databases for a long time and is an industry wide standard language within Business Intelligence too. In order to help users without software engineering skills to use Hadoop to analyse and query its data, different SQL on top of Hadoop solutions have been created. More and more competing solutions emerge now and then to try to give the users the easiest and fastest solution on the market (Turkington, 2013). According to T.T. Maposa and M. Seth (2015) the future of SQL on Hadoop is very bright because of the simplicity and wide user range of SQL and the power of the HDFS, together forming a very sought after solution (Maposa & Sethi, 2015).

2.3.6.1 Hive
Hive is an open source data warehousing solution for Hadoop and has an SQL like querying language HiveQL. Hive was developed at Facebook in 2007 and is a very broadly used tool that has become an industry wide standard. Hive compiles the HiveQL statements and into MapReduce jobs and returns the results to the user in a SQL like manner. The HiveQL language has many of the standard SQL
functions and users can add customized functions (UDFs) to add more functionality. Primitives, strings, arrays, maps and structs are many of the data types supported in the tables, which is one of many reasons for Hive’s popularity. Hive can be run both interactively like a normal dbms e.g. MySQL and through batch jobs in the background or after a schedule. The standard interfaces are command line and a web GUI called hue.

The data model of hive is organized in tables with a corresponding directory in HDFS, partitions that are divided into sub directories in the table directory, and buckets, which are the next level of partitioning per partition for faster processing of data.

The architecture of Hive consists of see fig. 11:

a. The client for user interaction
b. The HiveServer2 (Thrift) server that interprets the different external languages against Hive.
c. The Metastore that is the system catalogue that stores the Meta data about all the databases, tables and partitions created by in Hive.
d. The Driver handles the HiveQL statement during compilation and execution and every step between to and from the client goes through the Driver.
e. The Compiler that translates the HiveQL statement from the Driver into a plan that simply speaking consists of the MapReduce jobs that are submitted to the (Hadoop) execution engine. (Thusoo et al., 2009)(White, 2010)(Turkington, 2013).
2.3.6.2 Presto

Presto is an SQL engine that also was developed by Facebook and its main purpose is low-latency ad-hoc querying. Presto uses ANSI SQL as syntax. It can run on Hadoop but also on many other database engines. When connected to Hadoop, Presto uses the Hive Metastore to find the data on HDFS. The main difference between Presto and Hive is that Presto does not process the queries using MapReduce but instead allocates all the tasks to the workers (nodes) memory, and schedules new tasks to the workers once a task is finished (Teradata, 2016). This makes Presto more suitable for interactive querying, especially when the workers run out of memory because of the size of the data processed in a query. The architecture is built of the client that sends the query to a coordinator that parses the SQL statement, plans and schedules the execution. The execution is then pipelined and distributed to the workers located closest the data. The execution runs multiple stages simultaneously and data is streamed between all the stages only using memory for low-latency output (Presto, 2016).

Fig 10. Hive Architecture (Thusoo et al., 2009).
Fig 11. Presto Architecture (Teradata, 2016).

---

Sahir Ahmed  
D0032N – Bachelor Thesis  
Evaluating Presto as an SQL on Hadoop solution – *a case at truecaller*
3. Literature review

3.1 Purpose of literature review
There have been many SQL on Hadoop benchmarks carried out, to try to figure out which one of the many projects to go for. Many of these benchmarks are done by specific vendors trying to market their own product. For that reason they are not always trustworthy when looking to implement a solution at an organisation. The varying results often depend on the tested infrastructure, architecture and optimizations made for the specific solutions. Another factor is also when in time the tests were conducted. The open source projects are in constant development to improve and become market leaders, and new versions of the solution will therefore be more optimized than the previous ones. This will be clearer after reading the results from the different benchmarking attempts that will be presented. Most of the tested engines in the benchmarks below will not be included in the benchmarking at truecaller but will show the diversity of results. Therefore I think it is important to show the differences in the results and how these results cannot be the only source for choosing and trusting a solution by only reading a benchmark. One thing that is common in the benchmarks in this chapter is that that they all beat Hive in performance because they don’t use MapReduce for data processing, but mostly process the data in memory. I have only chosen Presto from the many engines that are mentioned in this section because it met all the requirements that truecaller had.

3.2 Benchmarks
3.2.1 Fast-data-hackathon
The allegro.tech blog reports results from the fast-data-hackathon where a group of users compared various SQL on Hadoop solutions on a 4 node test cluster in June 2015. They used 11 different queries in their tests (Allegro Tech, 2015, June 25). The tested solutions were:
- Hive on Tez
- Presto
- Impala
- Drill
- Spark
In the results above Impala and drill were the clear winners when it comes to execution time but according to the group conducting the tests they both lacked some functionality that could be found in other solutions such as Hive and Presto.

3.2.2 Renmin University

In 2014 at the Renmin University in Beijing, China a group of researchers tested five different SQL on Hadoop solutions to compare them and because of Hive being the standard tool and not being efficient enough for interactive querying. They used 11 different queries with different complexity on 25, 50 and 100 node clusters. The solutions tested were:

- Impala
- Presto
- Hive (to show difference)
- Stinger
- Shark
The results show not only the different response times, but also that some queries were not able to run on all solutions except Hive. According to the tests the researchers concluded that none of the solutions were mature enough to be used on a daily basis (Chen et al., 2014).

3.2.3 Commercial benchmarks

3.2.3.1 Pivotal (HAWQ)

Pivotal that developed the SQL on Hadoop solution HAWQ benchmarked their product against Cloudera’s Impala, Presto and Stinger on a 8 node cluster in June 2015.
The results in the graph shows how HAWQ beats Impala in the majority of the queries run in the test cases (Soliman et al., 2014) (Pivotal, 2014, June 25). It is worth highlighting that Impala has been the clear winner when it comes to speed in all the graphs above.

3.2.3.2 Cloudera (Impala)
In February 2016, Cloudera released their latest Impala benchmark comparing Impala to Hive on Tez and Spark on a 21 node cluster.
The graph above is one of many graphs on Cloudera’s blog showing how Impala beats its competitors in all ways (Cloudera, 2016, February 11). It is worth noting that this is a normal scenario coming from the vendor or developers trying to prove the efficiency of their products.

3.3 Hive not meant for low-latency querying

Because hive is based on HDFS and Hadoop, it is meant for batch jobs focusing on high throughput and scalability and can be used for both analysis and ETL. Because of Hive’s high fault tolerance and wide usability, it has become an industry-wide standard as a data warehouse tool for Hadoop (Turkington, 2013). It also therefore preferred for being reliable and stable and other low-latency solutions are used on the side for ad-hoc querying (Cloudera, 2016).
4. Research methodology

This thesis will be the start of a process to explore and test SQL on Hadoop query engines. The scope of this thesis will be the evaluation of the query engine Presto. Presto will be evaluated against hive and the main focus will be on the latency time of the use case queries, Secondly it will also evaluate the type of use cases it is suitable for and identifying and comparing the pros and cons against the baseline solution hive.

The methodology used in this thesis will be design-science and the design-science research process interpreted by Peffers et al. (2007) as the steps involved are suited for identifying, implementing and evaluating my chosen solution. The main focus using this methodology will be on the evaluation of the solution rather than building an artefact, focus will be on deploying and configuring Presto that is an already developed open source query engine. Defining the right use cases will also be a big part of the design.

The steps in design-science research process are:

1. Identify problem & Motivate.
2. Define Objectives for solution.
3. Design & Development.
4. Demonstration.
5. Evaluation.
6. Communication.

![Design Science Research Methodology](image)

Fig. 16 Design Science Research Methodology Peffers, Tuunanen, Rothenberger & Chatterjee, 2007).

4.1 Identify problems & Motivate

The first step in the design-science research process is to identify and motivate the problems. At truecaller the main problem is the latency and execution time of hive when running ad-hoc queries.

Problems:

1. Hive/MapReduce is too slow for data discovery/ad-hoc queries.
As noted earlier, the problem with the current setup at truecaller is that the data warehouse is accessed through Hive using MapReduce on Hadoop. Running complex queries takes often unnecessary long time and this is both bad for the analytics team and for the stakeholders that request ad-hoc analysis on a regular basis. MapReduce is better for batch jobs that can be scheduled for standard analysis rather than ad-hoc querying and data discovery. Hive and MapReduce takes time because of the processing is done by writing the output on disk.

4.2 Define Objectives for solution

The objectives for the solution will in this case be the requirements for identifying the new solution.

Requirements:

| 1. Open source               |
| 2. SQL interface            |
| 3. Support storage file type ORC |
| 4. Support same data types as Hive e.g. structs, array, Map |
| 5. UDF / UDA (User Defined Functions / Aggregations) |
| 6. Standard analytical functions |
| 7. Window functions         |

Table 4. Requirements

Before trying to identify the different solutions, discussions will be held with the end users at the analytics team to gather requirements regarding the solution. There are many open source solutions that can be tested, and with the right requirements, the list can be narrowed down.

Here the use cases will be defined based on everyday ad-hoc analysis carried out by the analytics team. The use cases will incrementally become more complex to compare the time and resources used by the solutions with each other. An initial baseline where the use cases are run in Hive would be the second step so that the solutions can be compared with the current situation. All the use cases will be tested in solutions supporting SQL-syntax and the queries will be the same or vary in native syntax. (Eg. Hive qsl does not support all standard SQL functions and SQL can also vary depending on the engine/dialect.)

4.3 Design & Development

In this step designing and developing and artefact is the common practice in design-science. In this case, researching and Identifying the solution will be the pre-step before deploying the solution on first a test, and after on the production environment (Hadoop cluster).

Presto was the chosen solution because it met all the requirements that was mentioned above and was easy enough to deploy using the YARN application version.

The first step will be to test the solution on a smaller test environment before implementing it to the production environment. For Presto, the test environment will be a docker container, which is a type
of virtual environment. The container will have a one node Hadoop cluster installed and ready to test on. The test setup was installing the solution, importing a dataset and test running some random queries.

4.4 Demonstration
This step will be for preparing the evaluation of the solution. In this case defining the use cases for the evaluation is the main goal of this step.

The implementation of the solution will not be heavily documented, even though a big part of the time in this project will be spent on deploying and configuring the solution to the current environment at truecaller. The testing in the production environment will be based on the use cases that were pre-defined. This step will show how the solution differs from Hive, and what the pros and cons for the particular queries are. The use cases will be run 4 times. 2 times during work hours when there is normal load on the cluster and 2 times during the evening. The average time in seconds will be used to show the result of the latency time.

4.5 Evaluation
This step is to evaluate the results of the solution.

The results of the use cases will be recorded and visualized using bar charts to show the difference in latency time between the different solutions. The result chapter of the thesis will be mainly focusing on this step of the methodology. Other conclusions from the use cases will also be drawn from the baseline and the new solution e.g. pros and cons and suitable use cases.

4.6 Communication
The thesis itself represents this step. Together with the thesis, the conclusions and recommendations for further testing at truecaller, a presentation at LTUs campus will be held for supervisors and other students.
5. Result

5.1 Cluster details
The Hadoop cluster at truecaller consists of 14 nodes with following components:

<table>
<thead>
<tr>
<th>OS</th>
<th>Linux (debian-kernel) 3.2.0.4-amd64</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU model</td>
<td>Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz</td>
</tr>
<tr>
<td>CPU cores</td>
<td>32</td>
</tr>
<tr>
<td>RAM</td>
<td>128 GB</td>
</tr>
<tr>
<td>HDD</td>
<td>~20 TB</td>
</tr>
</tbody>
</table>

Table 5. Cluster details

5.2 Implementation
Presto was deployed on truecaller’s Hadoop cluster using a version that allows it to run as a YARN application using all 14 nodes. One node shared as both Coordinator and Worker and the rest as Workers. The whole setup and configuration can be found under appendices 7.1 Presto installation.

5.3 Use cases and latency in seconds
1. Select * with restriction and Limitation type 1. Table size 14 Billion rows.
   Hive: ~20 sec
   Presto: ~5 sec
   Conclusion:
   Presto 4x faster.
2. Select * with restriction and Limitation type 2. Table size 14 Billion rows.
   Hive: ~40 sec
   Presto: ~7 sec
   Conclusion:
   Presto ~6x faster.
3. Select * with restriction and Limitation type 3. Table size 14 Billion rows.
   Hive: ~200 sec
   Presto: ~15 sec
   Conclusion:
   Presto ~13x faster.
4. Select count with restriction. Table size 14 Billion rows.
Hive: ~90 sec  
Presto: ~23 sec  
Conclusion:

Presto ~4x faster.

5. Select * with restriction, limitation and Order By. Table size 14 Billion rows.
Hive: ~340 sec  
Presto: Runs out of memory because of Order By, too expensive
Conclusion:

Not suitable use case for presto.

6. Select with 2 aggregate functions. Table size 14 Billion rows.
Hive: ~600 sec  
Presto: ~120 sec
Conclusion:

Presto ~5x faster.

7. Select with aggregates, group by and limitation type 1. Table size 14 Billion rows.
Hive: ~120 sec  
Presto: ~40 sec
Conclusion:

Presto ~3x faster.

8. Select with aggregates, group by and limitation type 2. Table size 14 Billion rows.
Hive: ~120 sec  
Presto: ~50 sec
Conclusion:

Presto ~2.5x faster.

9. Select with multiple aggregates, group by, limitations and Join table 3 times. Table size 14 Billion rows.
Hive: ~6000 sec  
Presto: ~7000 sec
Conclusion:

Hive 1.2x faster.
5.4 Result table

![Query execution time in seconds](image)

Fig 17. Query execution time in seconds
6. Discussion

6.1 Presto at truecaller
Looking at the queries where Presto performed faster than Hive, the average gain of latency time is around 5 times faster. This is a bit lower than some example benchmarks that can be found online. However it is still a big gain when needed to run queries over and over again during work. Presto showed clearly that it is more suited for returning fast insights on a smaller scale and is suited for ad-hoc analysis for someone who is familiar with the data sets and knows what to and where to look for. It is less suited for more expensive queries where the worker needs to store with large amounts of meta data, and risks to run out of memory. In the configuration and testing, allocating 70 GB of maximum query memory was not enough for query 5 which contained an order by operation, and ran out of memory. Presto is also not suited for multiple joins between large tables, where it performed slower than Hive. In truecaller’s case Presto is fine for many of the common day-to-day ad-hoc use cases. At truecaller multiple tools are used within the analytics team and knowing the pros and cons of the tools helps the users to choose which one to go for. For some ETL jobs, Spark is being used and therefor the team will not suffer from adding another tool to their toolbox. Presto has some minor syntax differences in but they are easy to learn and should not be seen as an obstacle. Another finding from this research is that Presto can be configured and tuned on very detailed level. The results presented in the graph are after numerous iterations of configurations and tuning. There is also the standard version, which requires more configurations on an operating system level. The version tested at truecaller was the YARN application version. Therefore there is more room for tuning and testing Presto before choosing the final settings for truecaller’s Hadoop cluster and environment. My recommendation would be after tuning and configuring Presto, to use Presto for some more lightweight queries together with Hive for the bigger jobs (ETL, batch etc.).

6.2 Use cases
The use cases chosen for the evaluation are based on work I did for truecaller on the side of the thesis work. The queries used are actual ad-hoc requests for the analytics team, which I took upon to learn more about the data that was analysed on a daily basis. That way I got a good picture about what types of queries were often run and will be run in a new solution if implemented.

6.3 Limitations
The goal from the beginning was to evaluate 2-3 solutions and compare them to Hive like I did with Presto. Due to some resource limitations and the complexity of the project, I could not proceed with testing more solutions. This had mostly to do with me not risking compromising the daily work by trying to implement new solutions on the production cluster without senior supervision.

6.4 Commercial alternatives
The reason to test open source solutions was as earlier mentioned to keep the costs down. There are otherwise commercial alternatives that perform very well and run stable with Hadoop. An example is Exasol an in-memory analytics database. I have tested and worked on Exasol in other projects and would recommend it as an low-latency solution for companies that can consider paying for such a solution.
6.5 Further research

Tuning and configuring Presto with professional help would be a first step to see how much faster Presto could become. It would also be interesting to see if there are any gains regarding heavier queries and jobs compared to the current configurations at truecaller. A next step would be to see if gains in execution time would be possible by trying to tune the parameters of the ORC file. This was another intention to test during this research but could not be done due to time limitations.

Apache Drill and Apache Tajo are the other two solutions that were meant to be tested and evaluated in this thesis work. Tajo was actually implemented but did not support some of the required file types and data types. I have been in close contact with the developers of Tajo and it should meet all of truecaller’s requirements in the next release, which was initially planned for some time in April 2016, but was delayed. After the release and if there is time and resources, looking into both of these solutions could be of interest for truecaller.

If there are additional resources and time, I would recommend testing Impala by changing file formats from ORC to Parquet and do run the same type of tests. Because of the results from most of the benchmarks read during this research, Impala is proven to be a strong candidate for a lightweight SQL on Hadoop tool. The only problem is the difference in the file types (ORC vs. Parquet) that truecaller has chosen to store.
7. Conclusion

There are many SQL on Hadoop solutions on the market today. Presto is one of the known names of open source SQL on Hadoop solutions and was chosen as the tool to be tested after meeting all the requirements. Since only Presto was tested, due to limitations, it cannot be said that it is the only and/or perfect solution for truecaller’s case. However, it has answered the research questions of this thesis by proving to be on average 5 times faster than Hive. Presto was proven work best for lighter ad-hoc analysis running on smaller data sets or sample data taken form larger sets to prove a point. Presto did not work well with larger data sets and heavier queries containing multiple joins/self joins. Presto can be tuned and optimized to fit the particular environment to perform at its best. The use-cases and all testing were done at truecaller in Stockholm and the results are all depending on the configuration and hardware of the truecaller’s Hadoop cluster. Presto is now in use by the analytics team at truecaller for specific ad-hoc use cases and will not permanently replace Hive or any other tool at the company. User’s that know the data will be able to make the right decision for when to and not to use Presto. Using Presto for the right use-cases will save truecaller’s analytics team time and will add efficiency to their work, which is was the main purpose for them to want to test another solution. By deploying Presto at truecaller, this goal has been reached for now.
8. References

8.1 Books and articles

8.2 Websites and blog posts

9. Appendices

9.1 Appendix A – Presto installation

Guide to install presto on hadoop cluster.

On docker container with cloudera one node cluster for test:

Prerequisites:

- HDFS
- YARN
- ZooKeeper
- Java
- Maven

**Step 1.** Install Slider to run presto application on yarn:

1. Create yarn user on containers hdfs with:

   ```
   hdfs dfs -mkdir -p /user/yarn hdfs dfs -chown yarn:yarn /user/yarn
   ```

2. Download slider 0.80.0 for yarn
   (http://apache.mirrors.spacedump.net/incubator/slider/0.80.0-incubating/slider-assembly-0.80.0-incubating-all.tar.gz)

3. Untar file with command:

   ```
   tar -xvf slider-0.80.0-incubating-all.tar.gz
   ```

4. Open new directory and go to slider-assembly/target directory and copy 0.80.0-incubating-all.zip to a new folder where you want to install slider.

5. Unzip file in folder, the slider folder will now be created.

6. Go to slider folder /conf and open file slider-env.sh set env variables:

   ```
   export JAVA_HOME=/usr/lib/jvm/java-8-oracle/ export HADOOP_CONF_DIR=/etc/hadoop/conf
   ```

7. Open file slider-client.xml change set values:

   ```
   <property>  <name>hadoop.registry.zk.quorum</name>  <value>localhost:2181</value>  </property>
   <property>  <name>yarn.resourcemanager.address</name>  <value>localhost:8032</value>  </property>
   <property>  <name>yarn.resourcemanager.scheduler.address</name>  <value>localhost:8030</value>  </property>
   <property>  <name>fs.defaultFS</name>  <value>hdfs://localhost:8020</value>  </property>
   ```

8. Test slider with command:

   ```
   ${slider-install-dir}/slider-0.80.0-incubating/bin/slider version
   ```
If you get output "Compiled against Hadoop 2.6.0", slider is installed right.

**Step 2. Install presto:**

1. Download presto-yarn package from github: (git clone https://github.com/prestodb/presto-yarn.git)
2. Build package with maven:
   
   mvn clean package
3. Go to folder created and folder presto-yarn-package/target and copy presto-yarn-package-1.0.0-SNAPSHOT-0.130.zip into slider folder.
4. Go back to presto-yarn-package folder and to /src/main/resources and copy files appConfig.json and resources-singlenode.json into slider folder.
5. In slider older rename resources-singlenode.json to resources.json and open it and edit settings:

```json
{
    "schema": "http://example.org/specification/v2.0.0",
    "metadata": {
    },
    "global": {
    "yarn.vcores": "1"
    },
    "components": {
    "slider-appmaster": {
    },
    "COORDINATOR": {
    "yarn.role.priority": "1",
    "yarn.component.instances": "1",
    "yarn.memory": "1500"
    }
    }
}
```

6. Open appConfig.json and edit settings:

```json
{
    "schema": "http://example.org/specification/v2.0.0",
    "metadata": {
    },
    "global": {
    "site.global.app_user": "yarn",
    "site.global.user_group": "hadoop",
    "site.global.data_dir": "/var/lib/presto/data",
    "site.global.config_dir": "/var/lib/presto/etc",
    "site.global.app_name": "presto-server-0.130",
    "site.global.app_pkg_plugin": "/$AGENT_WORK_ROOT/app/definition/package/plugins/",
    "site.global.singlenode": "true",
    "site.global.coordinator_host": "/$COORDINATOR_HOST",
    "site.global.presto_query_max_memory": "50GB",
    "site.global.presto_query_max_memory_per_node": "512MB",
    "site.global.presto_server_port": "8080",
```
"site.global.catalog": {"hive": ["connector.name=hive-cdh5","hive.metastore .uri=thrift://${NN_HOST}:9083"], "tpch": ["connector.name=tpch"]}, "site.global.jvm_args": ["-server", '-Xmx1024M', '-XX:+UseG1GC', '-XX:G1Heap pRegionSize=32M', '-XX:+UseGCOverheadLimit', '-XX:+ExplicitGCInvokesConcurrent', '-XX:+HeapDumpOnOutOfMemoryError', '-XX:OnOutOfMemoryError=kill -9 %p"],

"application.def": ".slider/package/PRESTO/presto-yarn-package-1.1-SNAPSHOT-0.130.zip",
"java_home": "/usr/lib/jvm/java-8-oracle/"},
"components": {
  "slider-appmaster": {
    "jvm.heapsize": "128M"
  }
}
}

Test different settings for environment node memory 1024MB was too much for us and we changed to:
site.global.presto_query_max_memory_per_node": "512MB"
7. In slider folder change user to yarn (su yarn) and run command to install presto:

bin/slider package --install --name PRESTO --package presto-yarn-package-*.zip

8. Run presto with command:

bin/slider create prestol --template appConfig.json --resources resources.json

If exited with status 0 presto app should now be running.

9. Go to YARN resource manager web UI to check if presto is running (http://science2.truecaller.net:8088):

(See first row application presto1 running).
10. Create new folder and download presto client jar file

(https://repository.sonatype.org/service/local/artifact/maven/content?r=central-proxy&g=com.facebook.presto&a=presto-cli&v=RELEASE)
11. Rename jar file to to only presto (mv *.jar presto). Run command to make it executable:

chmod +x presto

12. To test presto go to hive client and create a table with data. Now go back to folder with presto as yarn user (su yarn) and type command to run presto:

./presto --server localhost:8080 --catalog system --schema default
(or which schema wanted to use)

type show tables; if table is showing, tryout a query.

On hdp3 cluster:
Step 1. Install Slider to run presto application on yarn:

1. Create yarn user on containers hdfs with:

   ```bash
   hdfs dfs -mkdir -p /user/yarn hdfs dfs -chown yarn:yarn /user/yarn
   ```

2. Download slider 0.80.0 for yarn

   (http://apache.mirrors.spacedump.net/incubator/slider/0.80.0-incubating/slider-assembly-0.80.0-incubating-all.tar.gz)

3. Untar file with command:

   ```bash
   tar -xvf slider-0.80.0-incubating-all.tar.gz
   ```

4. Open new directory and go to slider-assembly/target directory and copy 0.80.0-incubating-all.zip to a new folder where you want to install slider.

5. Go to slider folder /conf and open file slider-env.sh set env variables:

   ```bash
   export JAVA_HOME=/usr/lib/jvm/jdk-8-oracle-x64/ export HADOOP_CONF_DIR=/etc/hadoop/conf
   ```

6. Open file slider-client.xml change set values:

   ```xml
   <property>
     <name>hadoop.registry.zk.quorum</name>
     <value>----------------</value>
   </property>
   <property>
     <name>yarn.resourcemanager.address</name>
     <value>-----</value>
   </property>
   <property>
     <name>yarn.resourcemanager.scheduler.address</name>
     <value></value>
   </property>
   <property>
     <name>fs.defaultFS</name>
     <value>hdfs---8020</value>
   </property>
   ```

7. Test slider with command:

   ```bash
   ${slider-install-dir}/slider-0.80.0-incubating/bin/slider version
   ```

   If you get output "Compiled against Hadoop 2.6.0", slider is installed right.

Step 2. Install presto:

1. Download presto-yarn package from github: (git clone https://github.com/prestodb/presto-yarn.git)

2. Build package with maven:

   ```bash
   mvn clean package
   ```

3. Go to folder created and folder presto-yarn-package/target and copy presto-yarn-package-1.0.0-SNAPSHOT-0.130.zip into slider folder.

4. Go back to presto-yarn-package folder and to /src/main/resources and copy files appConfig.json and resources-singlenode.json into slider folder.

5. In slider older rename resources-singlenode.json to resources.json and open it and edit settings:

   ```json
   {
   "schema": "http://example.org/specification/v2.0.0",
   "metadata": { }
   ,
   ```
"global": {
  "yarn.vcores": "8"
},
"components": {
  "slider-appmaster": {
  },
  "COORDINATOR": {
    "yarn.role.priority": "1",
    "yarn.component.instances": "1",
    "yarn.component.placement.policy": "1",
    "yarn.memory": "4000"
  },
  "WORKER": {
    "yarn.role.priority": "2",
    "yarn.component.instances": "13",
    "yarn.component.placement.policy": "1",
    "yarn.memory": "8000"
  }
}
6. Open appConfig.json and edit settings:
{
  "schema": "http://example.org/specification/v2.0.0",
  "metadata": {
  },
  "global": {
    "site.global.app_user": "yarn",
    "site.global.user_group": "hadoop",
    "site.global.data_dir": "/var/lib/presto/data",
    "site.global.config_dir": "/var/lib/presto/etc",
    "site.global.app_name": "presto-server-0.142",
    "site.global.app_pkg_plugin": "{AGENT_WORK_ROOT}/app/definition/package/plugins/",
    "site.global.singlenode": "true",
    "site.global.coordinator_host": "${COORDINATOR_HOST}"
  },
  "site.global.presto_query_max_memory": "50GB",
  "site.global.presto_query_max_memory_per_node": "4GB",
  "site.global.presto_server_port": "8080",
  "site.global.catalog": "{'hive': ['connector.name=hive-cdh5', 'hive.metastore.uri=thrift://hdp3.truecaller.net:9083'], 'tpch': ['connector.name=tpch']}",
  "site.global.jvm_args": "{'-server', '-Xmx8192M', '-XX:+UseG1GC', '-XX:G1HeapRegionSize=32M', '-XX:+UseGCOverheadLimit', '-XX:+ExplicitGCInvokesConcurrent', '-XX:+HeapDumpOnOutOfMemoryError', '-XX:OnOutOfMemoryError=kill -9 %p']",
  "application.def": ".slider/package/PRESTO/presto-yarn-package-1.1-SNAPSHOT-0.142.zip",
  "java_home": "/usr/lib/jvm/jdk-8-oracle-x64/"
},
"components": {
  "slider-appmaster": {
  },
  "jvm.heapsize": "128M"
7. In slider folder change user to yarn (su yarn) and run command to install presto:

   bin/slider package --install --name PRESTO --package presto-yarn-package-*.zip

8. Run presto with command:

   bin/slider create presto1 --template appConfig.json --resources resources.json

   If exited with status 0 presto app should now be running.

9. Go to YARN resourcemanager web UI to check if presto is running

   (See first row application presto1 running).

10. Create new folder and download presto client jar file

    (https://repo1.maven.org/maven2/com/facebook/presto/presto-cli/0.142/presto-cli-0.142-executable.jar)

11. Rename jar file to to only presto (mv *.jar presto). Run command to make it executable:

    chmod +x presto

12. To test presto go to hive client and create a table with data. Go to the resource manager web interface, find presto1 under running applications and find URL to coordinator node.

13. Now go back to folder with presto as yarn user (su yarn) and type command to run presto:

    ./presto --server h[URL to coordinator]:8080 --catalog hive --schema default (or which schema wanted to use)

    type show tables; if table is showing, tryout a query

9.2 Appendix B – Use case queries

1. SELECT * with restriction AND LIMIT 1:

   SELECT *
   FROM app_events
   WHERE UPPER(user.country_code) = 'IN'
   AND dt = 20160221
   AND h = 02
2. SELECT * with restriction AND LIMIT 2:

```sql
SELECT *
FROM app_events
WHERE dt = 20160221
AND appsearch.type IN ('1', '2')
LIMIT 10000;
```

3. SELECT * with restriction AND LIMIT 3:

```sql
SELECT * FROM app_events
WHERE user.register_id = 2107422
AND appsearch.type = '2'
AND dt between 20160123 AND 20160125
LIMIT 50;
```

4. SELECT COUNT(*) with restriction

```sql
SELECT COUNT(*) FROM app_events
WHERE upper(user.country_code) = 'IN'
AND appsearch.type = '2'
AND dt = 20160123;
```

5. SELECT * with restriction LIMIT AND ORDER BY:

```sql
SELECT *
FROM app_events
WHERE dt = 20160221
```
AND user.country_code IS NOT NULL
AND appsearch.type IN ('1', '2')
ORDER BY UPPER(user.country_code)
LIMIT 10000;

6. SELECT * with restriction LIMIT AND SORT BY:

SELECT *
FROM app_events
WHERE dt = 20160221
AND h = 02
AND user.country_code IS NOT NULL
AND appsearch.type IN ('1', '2')
SORT BY UPPER(user.country_code);

7. Simple SELECT with aggregates:

SELECT
SUM(appsearchv2.attempts[0].latency) sum_latency,
AVG(appsearchv2.attempts[0].latency) avg_latency
FROM data.app_events
WHERE size(appsearchv2.attempts) > 0
AND dt between 20160130 AND 20160202
AND upper(user.country_code) = 'IN'
AND appsearchv2.attempts[0].endpoint not like '%localhost%';

8. SELECT with aggregates AND GROUP BY:

SELECT user.register_id,
appsearch.searchrequest.normalizedphonenumber as incoming_nr,
SUM(if(appsearch.type = 2, 1, 0)) incoming_calls,
SUM(if(appsearch.type = 5, 1, 0)) sms
FROM data.app_events
WHERE dt = 20160121
AND appsearch.searchrequest.searchcountrycode = 'SE'
AND appsearch.searchrequest.normalizedphonenumber IS not NULL
GROUP BY user.register_id,
appsearch.searchrequest.normalizedphonenumber
LIMIT 10000;

9. SELECT with aggregates with group AND order by:

SELECT user.register_id,
appsearch.searchrequest.normalizedphonenumber as incoming_nr,
SUM(if(appsearch.type = 2, 1, 0)) incoming_calls,
SUM(if(appsearch.type = 5, 1, 0)) sms
FROM data.app_events
WHERE dt = 20160121
AND appsearch.searchrequest.searchcountrycode = 'SE'
AND appsearch.searchrequest.normalizedphonenumber IS not NULL
GROUP BY user.register_id,
appsearch.searchrequest.normalizedphonenumber
ORDER BY user.register_id,
incoming_nr
LIMIT 10000;

10. SELECT FROM inner query with inner GROUP BY AND outer order by:
SELECT register_id, incoming_nr, incoming_calls, sms 
FROM ( 
    SELECT user.register_id, appsearch.searchrequest.normalizedphonenumber as incoming_nr, 
            SUM(if(appsearch.type = 2, 1, 0)) incoming_calls, 
            SUM(if(appsearch.type = 5, 1, 0)) sms 
    FROM data.app_events 
    WHERE dt = '20160125' 
    AND appsearch.searchrequest.searchcountrycode = 'SE' 
    AND appsearch.searchrequest.normalizedphonenumber IS not NULL 
    GROUP BY user.register_id, appsearch.searchrequest.normalizedphonenumber 
)x 
WHERE incoming_calls > 0 OR sms > 0 
order by register_id 
LIMIT 10000;

11. JOIN WITH 2 INNER QUERIES WITH AGGREGATES AND ORDER BY:

SELECT
    a.city, 
    COUNT(DISTINCT a.registerid) nr_IOS_profiles 
FROM(
    SELECT
        LOWER(city) city, 
        registerid 
    FROM data.user_profile_data 
    WHERE LOWER(city) <> ''
)
AND dt = 20160221
AND registerid <> 0 ) a
INNER JOIN(
    SELECT
    id
    FROM userprimaryserviceprovider
    WHERE LOWER(os) LIKE 'iphone%'
    AND dt = 20160221) b
ON b.id = a.registerid
GROUP BY a.city
ORDER BY nr_IOS_profiles DESC
LIMIT 10;

12. 3 joins with aggregates AND sort by:

SELECT
    FROM_UNIXTIME(UNIX_TIMESTAMP() - 86400, 'yyyyMMdd') AS dt,
    a.countrycode,
    a.incoming_phonenr,
    COALESCE(c.incoming_calls_24h, 0L) incoming_calls_24h,
    COALESCE(c.sms_24h, 0L) sms_24h,
    COALESCE(b.incoming_calls_3d, 0L) incoming_calls_3d,
    COALESCE(b.sms_3d, 0L) sms_3d,
    a.incoming_calls_7d,
    a.sms_7d
FROM
(SELECT
    countrycode,
    incoming_phonenr,
    incoming_calls_7d,
sms_7d
FROM (SELECT
    appsearch.searchrequest.searchcountrycode as countrycode,
    appsearch.searchrequest.normalizedphonenumber AS incoming_phonenr,
    SUM(if(appsearch.type = 2, 1, 0)) incoming_calls_7d,
    SUM(if(appsearch.type = 5, 1, 0)) sms_7d
FROM data.app_events
WHERE dt >= 20160101 AND dt <= 20160107
AND upper(appsearch.searchrequest.searchcountrycode) IN ('SE', 'US')
AND appsearch.searchrequest.normalizedphonenumber IS NOT NULL
GROUP BY
    appsearch.searchrequest.searchcountrycode,
    appsearch.searchrequest.normalizedphonenumber) x
WHERE incoming_calls_7d > 0 OR sms_7d > 0
SORT BY countrycode)

LEFT JOIN
(SELECT
    countrycode, incoming_phonenr, incoming_calls_3d, sms_3d
FROM (SELECT
    appsearch.searchrequest.searchcountrycode as countrycode,
    appsearch.searchrequest.normalizedphonenumber AS incoming_phonenr,
    SUM(if(appsearch.type = 2, 1, 0)) incoming_calls_3d,
    SUM(if(appsearch.type = 5, 1, 0)) sms_3d
FROM data.app_events
WHERE dt >= 20160101 AND dt <= 20160103

WHERE dt >= 20160101 AND dt <= 20160107
AND upper(appsearch.searchrequest.searchcountrycode) IN ('SE', 'US')
AND appsearch.searchrequest.normalizedphonenumber IS NOT NULL
GROUP BY
    appsearch.searchrequest.searchcountrycode,
    appsearch.searchrequest.normalizedphonenumber) x
WHERE incoming_calls_3d > 0 OR sms_3d > 0
SORT BY countrycode) a

(SELECT
    countrycode, incoming_phonenr, incoming_calls_3d, sms_3d
FROM (SELECT
    appsearch.searchrequest.searchcountrycode as countrycode,
    appsearch.searchrequest.normalizedphonenumber AS incoming_phonenr,
    SUM(if(appsearch.type = 2, 1, 0)) incoming_calls_3d,
    SUM(if(appsearch.type = 5, 1, 0)) sms_3d
FROM data.app_events
WHERE dt >= 20160101 AND dt <= 20160103

WHERE dt >= 20160101 AND dt <= 20160107
AND upper(appsearch.searchrequest.searchcountrycode) IN ('SE', 'US')
AND appsearch.searchrequest.normalizedphonenumber IS NOT NULL
GROUP BY
    appsearch.searchrequest.searchcountrycode,
    appsearch.searchrequest.normalizedphonenumber) x
WHERE incoming_calls_3d > 0 OR sms_3d > 0
SORT BY countrycode) a

LEFT JOIN
(SELECT
    countrycode, incoming_phonenr, incoming_calls_3d, sms_3d
FROM (SELECT
    appsearch.searchrequest.searchcountrycode as countrycode,
    appsearch.searchrequest.normalizedphonenumber AS incoming_phonenr,
    SUM(if(appsearch.type = 2, 1, 0)) incoming_calls_3d,
    SUM(if(appsearch.type = 5, 1, 0)) sms_3d
FROM data.app_events
WHERE dt >= 20160101 AND dt <= 20160103

WHERE dt >= 20160101 AND dt <= 20160107
AND upper(appsearch.searchrequest.searchcountrycode) IN ('SE', 'US')
AND appsearch.searchrequest.normalizedphonenumber IS NOT NULL
GROUP BY
    appsearch.searchrequest.searchcountrycode,
    appsearch.searchrequest.normalizedphonenumber) x
WHERE incoming_calls_3d > 0 OR sms_3d > 0
SORT BY countrycode) a
AND upper(appsearch.searchrequest.searchcountrycode) IN ('SE', 'US')
AND appsearch.searchrequest.normalizedphonenum IS NOT NULL
GROUP BY
    appsearch.searchrequest.searchcountrycode,
    appsearch.searchrequest.normalizedphonenum
WHERE incoming_calls_3d > 0 OR sms_3d > 0
SORT BY countrycode) b
ON a.countrycode = b.countrycode
AND a.incoming_phonenr = b.incoming_phonenr
LEFT JOIN
(SELECT
    countrycode,
    incoming_phonenr,
    incoming_calls_24h,
    sms_24h
FROM (SELECT
    appsearch.searchrequest.searchcountrycode as countrycode,
    appsearch.searchrequest.normalizedphonenum AS incoming_phonenr,
    SUM(if(appsearch.type = 2, 1, 0)) incoming_calls_24h,
    SUM(if(appsearch.type = 5, 1, 0)) sms_24h
FROM data.app_events
WHERE dt = 20160101
AND upper(appsearch.searchrequest.searchcountrycode) IN ('SE', 'US')
AND appsearch.searchrequest.normalizedphonenum IS NOT NULL
GROUP BY
    appsearch.searchrequest.searchcountrycode,
    appsearch.searchrequest.normalizedphonenum
WHERE incoming_calls_24h > 0 OR sms_24h > 0
SORT BY countrycode) c
ON b.countrycode = c.countrycode
AND b.incoming_phonenr = c.incoming_phonenr
SORT BY countrycode

13. Create table out of 3 joins FROM large table with multiple aggregates AND GROUP BY.

CREATE TABLE sahir.nr_search_events_30days_v2 AS
SELECT
 a.dt,
 a.country_code,
 COALESCE(b.hits, 0L) AS total_result_hits,
 COALESCE(a.hits, 0L) AS 1_result_hits,
 COALESCE(c.hits, 0L) AS 0_result_hits
FROM
(SELECT
 dt,
 LOWER(user.country_code) country_code,
 COUNT(*) hits
FROM data.app_events
WHERE dt BETWEEN 20160201 AND 20160301
AND SIZE(search.sources) > 0
AND LOWER(search.sources[0]) LIKE '%ugc%'
AND search.result_size = 1
GROUP BY
dt,
LOWER(user.country_code)) a
LEFT JOIN
(SELECT
 dt,

FROM
(SELECT
 dt,
LOWER(user.country_code) country_code,
COUNT(*) hits
FROM data.app_events
WHERE dt BETWEEN 20160201 AND 20160301
AND SIZE(search.sources) > 0
AND search.result_size > 0
GROUP BY
dt,
LOWER(user.country_code)) b
ON a.dt = b.dt
AND LOWER(a.country_code) = LOWER(b.country_code)
LEFT JOIN
(SELECT
dt,
LOWER(user.country_code) country_code,
COUNT(*) hits
FROM data.app_events
WHERE dt BETWEEN 20160201 AND 20160301
AND SIZE(search.sources) > 0
AND search.result_size = 0
GROUP BY
dt,
LOWER(user.country_code)) c
ON b.dt = c.dt
AND LOWER(b.country_code) = LOWER(c.country_code)