

Big Data

Airlines on-time performance

A comparative study of Hadoop solutions

Table of Contents

Software	3
Source Code	3
Acronyms Definition	3
Abstract	3
Introduction.....	3
Experiment Methodology	4
Dataset Description.....	4
Experiment Description.....	6
Results.....	7
High level functional comparison.....	7
Query pre-fetch load time comparison	10
Query Response Time & Complexity Comparison	11
Lessons Learnt & Challenges	18
Spark.....	18
Hive	19
Potential Improvements	19
Hive	19
Spark.....	19
Conclusion	20
Bibliography	20
Appendix.....	21
Appendix A – Spark job run exception	21
Appendix B– HDFS to Hive table load time proofs.....	21
Appendix C– Sparks object building timings proofs (prior to querying).....	24

Software

Hadoop version – 2.5

Hive version – 0.14.0

Spark version – 1.4

Source Code

The Hive and Spark source code are available respectively in Hive_Scripts.txt and SparkProto.scala. It is fully commented.

Acronyms Definition

ANSI	– American National Standards Institute
CLI	– Command Line Interface
CRUD	– Create, Read, Update, Delete
DAG	– Directed Acyclic Graph
DDL	– Data Definition Language
DML	– Data Manipulation Language
EMS	– Enterprise Messaging System
HDFS	– Hadoop File System
JDBC	– Java Database Connection
ODBC	– Object Database Connection
UDF	– User Defined Functions
UI	– User Interface
RDBMS	– Relational Database Management system
SQL	– Structured Query Language

Abstract

The comparison of the Hive and Spark implementations, to answer airport delays/cancellations related questions, showed that both solution can handle the business reporting need. The experiment also demonstrates that although Hive was faster overall by 7% (using synchronous queries), the detail of the response times provided a different insight. First, complex queries (i.e. with joins) took on average 25% faster in Spark compared to Hive, with a standard deviation 2%. Simple queries (i.e. without joins) were more efficient in Hive by 5% on average, with a standard deviation 2%. Furthermore, the pre-fetch load time was 15 times slower in Hive compared to Spark. Therefore, depending on usage (need of real-time or batch), and the complexity of the queries, one of the solution could be viewed as more appropriate than the other. In terms of functionality, it was established that Spark supported a richer functionality set (e.g. it supports machine learning library – Mllib natively). It was also established that Hive is more biased towards the none programmer community, whether Spark requires strong programming and IT architectural skills.

Introduction

The purpose of this project is to implement and compare two big data solutions, namely Hive and Spark, to answer airport delays and cancellations related questions. The main body of this report is organised in four parts. The first part details the experiment, i.e. it describes the data and the process flow for each solution. The second section focuses on the results of the experiment, in terms of functionality and response time for each implementation. The third section reviews the implementations' challenges and the lesson learnt from the experiment execution. The last part proposes several improvements that could be applied to the current Hive and Spark implementations to improve response time and improve the code maintenance.

Experiment Methodology

Dataset Description

The data under analysis consists of a large dataset, stored in 21 compressed csv files of nearly 120 million records in total (c.f. Figure 1 from 1987.csv to 2008.csv). It represents a total of 1.6 gigabytes of space compressed, and 12 gigabytes when uncompressed. It contains the information relating to flights' arrivals and departures for all commercial flights within the USA, from October 1987 to April 2008 [0]. It also comprises a few smaller data sets containing i) airports details (e.g. name, country, latitude, longitude, etc.), ii) carriers' information and iii) the plane data (e.g. engine type, age, etc.). This data is contained respectively in the airports.csv, carriers.csv and plane-data.csv files. The full file list is displayed in Figure 1. Figure 2 shows a functional representation of the primary/foreign key relationships between the different datasets. This representation is necessary for querying purpose only, not for storing. Querying require 'joins' to connect two datasets together. Therefore, there is a need to understand the dataset relationships. However, as Hive is not an RDBMS, these relationships are not constraints at the database level.

Name	Date modified	Type	Size
1987.csv	20/02/2017 13:50	WinZip File	12,356 KB
1988.csv	20/02/2017 13:51	WinZip File	48,339 KB
1989.csv	20/02/2017 13:51	WinZip File	48,050 KB
1990.csv	20/02/2017 13:54	WinZip File	50,822 KB
1991.csv	20/02/2017 12:55	WinZip File	48,709 KB
1992.csv	20/02/2017 12:57	WinZip File	48,869 KB
1993.csv	20/02/2017 13:01	WinZip File	48,938 KB
1994.csv	20/02/2017 12:55	WinZip File	49,926 KB
1995.csv	20/02/2017 12:59	WinZip File	73,127 KB
1996.csv	20/02/2017 13:00	WinZip File	74,110 KB
1997.csv	20/02/2017 13:00	WinZip File	74,908 KB
1998.csv	20/02/2017 12:58	WinZip File	74,887 KB
1999.csv	20/02/2017 13:01	WinZip File	77,588 KB
2000.csv	20/02/2017 13:02	WinZip File	80,604 KB
2001.csv	20/02/2017 12:56	WinZip File	81,523 KB
2002.csv	20/02/2017 13:00	WinZip File	74,129 KB
2003.csv	20/02/2017 12:13	WinZip File	93,093 KB
2004.csv	20/02/2017 12:16	WinZip File	108,228 KB
2005.csv	20/02/2017 12:17	WinZip File	109,815 KB
2006.csv	20/02/2017 12:15	WinZip File	112,324 KB
2007.csv	20/02/2017 11:56	WinZip File	118,408 KB
2008.csv	09/02/2017 20:47	WinZip File	111,088 KB
airports	20/02/2017 11:30	Microsoft Excel Co...	209 KB
carriers	17/01/2017 14:01	Microsoft Excel Co...	43 KB
plane-data	17/01/2017 14:01	Microsoft Excel Co...	419 KB

Figure 1 – Raw data file list

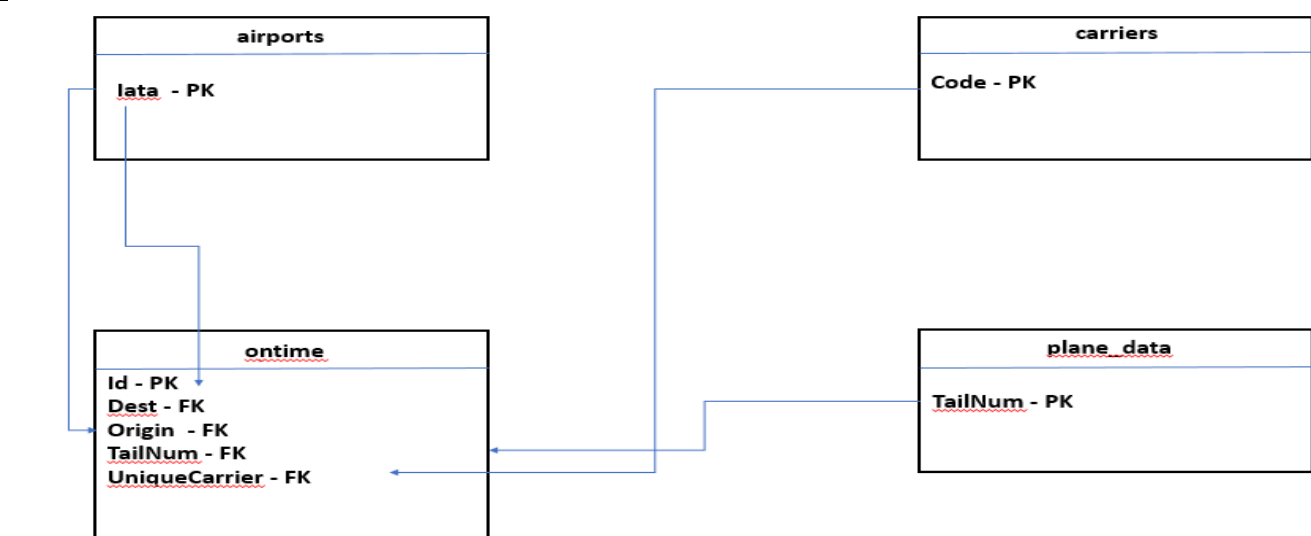


Figure 2 – High level database design. Only the Primary (PK) and Foreign (FK) keys have been represented.

The detailed table schema description is provided below. The columns of interest are highlighted in yellow.

Table Name	OnTime	
Column Name	Data Type	Description
Year	INT	1987-2008
Month	INT	1-12
DayofMonth	INT	1-31
DayOfWeek	INT	1 (Monday) - 7 (Sunday)
DepTime	INT	actual departure time (local, hhmm)
CRSDepTime	INT	scheduled departure time (local, hhmm)
ArrTime	INT	actual arrival time (local, hhmm)
CRSArrTime	INT	scheduled arrival time (local, hhmm)
UniqueCarrier	VARCHAR(5)	unique carrier code
FlightNum	INT	flight number
TailNum	VARCHAR(8)	plane tail number
ActualElapsedTime	INT	in minutes
CRSElapsedTime	INT	in minutes
AirTime	INT	in minutes
ArrDelay	INT	arrival delay, in minutes
DepDelay	INT	departure delay, in minutes
Origin	VARCHAR(3)	origin IATA airport code
Dest	VARCHAR(3)	destination IATA airport code
Distance	INT	in miles
TaxiIn	INT	taxi in time, in minutes
TaxiOut	INT	taxi out time in minutes
Cancelled	INT	was the flight cancelled?
CancellationCode	VARCHAR(1)	reason for cancellation (A = carrier, B = weather, C = NAS, D = security)
Diverted	VARCHAR(1)	1 = yes, 0 = no
CarrierDelay	INT	in minutes
WeatherDelay	INT	in minutes
NASDelay	INT	in minutes
SecurityDelay	INT	in minutes
LateAircraftDelay	INT	in minutes

Table Name	Airports	
Column Name	Data Type	Description
Iata	VARCHAR(3)	IATA airport code
Airport	VARCHAR(50)	Airport name
City	VARCHAR(50)	City where the airport is situated
State	VARCHAR(2)	State where the airport is situated
Country	VARCHAR(50)	Country where the airport is situated
Lat	FLOAT	Airport Latitude
Long	FLOAT	Airport Longitude

Table Name	Carriers	
Column Name	Data Type	Description
Code	VARCHAR(10)	Unique carrier id
Description	VARCHAR(100)	The long name of the carrier

Table Name	Plane_Data	
Column Name	Data Type	Description
Tailnum	VARCHAR(10)	The unique tail id
Type	VARCHAR(50)	The type of plane
Manufacturer	VARCHAR(100)	The manufacturer
Issue_Date	VARCHAR(10)	The date the plane started its service
Model	VARCHAR(50)	The plane model
Status	VARCHAR(20)	The plane status
Aircraft_Type	VARCHAR(50)	The type of plane
Year	INT	The year the plane started its service

Experiment Description

As shown in Figure 3 below, the first step in the experiment is to download the data from [1] onto a computer. Then move the data to the cluster file system via SSH. Subsequently, the data is loaded to HDFS. This is branching point where the Hive and Spark experiments are conducted.

- Hive:
 - The necessary schema is created.
 - The data is loaded from HDFS into the relevant Hive tables.
 - Numerous HiveQL DML query have been written to answer the domain questions.
 - Each query response time is recorded.
- Spark:
 - The data is read directly from HDFS and stored into RDDs.
 - The necessary Scala/RDDs code have been written to produce the results corresponding to the business questions.
 - Each result collection time is recoded

The aim is to compare the raw response time, the query complexity and the ability of each solution to answer the domain questions.

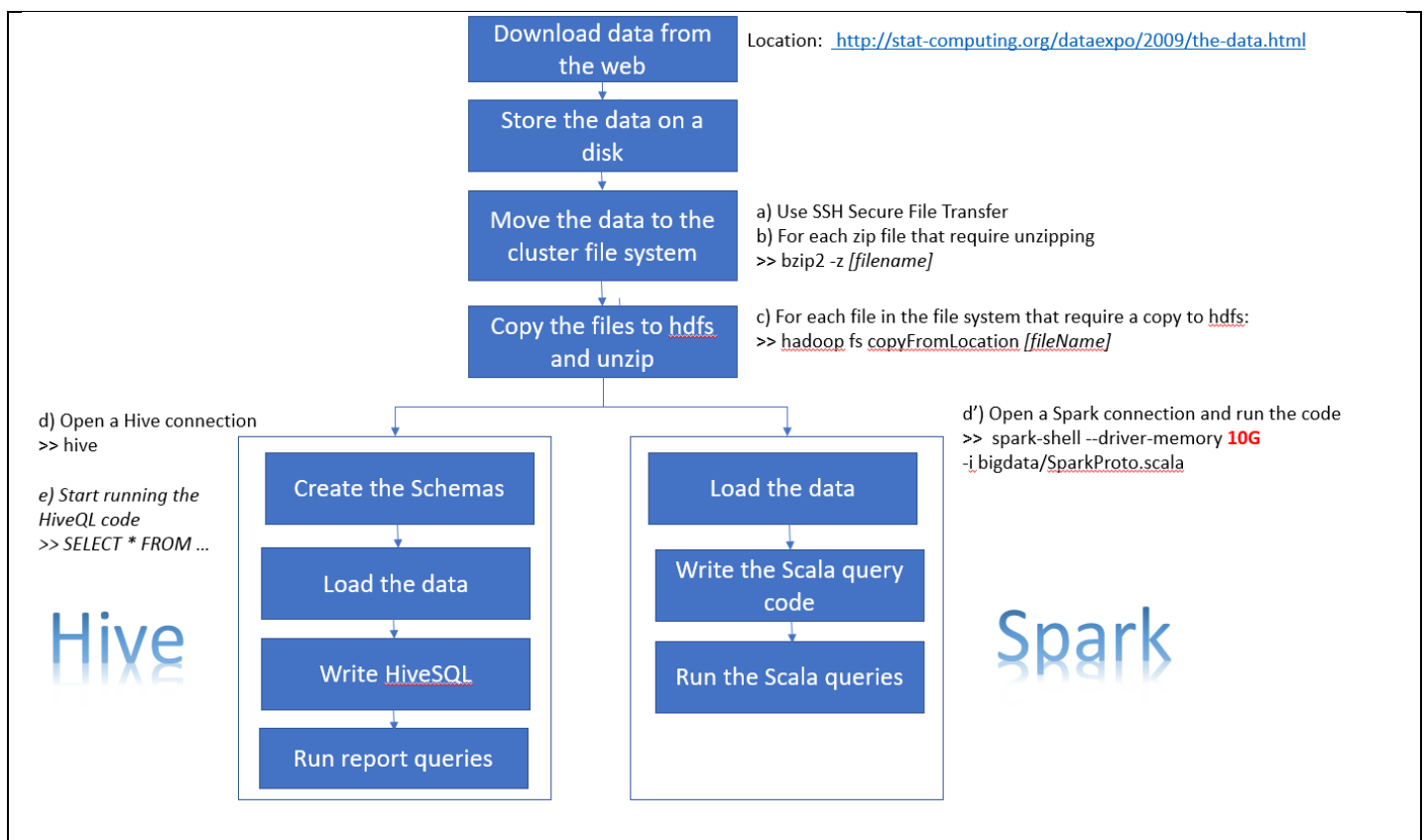


Figure 3 - The Hive Vs Spark process flow

Results

The first section list the breadth of functionalities supported by both solution. The second section details the implementation steps required in both cases. The third section illustrates the query time to response of both solutions.

High level functional comparison

Figure 4 below, borrowed from [2] sketches some of the Hadoop components and their interactions with the Hadoop infrastructure. The aim of this section is to describe Hive (SQL component) and Spark (Streaming component) and compare their functionality.

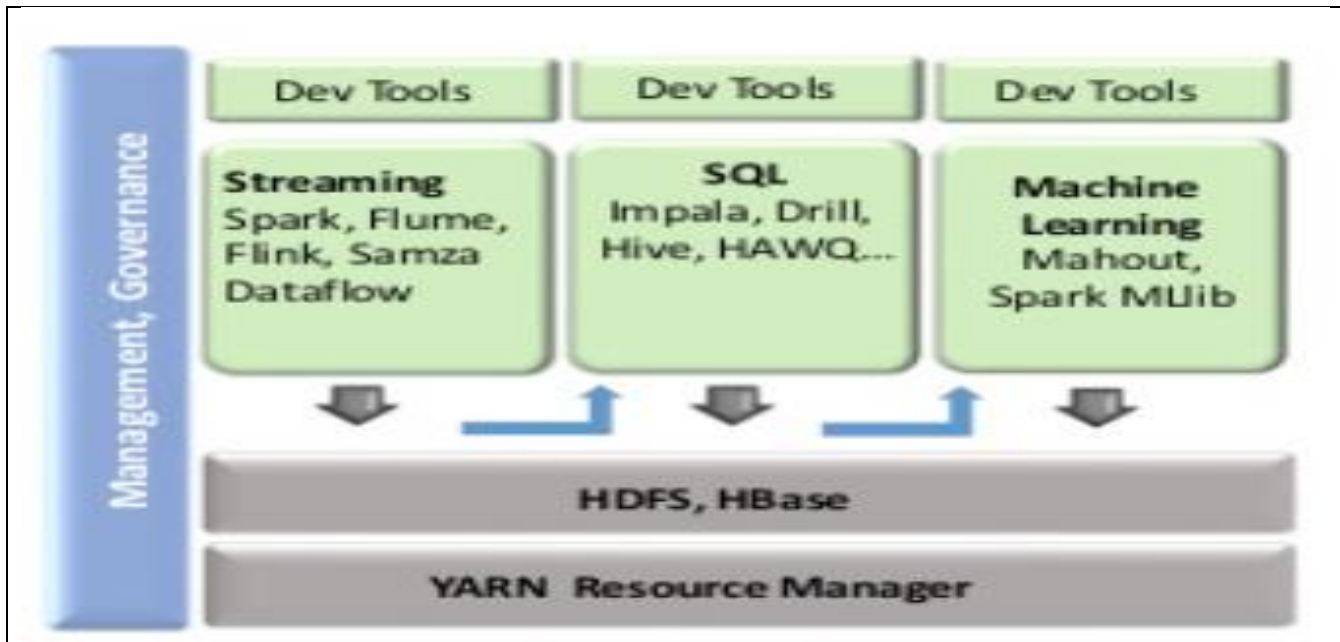


Figure 4 – High level Hadoop 2.0 infrastructure components

Hive

Apache Hive data warehouse is built at the top of Hadoop. It facilitates CRUD operations (Create, Read, Update, Delete) of large datasets, using a distributed storage and a SQL-like (HiveQL) interface [1]. Hive provides an abstraction layer above the Hadoop MapReduce. The HiveQL parser transforms the SQL-like statements into the MapReduce Java API automatically. This interface makes it easier to port and integrate SQL-based application to Hadoop.

The Hive main strengths are listed in the Table 1 below. Figure 5, borrowed from [3], supports Table 1 by showing the interaction between the different Hive components listed in Table 1.

Strengths	Comments
Maintenance	HiveQL supports UDFs and views to manipulate and encapsulate complex queries.
Robustness and Reliability	The metastore offers disaster-recovery functionality, as it replicates data regularly and can recover data in case of loss.
Extendibility/Interfacing	<p>Hive supports the analysis of large datasets stored in HDFS and other compatible file systems (e.g. Amazon S3 filesystem, RCFile, HBase, etc.)</p> <p>HiveQL can be transformed into MapReduce, Tez and/or Spark jobs. It also supports a CLI, a UI, and Thrift Server. The first two allow a user to interact with Hive by submitting DML/DDI queries, and monitoring the process status. Thrift server enable external clients to interact with Hive, in the same fashion as a JDBC or an ODBC servers would allow.</p>
Performance	<p>Hive provides indexes and partitions to improve query response time.</p> <p>The HiveQL optimiser automatically transforms the execution plan generated by the compiler into an optimized DAG. It can also split or join tasks if necessary to improve performance.</p>
Throughput	The use of distributed storage and parallel MapReduce jobs enable to cater for extra data load by extending the number of nodes (a.k.a. vertical extensibility).
Learning Curve	HiveQL is very close to standard ANSI SQL. The learning curve to produce HiveQL queries requires a low-level programming entry. Furthermore, the SQL skills is well spread in the programmer/business analyst community. Therefore, there is little of a risk related to skills shortage on the market place.

Table 1 – Hive main strengths

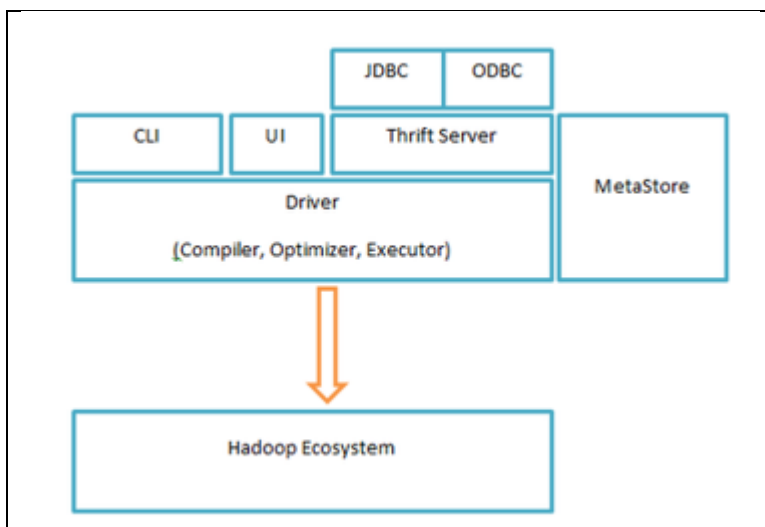


Figure 5 – High level Hive component interactions

Spark

Apache Spark stack was built in response to limitations of the MapReduce cluster computing model [5]. The development of Spark solutions is centered around the use of immutable and resilient distributed datasets (RDDs). RDDs are distributed over fault-tolerant machines clusters.

The Spark main strengths are listed in the Table 2, below. Figure 6, borrowed from [4], supports Table 5 by showing the interaction between the different Spark components listed in Table 2.

Strengths	Comments
Maintenance	<p>RDDs facilitates implementation of both iterative algorithms and explanatory data analysis (database query style).</p> <p>As Spark RDDs can be developed in Java/Scala/Python and other Object Oriented languages, the code can follow best practices for large software development. The user can implement the solution using design patterns and best architectural design practices.</p>
Robustness and Reliability	<p>The robustness and reliability can be achieved by engineering an infrastructure solution (including but not limited to message queues).</p> <p>RDDs can also be saved at given point in time and be recovered from these points onwards.</p>
Extendibility/Interfacing	<p>Spark natively communicate with numerous components, such as:</p> <ul style="list-style-type: none"> - Spark SQL: to execute Spark job using a pseudo SQL language. - Spark Streaming: a micro batching feature to simulate real-time interaction - MLlib: the Spark machine learning interface to generate prediction - GraphX: the Spark graphical interface. <p>Furthermore, Spark also interfaces with a wide variety of components such as , including HDFS, Cassandra, OpenStack Swift, Amazon S3, etc.</p>
Performance	<p>MapReduce programs require data to be i) read from disk, ii) mapped to perform actions across the data, iii) shuffled prior to the reduce task. The results of the reduce phase are then stored on disk.</p> <p>This is not the case in with Spark's RDDs. The data can be lazy loaded in memory, transformed in memory and saved to the disk when need be. This reduce read/write to disk time, network traffic and process intercommunication.</p>
Throughput	Clusters can be configured to self-adapt to high-throughput situations, e.g. when Spark Streaming is connected to an EMS such as Kafka.
Learning Curve	Developing Spark code requires more advanced development skills that may harder to find on the job market.
Running Environment	Can be run on the Hadoop platform interfacing through Yarn or as a standalone executor, independent from Hadoop.

Table 2 – Spark main strengths

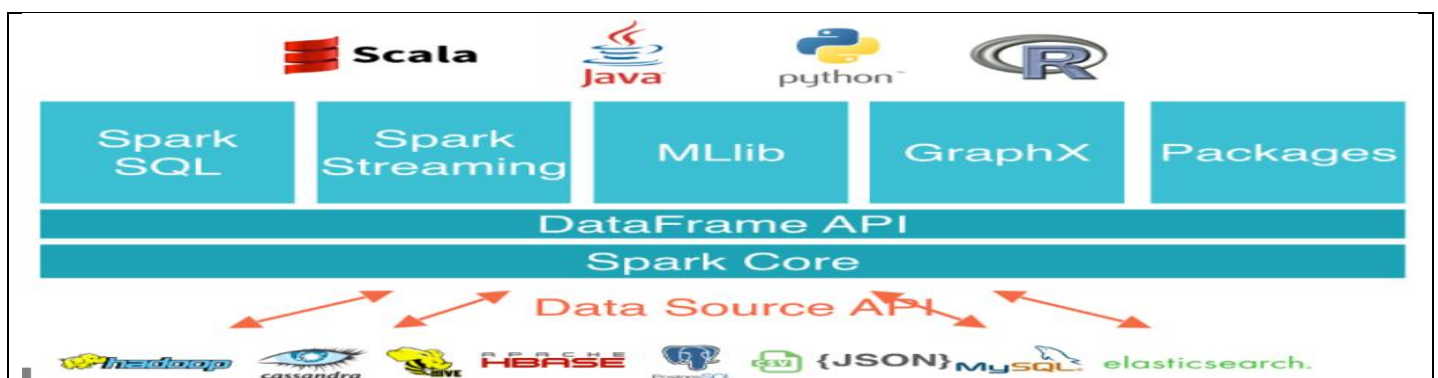


Figure 6 – High level Spark component interactions

Spark vs Hive

Functionality weaknesses of Hive are the strength of Spark and reverse. Hive, in its high-level usage is more akin to a traditional RDMS system. It enables the storage and retrieval of very high volume of data and it provides reporting capabilities. However, it suffers mainly from two limitations:

- i) Although Hive can interop with numerous external system, it is mainly biased towards CRUD operations and database performance tuning.
- ii) Hive is entirely dependent on the Hadoop platform and therefore dependent on the MapReduce performance.

Spark is more akin to a traditional second tier (server side component). Spark is biased towards reading/writing data from/to a data store and transforming the data in memory. Although Spark is natively feature rich, it contains several limitations:

- i) The Spark configuration to achieve optimised throughput and performance requires skills and experience.
- ii) A Spark code implementation requires specialist coding skills.
- iii) A Spark architecture that supports reliability and robustness (fault tolerance and data recovery) needs to be engineered upfront.
- iv) Managing the data load into RDDs needs to be managed with care, as the collection of RDDs implies a memory foot print. Loading very large datasets without proper filtering and/or partitioning may provoke out of memory exceptions. An example of such issue is shown in Log1 below:

```
17/02/27 23:32:58 ERROR util.SparkUncaughtExceptionHandler: Uncaught exception in thread Thread[Executor
task launch worker-3,5,main]
java.lang.OutOfMemoryError: GC overhead limit exceeded
    at java.nio.HeapCharBuffer.toString(HeapCharBuffer.java:561)
    at java.nio.CharBuffer.toString(CharBuffer.java:1201)
    ...
```

Log 1 – Out of Memory Exception in Spark

Query pre-fetch load time comparison

The time required to load the large airport csv file from the HDFS file system to the corresponding 'ontime' Hive table accounts for 190,624 milliseconds (c.f. Appendix B). The loading time of the extra small csv files was not recorded, as they account for a few milliseconds (ms).

On the Spark side, prior to running a query the following steps are required:

- Set-up Time (ms): 1,257
- Business Object Creation Time (ms): 1,543
- Shared Function Creation Time(ms): 1,944
- Data Load Time (ms): 6,191
- Data Union Creation Time (ms): 723
- Shared Transformed Time (ms): 371

This account for a total of 12,029 ms (c.f. Appending C for the detailed proofs).

Query Response Time & Complexity Comparison

This section compares the HiveQL and Spark Scala solution relating to answering domain questions. For each query the following parts are provided:

- The most ‘focused’ pieces of code, to compare the implementation complexity from a high-level view.
- The result of each query
- The query response time

Data Gathering

Query 1	HiveQL	Spark Scala
The routes the most affected by cancellations (by year).	<pre>SELECT cancelled.Year, cancelled.Origin, apOrigin.Airport, cancelled.Dest, apDest.Airport, cancelled.Sum_Cancelled FROM airport_flights.ontime_cancelled_view AS cancelled LEFT JOIN airport_flights.airports AS apOrigin ON cancelled.Origin = apOrigin.Iata LEFT JOIN airport_flights.airports AS apDest ON cancelled.Dest = apDest.Iata LIMIT 10;</pre>	<pre>val cancellations_df = all_delay_df .groupBy("Year", "Origin", "Dest") .agg(sum("Cancelled")); cancellations_df .join(airport_origin_df, cancellations_df("Origin") === airport_origin_df("Iata"), "left_outer") .join(airport_dest_df, cancellations_df("Dest") === airport_dest_df("Iata"), "left_outer") .select("Year", "Origin_Airport", "Origin", "Dest_Airport", "Dest", "SUM(Ca nelled)") .orderBy(desc("SUM(Cancelled)")) .show(5);</pre>
Results	<pre>2000 LAX Los Angeles International SFO San Francisco International 2205 2000 SFO San Francisco International LAX Los Angeles International 2198 1999 SFO San Francisco International LAX Los Angeles International 1751 1999 LAX Los Angeles International SFO San Francisco International 1747 1998 LAX Los Angeles International SFO San Francisco International 1702 1999 EWR Newark Intl ORD Chicago O'Hare International 1688 1998 SFO San Francisco International LAX Los Angeles International 1664 1999 ORD Chicago O'Hare International EWR Newark Intl 1648 2000 EWR Newark Intl ORD Chicago O'Hare International 1640 2000 ORD Chicago O'Hare International EWR Newark Intl 1578 Time taken: 84.558 seconds, Fetched: 10 row(s)</pre>	<pre>-----+-----+-----+-----+-----+ Year Origin_Airport Origin Dest_Airport Dest SUM(Cancelled) -----+-----+-----+-----+-----+ 2000 Los Angeles Inter... LAX null SFO 2205.0 2000 San Francisco Int... SFO null LAX 2198.0 1999 San Francisco Int... SFO null LAX 1751.0 1999 Los Angeles Inter... LAX null SFO 1747.0 1998 Los Angeles Inter... LAX null SFO 1702.0 -----+-----+-----+-----+-----+ timeElapsed: Long = 77771 ****Query 1 Response Time(ms): 77771</pre>
Response Time (ms)	84,558	77,771

The Spark ‘null’ values highlighted in red in the above table is a limitation in Spark. It was discovered while trying to join three tables. This restriction is discussed further in the below section named “Lessons Learnt & Challenges”.

Query 2	HiveQL	Spark Scala
The destinations the most affected by delays (by year)	SELECT dep_delay.Year, dep_delay.Dest, ap.Airport, dep_delay.Sum_DepDelay FROM airport_flights.ontime_dep_delay_view AS dep_delay LEFT JOIN airport_flights.airports AS ap ON dep_delay.dest = ap.IATA LIMIT 10;	val delay_per_destination_df = all_delay_df.groupBy("Year", "Dest", "Dest").agg(sum("DepDelay")); delay_per_destination_df .join(airport_dest_df, delay_per_destination_df("Dest") === airport_df("Iata"), "left_outer") .select("Year", "Dest_Airport", "Dest", "SUM(DepDelay)") .orderBy(desc("SUM(DepDelay)")) .show(10);
Results	<pre> 2007 ORD Chicago O'Hare International 5944152 2006 ORD Chicago O'Hare International 5656002 2008 ORD Chicago O'Hare International 5366358 2005 ATL William B Hartsfield-Atlanta Intl 4996238 2006 ATL William B Hartsfield-Atlanta Intl 4689461 2007 ATL William B Hartsfield-Atlanta Intl 4459995 2004 ORD Chicago O'Hare International 4299276 2004 ATL William B Hartsfield-Atlanta Intl 4111497 2000 ORD Chicago O'Hare International 4107347 2008 ATL William B Hartsfield-Atlanta Intl 3920548 Time taken: 77.827 seconds, Fetched: 10 row(s) </pre>	<pre> +-----+-----+-----+ Year Dest_Airport Dest SUM(DepDelay) +-----+-----+-----+ 2007 Chicago O'Hare In... ORD 5944152.0 2006 Chicago O'Hare In... ORD 5656002.0 2008 Chicago O'Hare In... ORD 5366358.0 2005 William B Hartsfi... ATL 4996238.0 2006 William B Hartsfi... ATL 4689461.0 2007 William B Hartsfi... ATL 4459995.0 2004 Chicago O'Hare In... ORD 4299276.0 2004 William B Hartsfi... ATL 4111497.0 2000 Chicago O'Hare In... ORD 4107347.0 2008 William B Hartsfi... ATL 3920548.0 +-----+-----+-----+ timeElapsed: Long = 74626 ****Query 2 Response Time(ms): 74626 </pre>
Response Time (ms)	77,827	74,626

Query 3	HiveQL	Spark Scala
The airlines that suffer from the longest delays (by year)	SELECT dep_delay_by_carrier.Year, dep_delay_by_carrier.UniqueCarrier, ca.Description, dep_delay_by_carrier.Sum_DepDelay FROM airport_flights.ontime_dep_delay_by_carrier_view AS dep_delay_by_carrier LEFT JOIN airport_flights.carriers AS ca ON CONCAT("","dep_delay_by_carrier.UniqueCarrier, """) = ca.Code WHERE Year IS NOT NULL LIMIT 10;	val delay_per_carrier_df = all_delay_df .groupBy("Year", "UniqueCarrier") .agg(sum("DepDelay")); delay_per_carrier_df .join(carrier_df, delay_per_carrier_df("UniqueCarrier") === carrier_df("Code"), "left_outer") .select("Year", "UniqueCarrier", "SUM(DepDelay)") .orderBy(desc("SUM(DepDelay)")) .show(10);
Results	<pre> dep_delay_by_carrier.year dep_delay_by_carrier.uniquecarrier ca.descr ption dep_delay_by_carrier.sum_depdelay 2006 HA "Hawaiian Airlines Inc." -63173 2005 HA "Hawaiian Airlines Inc." -53692 2007 HA "Hawaiian Airlines Inc." -49390 2008 AQ "Aloha Airlines Inc." -10844 2004 HA "Hawaiian Airlines Inc." -10680 2006 AQ "Aloha Airlines Inc." -3908 2003 HA "Hawaiian Airlines Inc." 8062 2000 AQ "Aloha Airlines Inc." 17268 2007 AQ "Aloha Airlines Inc." 20168 2008 HA "Hawaiian Airlines Inc." 27887 Time taken: 76.076 seconds, Fetched: 10 row(s) </pre>	<pre> +-----+-----+-----+ Year UniqueCarrier SUM(DepDelay) +-----+-----+-----+ 2000 UA 1.2895873E7 2008 WN 1.234954E7 2007 WN 1.2094777E7 2000 WN 1.1470445E7 2006 WN 1.1224333E7 2005 WN 1.0423665E7 2004 WN 9955591.0 1996 DL 9130025.0 2007 AA 9066828.0 1996 UA 8938005.0 +-----+-----+-----+ timeElapsed: Long = 73122 ****Query 3 Response Time(ms): 73122 </pre>
Response Time (ms)	76,076	73,122

Query 4.1	HiveQL	Spark Scala
The best 5 years to fly to minimise delays	SELECT Year, SUM(DepDelay) AS Sum_DepDelay FROM airport_flights.ontime GROUP BY Year HAVING Year IS NOT NULL ORDER BY Sum_DepDelay ASC LIMIT 5;	val delay_per_year_df = all_delay_df.groupBy("Year") .agg(sum("DepDelay")) .orderBy(asc("SUM(DepDelay)")) .select("Year", "SUM(DepDelay)") .select("Year", "SUM(DepDelay)"); delay_per_year_df.show(5);
Results	<pre> year sum_depdelay 1987 10419357 1992 28665029 2002 28802225 1991 28961203 1993 30678147 Time taken: 58.447 seconds, Fetched: 5 row(s) </pre>	<pre> +-----+-----+ Year SUM(DepDelay) +-----+-----+ 1987 1.0419357E7 1992 2.8665029E7 2002 2.8802225E7 1991 2.8961203E7 1993 3.0678147E7 +-----+-----+ timeElapsed: Long = 71504 ****Query 4.1 Response Time(ms): 71504 </pre>
Response Time (ms)	58,447	71,504

Query 4.2	HiveQL	Spark Scala
The best 5 months to fly to minimise delays	SELECT Month, SUM(DepDelay) AS Sum_DepDelay FROM airport_flights.ontime GROUP BY Month HAVING Month IS NOT NULL ORDER BY Sum_DepDelay ASC LIMIT 5;	val delay_per_year_month_df = all_delay_df.groupBy("Month") .agg(sum("DepDelay")) .orderBy(asc("SUM(DepDelay)")) .select("Month", "SUM(DepDelay)"); delay_per_year_month_df.show(5);
Results	<pre> OK 9 50306252 10 64267866 4 66540420 5 68519868 11 69198681 Time taken: 54.46 seconds, Fetched: 5 row(s) </pre>	<pre> +-----+-----+ Month SUM(DepDelay) +-----+-----+ 9 5.0306252E7 10 6.4267866E7 4 6.654042E7 5 6.8519868E7 11 6.9198681E7 +-----+-----+ timeElapsed: Long = 69607 ****Query 4.2 Response Time(ms): 69607 </pre>
Response Time (ms)	54,460	69,607

Query 4.3	HiveQL	Spark Scala
The best 5 days of month to fly to minimise delays	SELECT DayOfMonth, SUM(DepDelay) AS Sum_DepDelay FROM airport_flights.ontime GROUP BY DayOfMonth HAVING DayOfMonth IS NOT NULL ORDER BY Sum_DepDelay ASC LIMIT 5;	val delay_per_year_dayofmonth_df = all_delay_df .groupBy("DayOfMonth") .agg(sum("DepDelay")) .orderBy(asc("SUM(DepDelay)")) .select("DayOfMonth","SUM(DepDelay)"); delay_per_year_dayofmonth_df.show(5);
Results	<pre> OK 31 18359893 30 28371774 29 28765594 7 29066973 4 29345367 Time taken: 55.863 seconds, Fetched: 5 row(s) hive> </pre>	<pre> +-----+-----+ DayOfMonth SUM(DepDelay) +-----+-----+ 31 1.8359893E7 30 2.8371774E7 29 2.8765594E7 7 2.9066973E7 4 2.9345367E7 +-----+-----+ timeElapsed: Long = 69234 ****Query 4.3 Response Time(ms): 69234 </pre>
Response Time (ms)	55,863	69,234

4.4	HiveQL	Spark Scala
The best 5 days of week to fly to minimise delays	SELECT DayOfWeek, SUM(DepDelay) AS Sum_DepDelay FROM airport_flights.ontime GROUP BY DayOfWeek HAVING DayOfWeek IS NOT NULL ORDER BY Sum_DepDelay ASC LIMIT 5;	val delay_per_year_dayofweek_df = all_delay_df.groupBy("DayOfWeek") .agg(sum("DepDelay")) .orderBy(asc("SUM(DepDelay)")) .select("DayOfWeek","SUM(DepDelay)"); delay_per_year_dayofweek_df.show(5);
Results	<pre> OK 6 107798391 2 121250155 3 135731834 1 139666230 7 141995061 Time taken: 55.712 seconds, Fetched: 5 row(s) hive> </pre>	<pre> +-----+-----+ DayOfWeek SUM(DepDelay) +-----+-----+ 6 1.07798391E8 2 1.21250155E8 3 1.35731834E8 1 1.3966623E8 7 1.41995061E8 +-----+-----+ timeElapsed: Long = 70022 ****Query 4.4 Response Time(ms): 70022 </pre>
Response Time (ms)	55,712	70,022

Query 5	HiveQL	Spark Scala
Did older planes suffer more delays?	<pre>SELECT pd.Year, SUM(ot.DepDelay) AS Sum_DepDelay FROM airport_flights.ontime ot LEFT JOIN airport_flights.plane_data AS pd ON pd.Tailnum = ot.Tailnum GROUP BY pd.Year HAVING pd.Year IS NOT NULL ORDER BY Sum_DepDelay DESC LIMIT 20;</pre>	<pre>all_delay_df .join(plane_data_df, all_delay_df("TailNum") === plane_data_df("TailNum"), "left_outer") .groupBy("Year") .agg(sum("DepDelay")) .orderBy(desc("SUM(DepDelay)")) .show(20);</pre>
Results	<pre>pd.year sum_depdelay 2001 24411942 1999 23047982 2002 22070348 2000 21861081 1990 21630434 1998 21075689 1991 20840177 1988 20741604 2003 19992375 1992 19800412 1989 19066947 1987 18303923 2004 15745802 1994 12723062 1997 12371545 1986 11767217 1985 11081557 1996 10889979 2005 10778842 1993 10356982 Time taken: 67.959 seconds, Fetched: 20 row(s)</pre>	<pre>pd.year sum_depdelay 2001 24411942 1999 23047982 2002 22070348 2000 21861081 1990 21630434 1998 21075689 1991 20840177 1988 20741604 2003 19992375 1992 19800412 1989 19066947 1987 18303923 2004 15745802 1994 12723062 1997 12371545 1986 11767217 1985 11081557 1996 10889979 2005 10778842 1993 10356982</pre>
Response Time (ms)	67,959	64,561

Evaluation

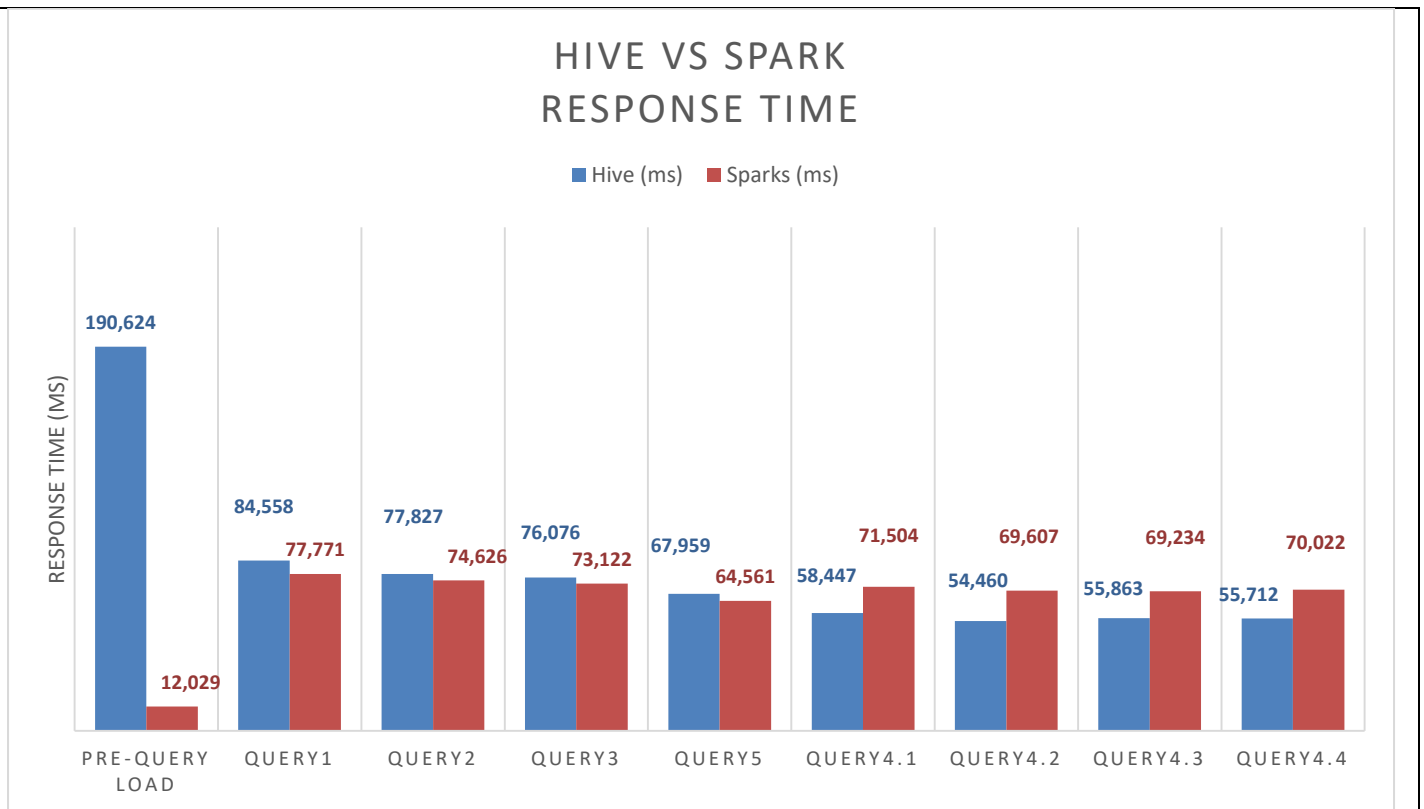
Key facts

Questions	Answers
What are the routes the most affected by cancellations (by year)?	The San Francisco International – Los Angeles route and reverse between 1998 and 2000.
What are the destinations the most affected by delays (by year)?	Chicago O'Hare International between 2006 and 2008
What is the airline that suffered from the longest delays (by year)?	Hawaiian Airlines Inc between 2005 and 2007
What were the best 5 years to fly to minimise delays?	1987,1992,2002,1991 and 1993 in order
What were the best 5 months to fly to minimise delays?	September, October, April, May and November in order.
What were the best 5 days of week to fly to minimise delays	Saturday, Tuesday, Wednesday, Thursday, Monday and Sunday in order
Did older planes suffer more	Generally, yes. Although it is difficult to draw an objective conclusion.

Response time

Graph 1 shows that the load time of the csv file into the Hive database takes 15 times longer than i) creating the Spark configuration environment, ii) instantiating the supporting business object and iii) generating the lazy loaded RDDs (that are ready for collection at a later stage). This is shown by the pre-query load buckets in Graph 1. Indeed, Hive fronts load all the data into the necessary tables, whereas Spark create the 'container' object in advance. However, Spark only loads the data on demand, when collected. This graph also shows that, on average, querying the data in Hive is 7% faster than in Spark. However, this average hides an interesting pattern:

- More complex queries (i.e. including joins) run faster on Spark (c.f. query1/2/3/5) by 5% (std dev = 2%)
- Simpler queries (i.e. not including joins) run faster on hive (c.f. queries4.1/.../4.4), by 25% (std dev = 2%).

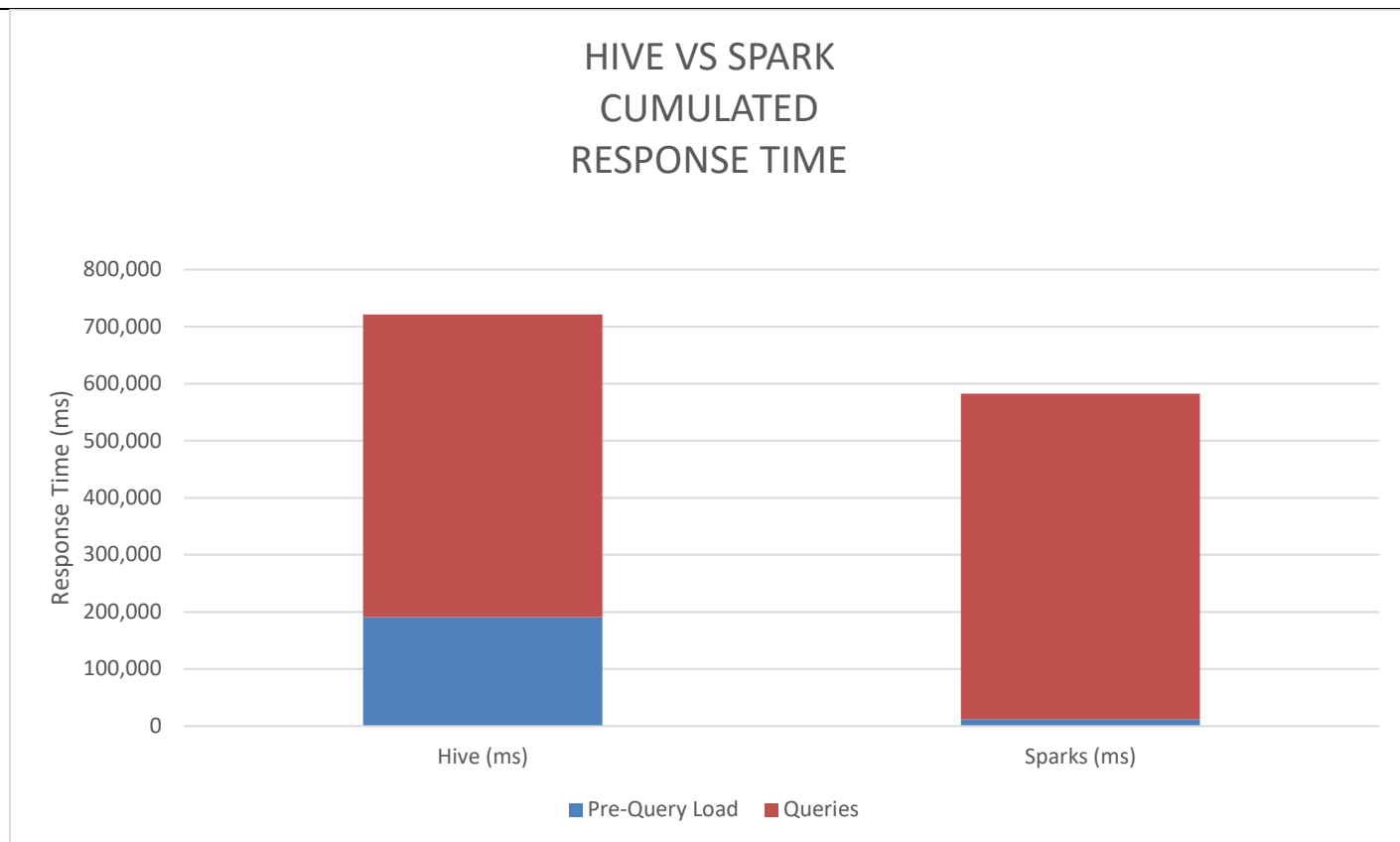


		Hive (ms)	Sparks (ms)	Response Time Ratio
Pre-Query Load	Contains Joins	190,624	12,029	
Query1	Yes	84,558	77,771	8%
Query2	Yes	77,827	74,626	4%
Query3	Yes	76,076	73,122	4%
Query5	Yes	67,959	64,561	5%
Query4.1	No	58,447	71,504	-22%
Query4.2	No	54,460	69,607	-28%
Query4.3	No	55,863	69,234	-24%
Query4.4	No	55,712	70,022	-26%

Contains Joins/Response Time			
Ratio	Avg	Std Dev	
Yes	5%	2%	
No	-25%	2%	

Graph 1 – A comparison of the response times for data preloading and query runs.

Graph2 shows that cumulating the response times, relating to front loading and querying, implies that Hive is overall 19% slower than Spark. This has an impact in terms of the choice of solution depending on the business need. Indeed, if the use case relates to loading the data once day, as a batch, and having business users only querying the data though the day, then Hive could be a better solution. If the use case relates to real-time data, then Spark would certainly be the preferred solution, as the pre-query load is minimum.



	Hive (ms)	Sparks (ms)	Response Time Ratio
Pre-Query Load	190,624	12,029	
Queries	530,902	570,447	
Total	721,526	582,476	81%

Graph 2 – The cumulated response times for Hive and Spark

Language support

As expected writing Spark Scala code is more involved than writing HiveQL code. However, Spark Scala is more flexible (as it is an Object-Oriented language).

Lessons Learnt & Challenges

This section highlights some of Hive's and Spark's idiosyncrasies, that may be counter intuitive for a developer used to write SQL for RDBMS, or server code in a language that differs from Scala for RDDs.

Spark

- As shown in Figure 3 above (spark-shell --driver-memory **10G** -i bigdata/SparkProto.scala), when a spark script is run, its memory should be allocated to match the script query memory foot print required. If it is not defined or set to a level inferior to the one needed, some or all the RDD collections will return an empty set. As no warning or error message show in the log, it is an issue that is complex to trouble shoot.
- The following libraries highlighted in blue, in Code snippet 1, need to be written in the Scala script just after the spark context is created. Failing this, the data frame will return an empty set. Once again, the logging is poor around this. It is easy to miss and it takes a lot of time to realise what the issue might be.

```
config.set("fs.defaultFS", uri);
var folderPath = "hdfs://user/fmare001/"
//Create the SQLContext
val sc: SparkContext;
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
// For implicit conversions like converting RDDs to DataFrames
import org.apache.spark.implicit._
import sqlContext.implicit._
```

Code snippet 1 – Dataframe dependent libraries

- The syntax for writing left join is counter intuitive and the official Spark documentation is a bit light on this area.

```
//Now join on the airport_dest_df and airport_origin_df to get the readable airport names
cancellations_df.join(airport_origin_df, cancellations_df("Origin") == airport_df("Iata"), "left_outer").join(
airport_dest_df, cancellations_df("Dest") == airport_df("Iata"), "left_outer").select("Year", "Origin_Airport", "Origin",
"Dest_Airport", "Dest", "SUM(Cancelled)").show();
```

Code snippet 2 – Writing a join in Scala

- The results generated in Spark, from joining the main data frame ('cancellations_df') to two other data frames ('airport_origin_df' and 'airport_dest_df'), display unexpected 'null' data. The join on the second data frame ('airport_dest_df') fails, which result in the production of 'null' values (circled in red). This issue could be resolved in three steps:
 - Joining the 'cancellations_df' data frame to 'airport_origin_df'.
 - Joining the 'cancellations_df' data frame to 'airport_dest_df'.
 - Finally join the results from the two above joins.

This will remove the 'null' data issue, but this solution has a higher memory foot print.

```
val cancellations_df = all_delay_df
.groupBy("Year", "Origin", "Dest")
.agg(sum("Cancelled"));

cancellations_df
.join(airport_origin_df, cancellations_df("Origin") == airport_origin_df("Iata"), "left_outer")
.join(airport_dest_df, cancellations_df("Dest") == airport_dest_df("Iata"), "left_outer")
.select("Year", "Origin_Airport", "Origin", "Dest_Airport", "Dest", "SUM(Cancelled)")
.orderBy(desc("SUM(Cancelled)"))
.show(5);
```

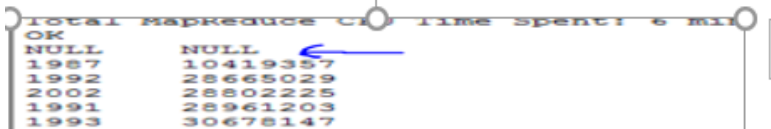
Year	Origin_Airport	Origin	Dest_Airport	Dest	SUM(Cancelled)
2000	Los Angeles Inter...	LAX	null	SFO	2205.0
2000	San Francisco Int...	SFO	null	LAX	2198.0
1999	San Francisco Int...	SFO	null	LAX	1751.0
1999	Los Angeles Inter...	LAX	null	SFO	1747.0
1998	Los Angeles Inter...	LAX	null	SFO	1702.0

```
timeElapsed: Long = 77771
****Query 1 Response Time (ms): 77771
```

Hive

- When a Hive process is running and an unexpected server disconnection happens, then it is not possible to request a new Hive process to be ran. The user needs to kill the currently running Hive process manually as the following:
 - Find the running Hive process id: `ps -ef | grep hive`
 - Kill the process: `kill -9 [processId]`
- Extra 'NULL' values occur in columns that are not of type string. Therefore, it needs to be filtered out by the query. As shown in Table 3, the default result of the following query returns an extra NULL row:

```
SELECT COUNT(Year)
FROM airport_flights.ontime
GROUP BY Year;
```



OK	
NULL	NULL
1987	10419357
1992	28465029
2002	28802225
1991	28961203
1993	30678147

Table 3 – Hive extra NULL row

- Column headers are not printed by default. The user can request the column print as part a SELECT query by preceding it with this statement: **`SET hive.cli.print.header=TRUE;`**

Potential Improvements

This section only focuses on a small list of potential improvements that could be applied to the current solution to reduce maintenance and improve performance.

Hive

- Index – Indices help with the query response time. There are two types of indices that identify the data across the nodes in a cluster [6]:
 - “Compact indexing stores the pair of indexed column’s value and its blockid.”
 - “Bitmap indexing stores the combination of indexed column value and list of rows as a bitmap”
- Partitions – Hive allows for vertical and horizontal partitions [7]. This is a way to divide a table in related parts (horizontal) and buckets (vertical) to improve queries efficiencies.

Spark

- Use `sc.parallelize()` function to copy the elements of the collection into a distributed dataset that can be operated in parallel, e.g. `val distData = sc.parallelize(data)`.
- Spark 2.0 provides numerous improvements to SparkSQL, which helps with the code maintenance and query time response [8].
- Spark 2.0 supports pulling data sets into a cluster-wide in-memory cache. This is useful when a piece of data needs to be accessed frequently.
- GraphX can be used to generate graphics.
- MLlib can be used to perform predictions.

Conclusion

The report showed that Hive and Spark are adequate big data solutions to answer the airport problem domain questions. When the pre-fetch and queries are run synchronously, Spark shows a small advantage over Hive in terms of response. However, this experiment should be tested a larger scale dataset, with frequent simple and complex query requests, to establish whether this still holds true. Depending on the use case, whether batch or real-time, and the need for distribution of the transformed results, one or the other solution could be viewed as optimal. Spark provides a rich feature set (e.g. a machine learning library, graphical capabilities, etc.) but it comes at a price in terms of code development and maintenance. Specialist skill sets are required for Spark, which is not the case with Hive.

Bibliography

- [0] Airline on-time performance [Online]. Available At: <http://stat-computing.org/dataexpo/2009/> [Accessed: 25-February-2017]
- [1] Apache Hive TM [Online]. Available At: <https://hive.apache.org/> [Accessed: 06-March-2017]
- [2] Big Data Platform Evolution – Simplifying Operations and Development [Online]. Available At: <https://www.bing.com/images/search?view=detailV2&iss=VS&imgurl=http%3a%2f%2fimage.slidesharecdn.com%2fggilbertpresentation-151006211550-lva1-app6891%2f95%2fsystems-of-intelligence-the-biggest-change-in-enterprise-applications-in-50-years-6-638.jpg> [Accessed: 06-March-2017]
- [3] Apache Hive[Online]. Available At: https://en.wikipedia.org/wiki/Apache_Hive [Accessed: 06-March-2017]
- [4] [Online]. Available At: https://www.bing.com/images/search?view=detailV2&ccid=dnyzRP5S&id=552179C7FBF35F3F0D93815334261860A090F027&q=Spark+Framework&simid=608000566852781609&selectedindex=3&mode=overlay&first=1&thid=OIP.dnyzRP5SHx6bPDLrsxUm_QEsCz [Accessed: 06-March-2017]
- [5] Apache Spark [Online]. Available At: https://en.wikipedia.org/wiki/Apache_Spark [Accessed: 06-March-2017]
- [6] Hive Indexing [Online]. Available At: <https://acadgild.com/blog/indexing-in-hive/> [Accessed: 14-March-2017]
- [7] Hive – Partitioning [Online]. Available At: https://www.tutorialspoint.com/hive/hive_partitioning.htm [Accessed: 14-March-2017]
- [8] Apache Spark™ 2.0: Impressive Improvements to Spark SQL [Online]. Available At: <http://spark.tc/apache-spark-2-0-2-spark-sql/>

Appendix

Appendix A – Spark job run exception

```
17/03/08 20:53:12 INFO scheduler.TaskSetManager: Finished task 85.0 in stage 0.0
(TID 85) in 8910 ms on localhost (87/97)
17/03/08 20:53:12 INFO executor.Executor: Running task 1.0 in stage 1.0 (TID 98)
17/03/08 20:53:12 INFO rdd.HadoopRDD: Input split: hdfs://dsml:9000/user/fmare00
1/plane-data.csv:214398+214398
17/03/08 20:53:12 ERROR executor.Executor: Exception in task 0.0 in stage 1.0 (T
ID 97)
java.lang.ArrayIndexOutOfBoundsException: 1
    at $line66.$read$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$
$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$anonfun$2.apply(<console>:49)
    at $line66.$read$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$
$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$anonfun$2.apply(<console>:49)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:328)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:328)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:328)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:328)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:328)
    at org.apache.spark.util.collection.WritablePartitionedIterator$$anon$3.
writeNext(WritablePartitionedPairCollection.scala:105)
    at org.apache.spark.util.collection.ExternalSorter.spillToPartitionFiles
(ExternalSorter.scala:375)
    at org.apache.spark.util.collection.ExternalSorter.insertAll(ExternalSor
ter.scala:208)
    at org.apache.spark.shuffle.sort.SortShuffleWriter.write(SortShuffleWrit
er.scala:62)
    at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scal
a:70)
    at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scal
a:41)
    at org.apache.spark.scheduler.Task.run(Task.scala:70)
```

Appendix B– HDFS to Hive table load time proofs

```
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/2008.csv' OVERWRITE INTO TA
BLE ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=1, numRows=0, totalSize=689413344,
  rawDataSize=0]
OK
Time taken: 7.536 seconds, Fetched: 1 row(s)
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/2006.csv' INTO TABLE airpor
t_flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=3, numRows=0, totalSize=2064359633
, rawDataSize=0]
OK
Time taken: 6.962 seconds
Time taken: 21.813 seconds, Fetched: 1 row(s)
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/2007.csv' INTO TABLE airpor
t_flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=1, numRows=0, totalSize=702878193,
  rawDataSize=0]
OK
Time taken: 8.843 seconds
```



```

hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/2005.csv' INTO TABLE airport_
t_flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=4, numRows=0, totalSize=2735386898,
rawDataSize=0]
OK
Time taken: 7.569 seconds
hive>
  > LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/2004.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=5, numRows=0, totalSize=3405266011,
rawDataSize=0]
OK
Time taken: 7.049 seconds
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/2003.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=6, numRows=0, totalSize=4032011253,
rawDataSize=0]
OK
Time taken: 8.726 seconds

hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/2002.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=7, numRows=0, totalSize=4562518266,
rawDataSize=0]
OK
Time taken: 7.144 seconds
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/2001.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=8, numRows=0, totalSize=5162929728,
rawDataSize=0]
OK
Time taken: 6.268 seconds
hive>
  > LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/2000.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=9, numRows=0, totalSize=5733081341,
rawDataSize=0]
OK
Time taken: 7.056 seconds
hive>

```

```

hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1999.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=10, numRows=0, totalSize=6286007363,
  rawDataSize=0]
OK
Time taken: 5.608 seconds
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1998.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=11, numRows=0, totalSize=6824440238,
  rawDataSize=0]
OK
Time taken: 6.818 seconds
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1997.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=12, numRows=0, totalSize=7364788099,
  rawDataSize=0]
OK
Time taken: 5.746 seconds
hive>
  > LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1996.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=13, numRows=0, totalSize=7898710462,
  rawDataSize=0]
OK
Time taken: 6.182 seconds
hive>
  > LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1995.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=14, numRows=0, totalSize=8429462030,
  rawDataSize=0]
OK
Time taken: 6.498 seconds
hive>
  > LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1994.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=15, numRows=0, totalSize=8931020695,
  rawDataSize=0]
OK
Time taken: 6.486 seconds
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1993.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=16, numRows=0, totalSize=9421774347,
  rawDataSize=0]
OK
Time taken: 5.171 seconds
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1992.csv' INTO TABLE airport_
flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=17, numRows=0, totalSize=9914088078, rawDataSize=0]
OK
Time taken: 5.152 seconds
hive>
  > LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1991.csv' INTO TABLE airport_flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=18, numRows=0, totalSize=10405298171, rawDataSize=0]
OK
Time taken: 5.264 seconds

```

```

hive>
> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1990.csv' INTO TABLE airport_flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=19, numRows=0, totalSize=10914492858, rawDataSize=0]
OK
Time taken: 5.764 seconds
hive>
> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1989.csv' INTO TABLE airport_flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=20, numRows=0, totalSize=11401011679, rawDataSize=0]
OK
Time taken: 5.412 seconds
hive> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1987.csv' INTO TABLE airport_flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=21, numRows=0, totalSize=11528174621, rawDataSize=0]
OK
Time taken: 1.689 seconds
> LOAD DATA LOCAL INPATH '/home/fmare001/bigdata/1988.csv' INTO TABLE airport_flights.ontime;
Loading data to table airport_flights.ontime
Table airport_flights.ontime stats: [numFiles=22, numRows=0, totalSize=12029214093, rawDataSize=0]
OK
Time taken: 5.225 seconds

```

Appendix C– Sparks object building timings proofs (prior to querying)

```

sqlContext: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@25f639

```

```

<console>:36: error: object implicits is not a member of package org.apache.spark

```

```

import org.apache.spark.implicits._
      ^

```

```

import sqlContext.implicits._

```

```

timeElapsed: Long = 1257

```

```

****Set-up Time(ms): 1257

```

```

now: Long = 1489060224511

```

```

defined class DelayRow

```

```

defined class AirportRow

```

```

defined class CarrierRow

```

```

defined class PlaneDataRow

```

```

timeElapsed: Long = 1543

```

```

****Business Object Creation Time(ms): 1543

```

```

timeElapsed: Long = 1944

```

```

****Shared Function Creation Time(ms): 1944

```

```

, diverted: String]

```

```

timeElapsed: Long = 6191

```

```

****Data Load Time(ms): 6191

```

```

****Data Union Creation Time(ms): 723

```

```

timeElapsed: Long = 723

```

```

****Data Union Creation Time(ms): 723

```

```

timeElapsed: Long = 371

```

```

****Shared Transforms Creation Time(ms): 371

```