Technical Paper

Data Modeling Considerations in Hadoop and Hive

Clark Bradley, Ralph Hollinshead, Scott Kraus, Jason Lefler, Roshan Taheri

October 2013

Table of Contents

Introduction

It would be an understatement to say that there is a lot of buzz these days about big data. Because of the proliferation of new data sources such as machine sensor data, medical images, financial data, retail sales data, radio frequency identification, and web tracking data, we are challenged to decipher trends and make sense of data that is orders of magnitude larger than ever before. Almost every day, we see another article on the role that big data plays in improving profitability, increasing productivity, solving difficult scientific questions, as well as many other areas where big data is solving problems and helping us make better decisions. One of the technologies most often associated with the era of big data is Apache Hadoop.

Although there is much technical information about Hadoop, there is not much information about how to effectively structure data in a Hadoop environment. Even though the nature of parallel processing and the MapReduce system provide an optimal environment for processing big data quickly, the structure of the data itself plays a key role. As opposed to relational data modeling, structuring data in the Hadoop Distributed File System (HDFS) is a relatively new domain. In this paper, we explore the techniques used for data modeling in a Hadoop environment. Specifically, the intent of the experiments described in this paper was to determine the best structure and physical modeling techniques for storing data in a Hadoop cluster using Apache Hive to enable efficient data access. Although other software interacts with Hadoop, our experiments focused on Hive. The Hive infrastructure is most suitable for traditional data warehousing-type applications. We do not cover Apache HBase, another type of Hadoop database, which uses a different style of modeling data and different use cases for accessing the data.

In this paper, we explore a data partition strategy and investigate the role indexing, data types, files types, and other data architecture decisions play in designing data structures in Hive. To test the different data structures, we focused on typical queries used for analyzing web traffic data. These included web analyses such as counts of visitors, most referring sites, and other typical business questions used with weblog data.

The primary measure for selecting the optimal structure for data in Hive is based on the performance of web analysis queries. For comparison purposes, we measured the performance in Hive and the performance in an RDBMS. The reason for this comparison is to better understand how the techniques that we are familiar with using in an RDBMS work in the Hive environment. We explored techniques such as storing data as a compressed sequence file in Hive that are particular to the Hive architecture.

Through these experiments, we attempted to show that how data is structured (in effect, data modeling) is just as important in a big data environment as it is in the traditional database world.

Understanding HDFS and Hive

Similar to massively parallel processing (MPP) databases, the power of Hadoop is in the parallel access to data that can reside on a single node or on thousands of nodes. In general, MapReduce provides the mechanism that enables access to each of the nodes in the cluster. Within the Hadoop framework, Hive provides the ability to create and query data on a large scale with a familiar SQL-based language called HiveQL. It is important to note that in these experiments, we strictly used Hive within the Hadoop environment. For our tests, we simulated a typical data warehouse-type workload where data is loaded in batch, and then queries are executed to answer strategic (not operational) business questions.

According to the Apache Software Foundation, here is the definition of Hive:

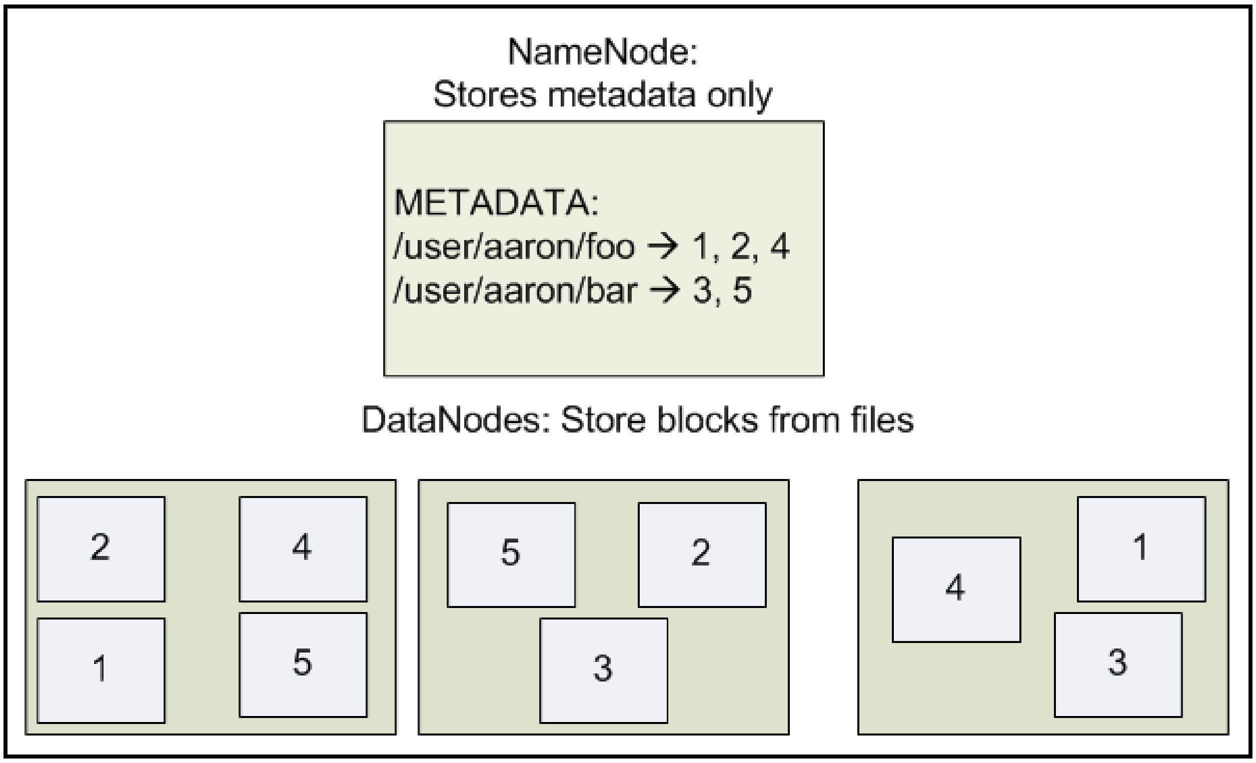
> "Hive is a data warehouse system for Hadoop that facilitates easy data summarization, ad-hoc queries, and the analysis of large datasets stored in Hadoop compatible file systems. Hive provides a mechanism to project structure onto this data and query the data using a SQL-like language called HiveQL. At the same time this language also allows traditional map/reduce programmers to plug in their custom mappers and reducers when it is inconvenient or inefficient to express this logic in HiveQL."

To demonstrate how to structure data in Hadoop, our examples used the Hive environment. Using the SAS/ACCESS engine, we were able to run our test queries through the SAS interface, which is executed in the Hive environment within our Hadoop cluster. In addition, we performed a cursory examination of Impala, the "SQL on top of Hadoop" tool offered by Cloudera.

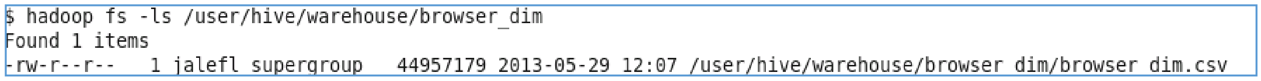
All queries executed through SAS/ACCESS to Hadoop were submitted via the Hive environment and were translated into MapReduce jobs. Although it is beyond the scope of this paper to detail the inner-workings of MapReduce, it is important to understand how data is stored in HDFS when using Hive to better understand how we should structure our tables in Hadoop. By gaining some understanding in this area, we are able to appreciate the effect data modeling techniques have in HDFS.

In general, all data stored in HDFS is broken into blocks of data. We used Cloudera's distribution of version 4.2 of Hadoop for these experiments. The default size of each data block in Cloudera Hadoop 4.2 is 128 MB. As shown in Figure 1, the same blocks of data were replicated across multiple nodes to provide reliability if a node failed, and also to increase the performance during MapReduce jobs. Each block of data is replicated three times by default in the Hadoop environment. The NameNode in the Hadoop cluster serves as the metadata repository that describes where blocks of data are located for each file stored in HDFS.

Figure 1: HDFS Data Storage[5]

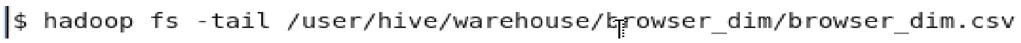


At a higher level, when a table is created through Hive, a directory is created in HDFS on each node that represents the table. Files that contain the data for the table are created on each of the nodes, and the Hive metadata keeps track of where the files that make up each table are located. These files are located in a directory with the name of the table in HDFS in the **`/user/hive/warehouse`** folder by default. For example, in our tests, we created a table named BROWSER_DIM. We can use an HDFS command to see the new table located in the **`/user/hive/warehouse`** directory. By using the command **`hadoop fs -ls`**, the contents of the **`browser_dim`** directory are listed. In this directory, we find a file named browser_dim.csv. HDFS commands are similar to standard Linux commands.

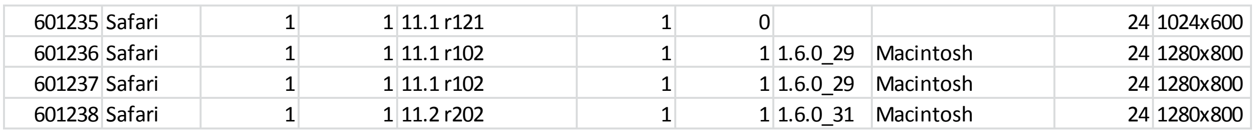


```
$ hadoop fs -ls /user/hive/warehouse/browser_dim
Found 1 items
-rw-r--r--   1 jalefl supergroup   44957179 2013-05-29 12:07 /user/hive/warehouse/browser_dim/browser_dim.csv
```

By default, Hadoop distributes the contents of the browser_dim table into all of the nodes in the Hadoop cluster. The following **`hadoop fs –tail`** command lists the last kilobyte of the file listed:



```
$ hadoop fs -tail /user/hive/warehouse/browser_dim/browser_dim.csv
```



601235	Safari	1	1	11.1 r121	1	0			24	1024x600
601236	Safari	1	1	11.1 r102	1	1	1.6.0_29	Macintosh	24	1280x800
601237	Safari	1	1	11.1 r102	1	1	1.6.0_29	Macintosh	24	1280x800
601238	Safari	1	1	11.2 r202	1	1	1.6.0_31	Macintosh	24	1280x800

The important takeaway is to understand at a high level how data is stored in HDFS and managed in the Hive environment. The physical data modeling experiments that we performed ultimately affect how the data is stored in blocks in HDFS and in the nodes where the data is located and how the data is accessed. This is particularly true for the tests in which we partitioned the data using the **`Partition`** statement to redistribute the data based on the buckets or ranges defined in the partitions.

Project Environment

Hardware

The project hardware was designed to emulate a small-scale Hadoop cluster for testing purposes, not a large-scale production environment. Our blades had only two CPUs each. Normally, Hadoop cluster nodes have more. However, the size of the cluster and the data that we used are large enough to make conclusions about physical data modeling techniques. As shown in Figure 2, our hardware configuration was as follows:

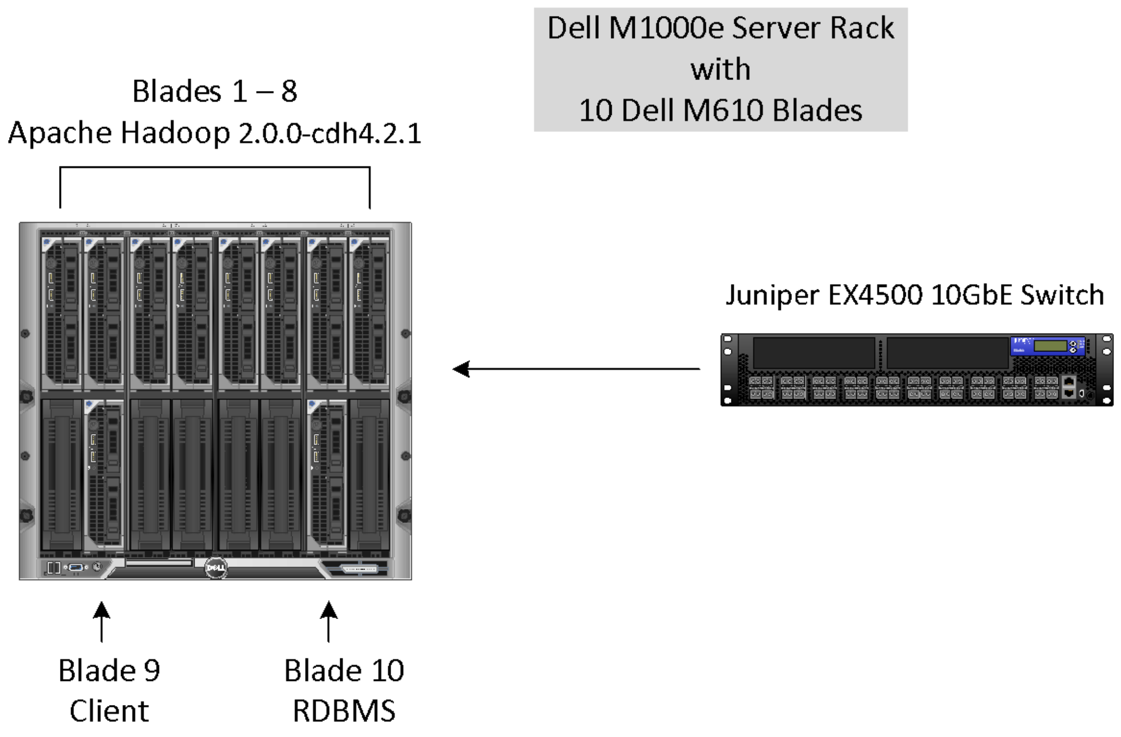
Overall hardware configuration:

- 1 Dell M1000e server rack
- 10 Dell M610 blades
- Juniper EX4500 10 GbE switch

Blade configuration:

- Intel Xeon X5667 3.07GHz processor
- Dell PERC H700 Integrated RAID controller
- Disk size: 543 GB
- FreeBSD iSCSI Initiator driver
- HP P2000 G3 iSCSI dual controller
- Memory: 94.4 GB
- Linux 2.6.32

Figure 2: The Project Hardware Environment



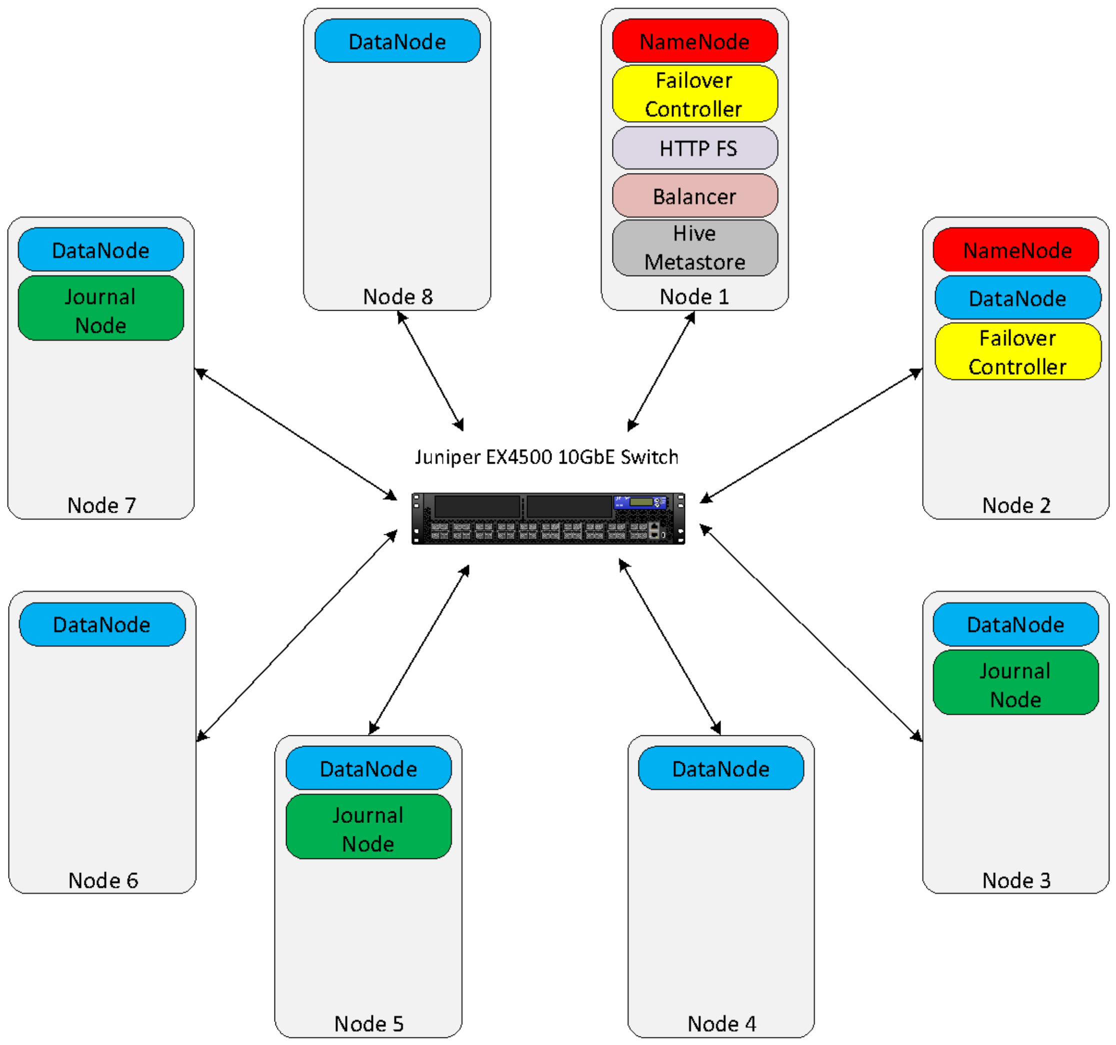
Software

The project software created a small-scale Hadoop cluster and included a standard RDBMS server and a client server with release 9.3 of Base SAS software with supporting software.

The project software included the following components:

- CDH (Cloudera's Distribution Including Apache Hadoop) version 4.2.1
    - Apache Hadoop 2.0.0
    - Apache Hive 0.10.0
    - HUE (Hadoop User Experience) 2.2.0
    - Impala 1.0
    - Apache MapReduce 0.20.2
    - Apache Oozie 3.3.0
    - Apache ZooKeeper 3.4.5
- Apache Sqoop 1.4.2
- Base SAS 9.3
- A major relational database

Figure 3: The HDFS Architecture



The Hadoop Cluster

The Hadoop cluster can logically be divided into two areas: HDFS, which stores the data, and MapReduce, which processes all of the computations on the data (with the exception of a few tests where we used Impala).

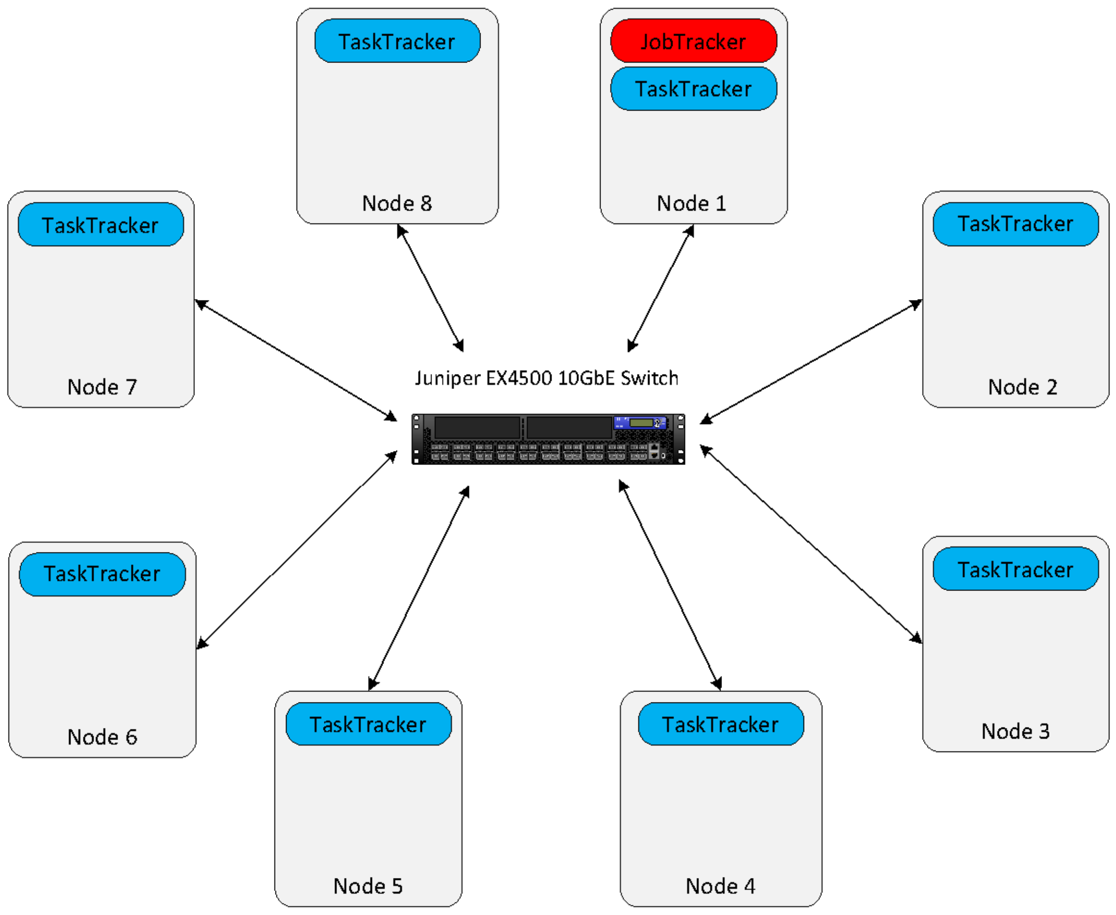
The NameNode on nodes 1 and 2 and the JobTracker on node 1 (in the next figure) serve as the master nodes. The other six nodes are acting as slaves.[1]

Figure 3 shows the daemon processes of the HDFS architecture, which consist of two NameNodes, seven DataNodes, two Failover Controllers, three Journal Nodes, one HTTP FS, one Balancer, and one Hive Metastore. The NameNode located on blade Node 1 is designated as the active NameNode. The NameNode on Node 2 is serving as the standby. Only one NameNode can be active at a time. It is responsible for controlling the data storage for the cluster. When the NameNode on Node 2 is active, the DataNode on Node 2 is disabled in accordance with accepted HDFS procedure. The DataNodes act as instructed by the active NameNode to coordinate the storage of data. The Failover Controllers are daemons that monitor the NameNodes in a high-availability environment. They are responsible for updating the ZooKeeper session information and initiating state transitions if the health of the associated NameNode wavers.[2] The

JournalNodes are written to by the active NameNode whenever it performs any modifications in the cluster. The standby NameNode has access to all of the modifications if it needs to transition to an active state.[3] The HTTP FS provides the interface between the operating system on the server and HDFS.[4] The Balancer utility distributes the data blocks across the nodes evenly.[5] The Hive Metastore contains the information about the Hive tables and partitions in the cluster.[6]

Figure 4 depicts the system's MapReduce architecture. The JobTracker is responsible for controlling the parallel processing of the MapReduce functionality. The TaskTrackers act as instructed by the JobTracker to process the MapReduce jobs.[1]

Figure 4: The MapReduce Architecture



The Client Server

The client server (Node 9, not pictured) had Base SAS 9.3, Hive 0.8.0, a Hadoop 2.0.0 client, and a standard RDBMS installed. The SAS installation included Base SAS software and SAS/ACCESS products.

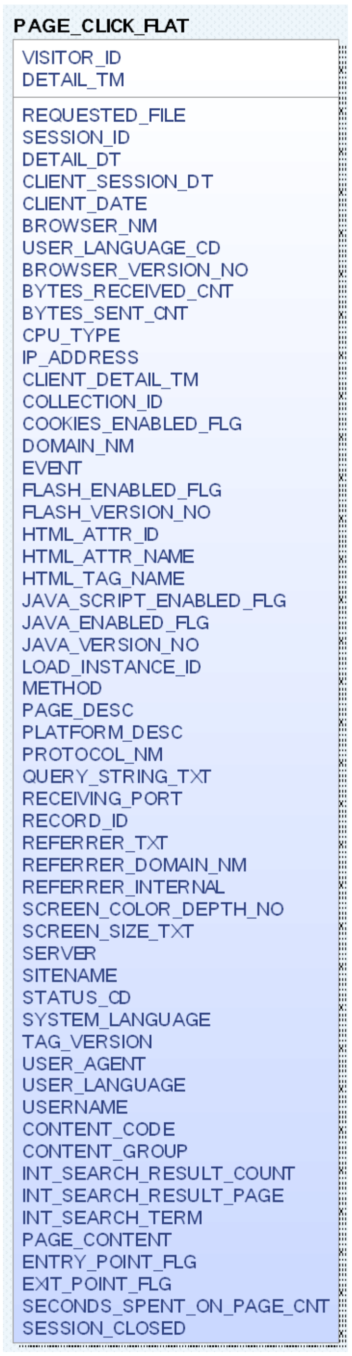
The RDBMS Server

A relational database was installed on Node 10 (not pictured) and was used for comparison purposes in our experiments.

Data Environment Setup

The data for our experiments was generated to resemble a technical company's support website. The company sells its products worldwide and uses Unicode to support foreign character sets. We created 25 million original weblog sessions featuring 90 million clicks, and then duplicated it 90 times by adding unique session identifiers to each row. This bulked-up flat file was loaded into the RDBMS and Hadoop via SAS/ACCESS and Sqoop. For our tests, we needed both a flat file representation of the data and a typical star schema design of the same data. Figure 5 shows the data in the flat file representation.

Figure 5: The Flat File Representation



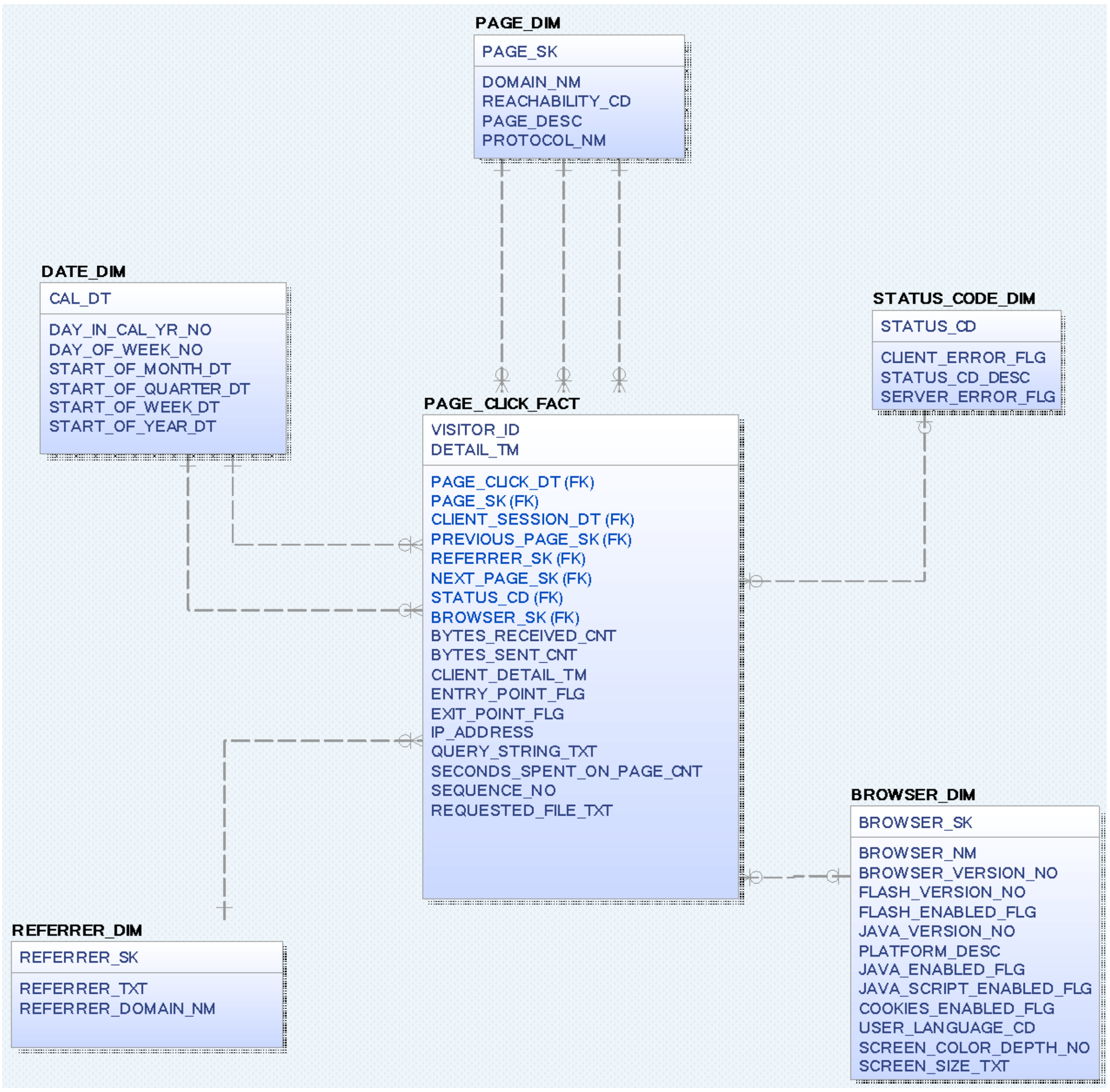
PAGE_CLICK_FLAT

VISITOR_ID
DETAIL_TM

REQUESTED_FILE
SESSION_ID
DETAIL_DT
CLIENT_SESSION_DT
CLIENT_DATE
BROWSER_NM
USER_LANGUAGE_CD
BROWSER_VERSION_NO
BYTES_RECEIVED_CNT
BYTES_SENT_CNT
CPU_TYPE
IP_ADDRESS
CLIENT_DETAIL_TM
COLLECTION_ID
COOKIES_ENABLED_FLG
DOMAIN_NM
EVENT
FLASH_ENABLED_FLG
FLASH_VERSION_NO
HTML_ATTR_ID
HTML_ATTR_NAME
HTML_TAG_NAME
JAVA_SCRIPT_ENABLED_FLG
JAVA_ENABLED_FLG
JAVA_VERSION_NO
LOAD_INSTANCE_ID
METHOD
PAGE_DESC
PLATFORM_DESC
PROTOCOL_NM
QUERY_STRING_TXT
RECEIVING_PORT
RECORD_ID
REFERRER_TXT
REFERRER_DOMAIN_NM
REFERRER_INTERNAL
SCREEN_COLOR_DEPTH_NO
SCREEN_SIZE_TXT
SERVER
SITENAME
STATUS_CD
SYSTEM_LANGUAGE
TAG_VERSION
USER_AGENT
USER_LANGUAGE
USERNAME
CONTENT_CODE
CONTENT_GROUP
INT_SEARCH_RESULT_COUNT
INT_SEARCH_RESULT_PAGE
INT_SEARCH_TERM
PAGE_CONTENT
ENTRY_POINT_FLG
EXIT_POINT_FLG
SECONDS_SPENT_ON_PAGE_CNT
SESSION_CLOSED

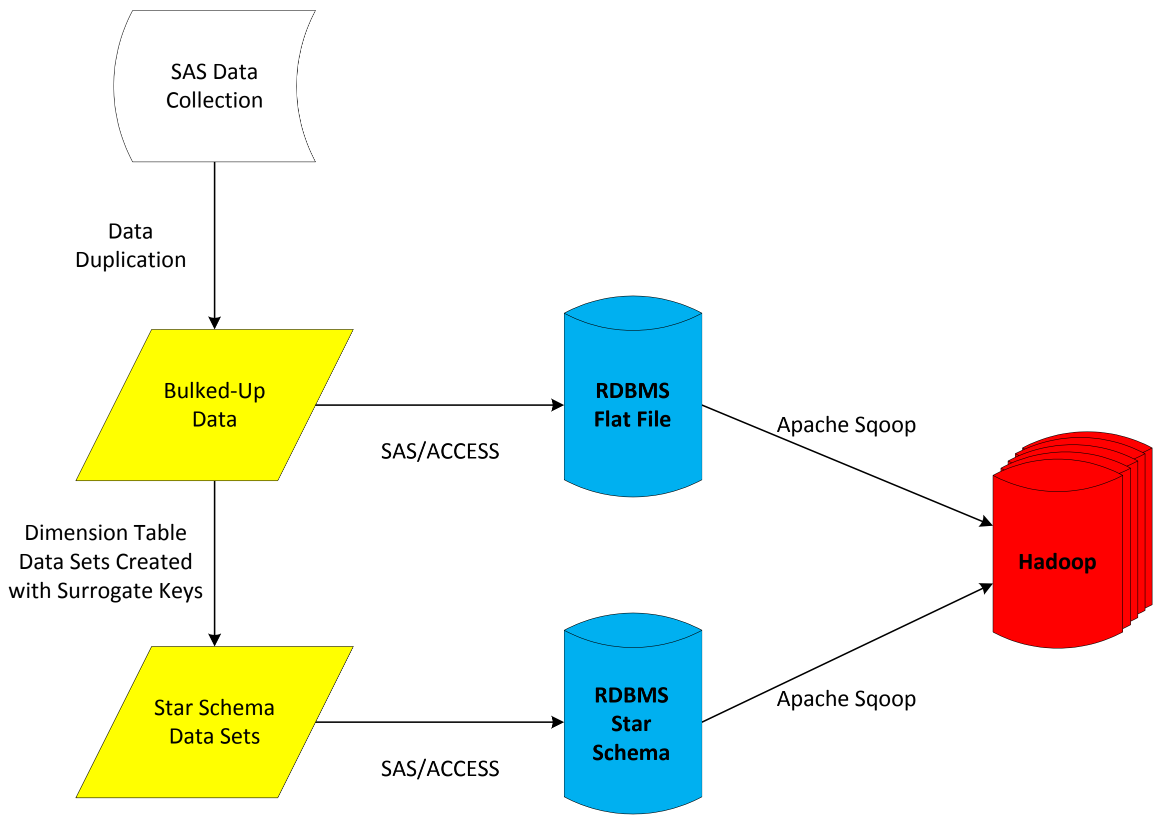
A star schema was created to emulate the standard data mart architecture. Its tables are depicted in Figure 6.

Figure 6: The Entity-Relationship Model for the Star Schema



To load the fact and dimension tables in the star schema, surrogate keys were generated and added to the flat file data in SAS before loading the star schema tables in the RDBMS and Hadoop. The dimension tables and the PAGE_CLICK_FACT table in the RDBMS were loaded directly through a SAS program and loaded directly into the RDBMS through the SAS/ACCESS engine. The surrogate keys from the dimension tables were added to the PAGE_CLICK_FACT table via SQL in the RDBMS. The star schema tables were loaded directly from the RDBMS into Hadoop using the Sqoop tool. The entire process for loading the data in both star schemas is illustrated in Figure 7.

Figure 7: The Data Load Process



As a side note, we uncovered a quirk that occurs when loading data from an RDBMS to HDFS or vice versa through Hive. Hive uses the Ctrl+A ASCII control character (also known as the start of heading or SOH control character in Unicode) as its default delimiter when creating a table. Our data had ^A sprinkled in the text fields. When we used the Hive default delimiter, Hive was not able to tell where a column started and ended due to the dirty data. All of our data loaded, but when we queried the data, we discovered the issue. To fix this, we redefined the delimiter. The takeaway is that you need to be data-aware before choosing delimiters to load data into Hadoop using the Sqoop utility.

Once the data was loaded, the number of rows in each table was observed as shown in Figure 8.

Figure 8: Table Row Numbers

Table Name	Rows
PAGE_CLICK _FACT	1.45 billion
PAGE_DIM	2.23 million
REFERRER_DIM	10.52 million
BROWSER_DIM	164.2 thousand
STATUS_CODE	70
PAGE_CLICK_FLAT	1.45 billion

In terms of the actual size of the data, we compared the size of the fact tables and flat tables in both the RDBMS and Hadoop environment. Because we performed tests in our experiments on both the text file version of the Hive tables as well as the compressed sequence file version, we measured the size of the compressed version of the tables. Figure 9 shows the resulting sizes of these tables.

Figure 9: Table Sizes

Table Name	RDBMS	Hadoop (Text File)	Hadoop (Compressed Sequence File)
PAGE_CLICK_FACT	573.18 GB	328.30 GB	42.28 GB
PAGE_CLICK_FLAT	1001.11 GB	991.47 GB	124.59 GB

Approach for Our Experiments

To test the various data modeling techniques, we wrote queries to simulate the typical types of questions business users might ask of clickstream data. The full SQL queries are available in Appendix A. Here are the questions that each query answers:

1. What are the most visited top-level directories on the customer support website for a given week and year?
2. What are the most visited pages that are referred from a Google search for a given month?
3. What are the most common search terms used on the customer support website for a given year?
4. What is the total number of visitors per page using the Safari browser?
5. How many visitors spend more than 10 seconds viewing each page for a given week and year?

As part of the criteria for the project, the SQL statements were used to determine the optimal structure for storing the clickstream data in Hadoop and in an RDBMS. We investigated techniques in Hive to improve the performance of the queries. The intent of these experiments was to investigate how traditional data modeling techniques apply to the Hadoop and Hive environment. We included an RDBMS only to measure the effect of tuning techniques within the Hadoop and Hive environment and to see how comparable techniques work in an RDBMS. It is important to note that there was no intent to compare the performance of the RDBMS to the Hadoop and Hive environment, and the results were for our particular hardware and software environment only. To determine the optimal design for our data architecture, we had the following criteria:

- There would be no unnecessary duplication of data. For example, we did not want to create two different flat files tuned for different queries.
- The data structures would be progressively tuned to get the best overall performance for the average of most of the queries, not just for a single query.

We began our experiments without indexes, partitions, or statistics in both schemas and in both environments. The intent of the first experiment was to determine whether a star schema or flat table performed better in Hive or in the RDBMS for our queries. During subsequent rounds of testing, we used compression and added indexes and partitions to tune the data structures. As a final test, we ran the same queries against our final data structures using Impala. Impala bypasses the MapReduce layer used by Hive.

The queries were run using Base SAS on the client node with an explicit SQL pass-through for both environments. All queries were run three times on a quiet environment to obtain accurate performance information. We captured the timings through the SAS logs for the Hive and RDBMS tests. Client session timing was captured in Impala for the Impala tests because SAS does not currently support Impala.

Results

Experiment 1: Flat File versus Star Schema

The intent of this first experiment was to determine whether the star schema or flat table structure performed better in each environment in a series of use cases. The tables in this first experiment did not have any tuning applied such as indexing. We used standard text files for the Hadoop tables.

Results for Experiment 1

Hadoop Flat File	Query	Min. Time (MM:SS)	Max. Time (MM:SS)	Average (MM:SS)
	1	51:42	52:13	52:00
	2	49:55	50:46	50:36
	3	54:53	56:36	55:54
	4	50:37	52:37	51:28
	5	49:43	50:25	50:00

Hadoop Star Schema	Query	Min. Time (H:MM:SS)	Max. Time (H:MM:SS)	Average (H:MM:SS)
	1	09:40	11:22	10:33
	2	09:08	09:57	09:35
	3	49:53	55:37	52:46
	4	13:04	15:14	14:33
	5	9:57	10:32	10:13

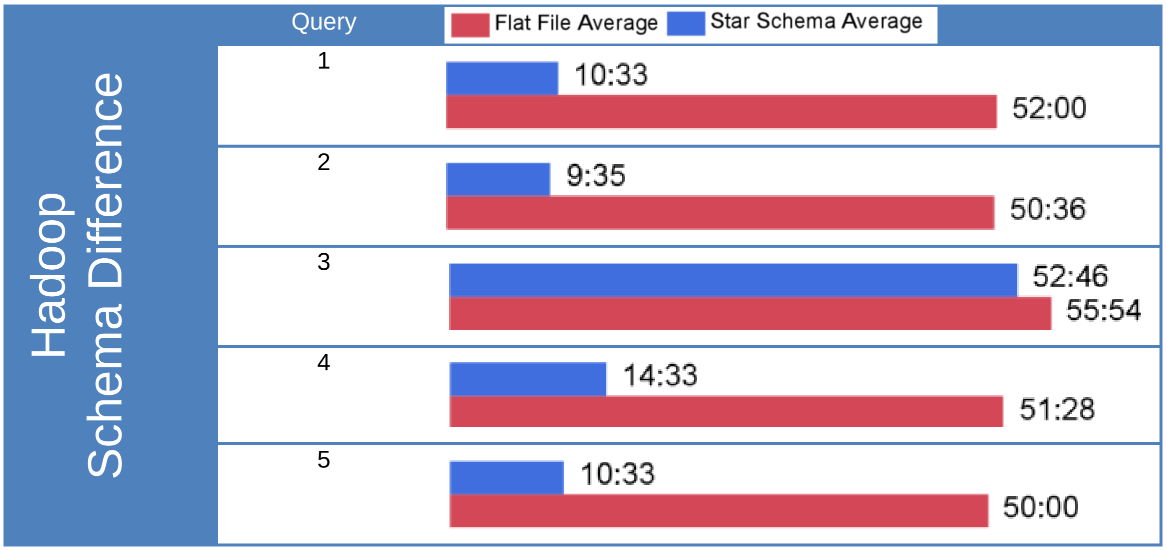
RDBMS Flat File	Query	Min. Time (H:MM:SS)	Max. Time (H:MM:SS)	Average (H:MM:SS)
	1	1:04:35	1:17:49	1:09:13
	2	1:09:26	1:09:52	1:09:35
	3	1:08:14	1:08:53	1:08:32
	4	1:06:22	1:07:44	1:07:06
	5	1:04:20	1:04:57	1:04:31

RDBMS Star Schema	Query	Min. Time (MM:SS)	Max. Time (MM:SS)	Average (MM:SS)
	1	33:03	33:41	33:26
	2	33:19	33:35	33:28
	3	33:28	34:27	34:02
	4	32:58	33:09	33:03
	5	33:00	33:56	33:35

Analysis of Experiment 1

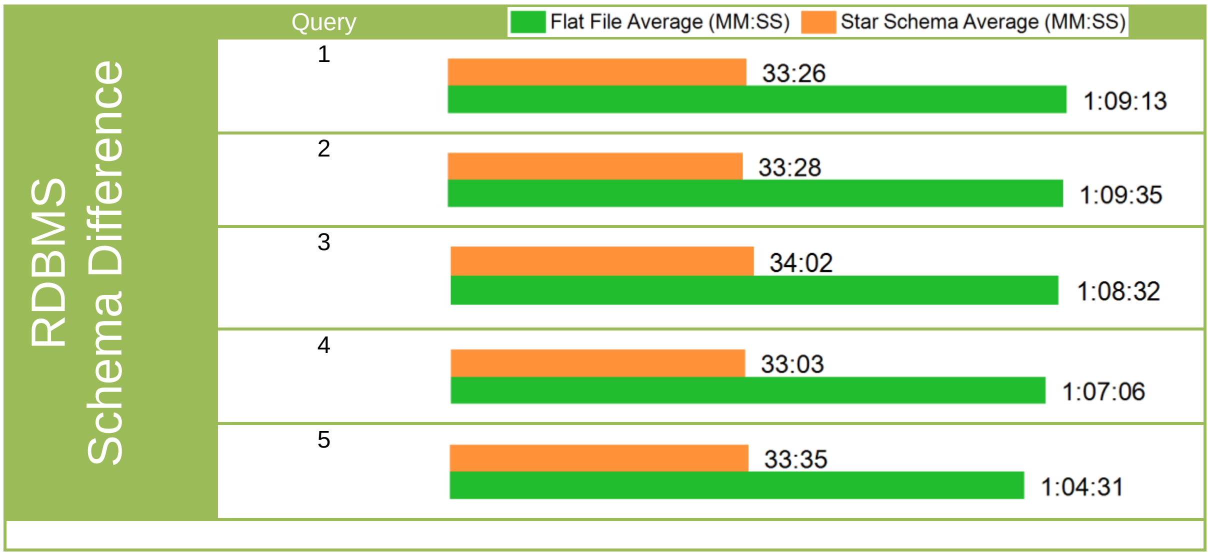
Hadoop Schema Difference

Query	Flat File Average (MM:SS)	Star Schema Average (H:MM:SS)	Improvement (Flat to Star)
1	52:00	10:33	42:27
2	50:36	09:35	41:01
3	55:54	52:46	03:08
4	51:28	14:33	36:55
5	50:00	10:33	39:27



RDBMS Schema Difference

Query	Flat File Average (H:MM:SS)	Star Schema Average (MM:SS)	Improvement (Star to Flat)
1	1:09:13	33:26	35:47
2	1:09:35	33:28	36:07
3	1:08:32	34:02	34:30
4	1:07:06	33:03	34:03
5	1:04:31	33:35	30:56



As you can see, both the Hive table and the RDBMS table in the star schema structure performed significantly faster compared to the flat file structure. This results for Hive were surprising, given the more efficient practice in HDFS of storing data in a denormalized structure to optimize I/O.

Although the star schema was faster in the Hadoop text file environment, we decided to complete the remaining experiments for Hadoop using the flat file structure because it is the more efficient data structure for Hadoop and Hive. The book *Programming Hive* says, “The primary reason to avoid normalization is to minimize disk seeks, such as those typically required to navigate foreign key relations. Denormalizing data permits it to be scanned from or written to large, contiguous sections of disk drives, which optimizes I/O performance. However, you pay the penalty of denormalization, data duplication and the greater risk of inconsistent data.”[8]

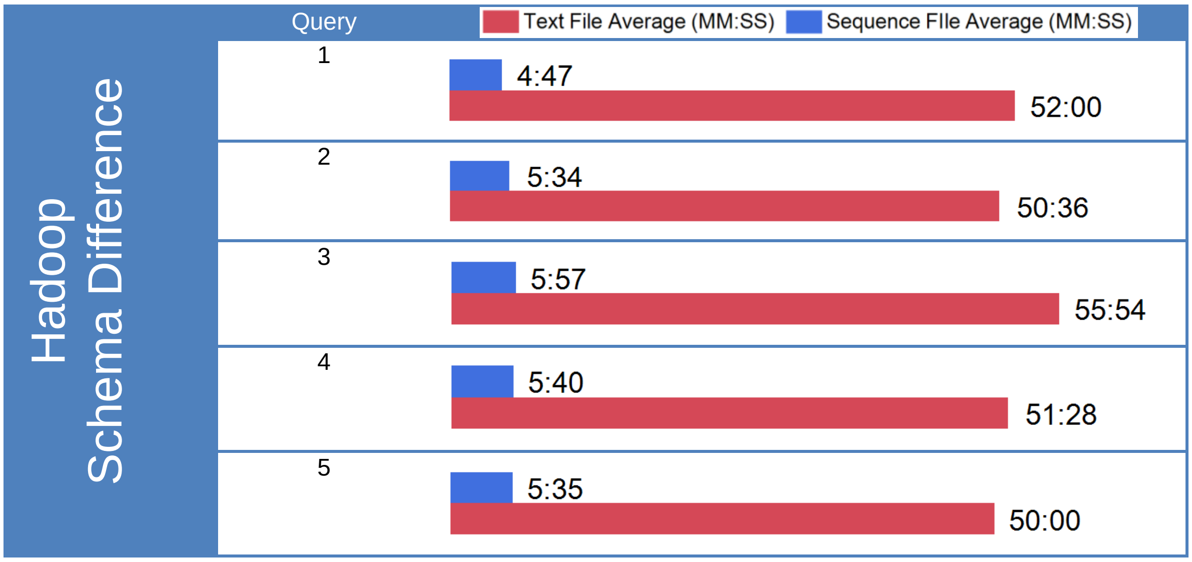
Experiment 2: Compressed Sequence Files

The second experiment applied only to the HIve environment. In this experiment, the data in HDFS was converted from uncompressed text files to compressed sequence files to determine whether the type of file for the table in HDFS made a difference in query performance.

Results for Experiment 2

Hadoop Sequen ce File	Query	Min. Time (MM:SS)	Max Time (MM:SS)	Average (MM:SS)
	1	04:44	04:48	04:47
	2	05:27	04:41	05:34
	3	05:51	06:04	05:57
	4	05:35	05:47	05:40
	5	05:30	05:40	05:35

Hadoop Schema Difference	Query	Text File Average (MM:SS)	Sequence File Average (H:MM:SS)	Improvement (Text to Sequence)
	1	52:00	04:47	47:13
	2	50:36	05:34	45:02
	3	55:54	05:57	49:57
	4	51:28	05:40	45:48
	5	50:00	05:35	44:25



The results of this experiment clearly show that the compressed sequence file was a much better file format for our queries than the uncompressed text file.

Experiment 3: Indexes

In this experiment, indexes were applied to the appropriate columns in the Hive flat table and in the RDBMS fact table. Statistics were gathered for the fourth set of tests. In Hive, a B-tree index was added to each of the six columns (BROWSER_NM, DETAIL_TM, DOMAIN_NM, FLASH_ENABLED_FLG, QUERY_STRING_TXT, and REFERRER_DOMAIN_NM) used in the queries. In the RDBMS, a bitmap index was added to each foreign key in the PAGE_CLICK_FACT table, and a B-tree index was added to each of the five columns (DOMAIN_NM, FLASH_ENABLED_FLG, REFERRER_DOMAIN_NM, QUERY_STRING_TXT, and SECONDS_SPENT_ON_PAGE_CNT) used in the queries that were not already indexed.

Results for Experiment 3

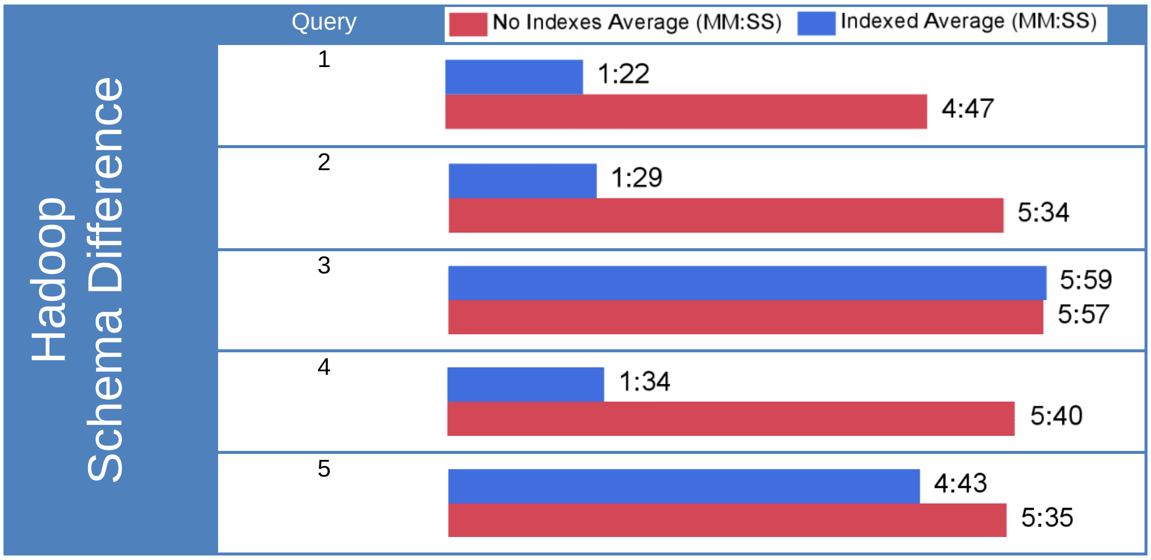
Hadoop Flat File	Query	Min. Time (MM:SS)	Max Time (MM:SS)	Average (MM:SS)
	1	01:17	01:28	01:22
	2	01:25	01:33	01:29
	3	05:55	06:03	05:59
	4	01:32	01:37	01:34
	5	04:42	04:45	04:43

RDBMS Star Schema

Query	Min. Time (MM:SS)	Max Time (MM:SS)	Average (MM:SS)
1	00:04	00:04	00:04
2	00:25	01:01	00:39
3	00:25	00:43	00:31
4	00:07	00:07	00:07
5	00:25	00:31	00:27

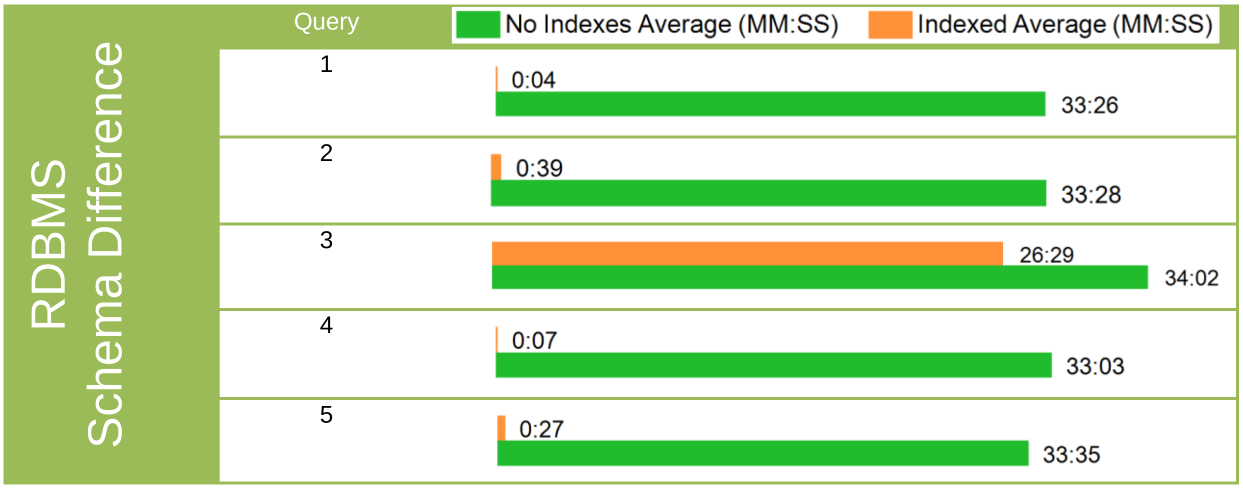
Hadoop Schema Difference

Query	No Indexes Average (MM:SS)	Indexed Average (MM:SS)	Improvement (No Indexes to Indexed)
1	04:47	01:22	03:25
2	05:34	01:29	04:05
3	05:57	05:59	(00:02)
4	05:40	01:34	04:06
5	05:35	04:43	00:52



RDBMS Schema Difference

Query	No Indexes Average (H:MM:SS)	Indexed Average (MM:SS)	Improvement (No Indexes to Indexed)
1	33:26	00:04	33:22
2	33:28	00:39	32:49
3	34:02	26:29	07:33
4	33:03	00:07	32:56
5	33:35	00:27	33:08



Analysis of Experiment 3:

With the notable exception of the third query in the Hadoop environment, adding indexes provided a significant increase in performance across all of the queries.

Experiment 4: Partitioning

In experiment 4, we added partitioning to the DETAIL_DT column in both the flat table in Hive and in the fact table in the star schema in the RDBMS. A partition was created for every date value.

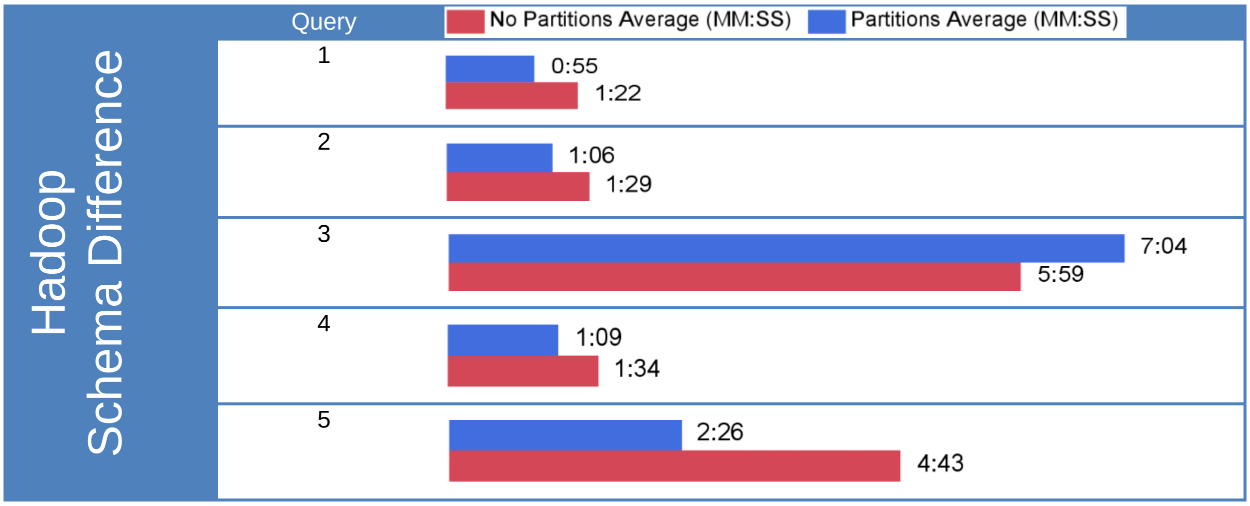
Results for Experiment 4

Hadoop Flat File	Query	Min. Time (MM:SS)	Max Time (MM:SS)	Average (MM:SS)
	1	00:50	01:00	00:55
	2	01:04	01:10	01:06
	3	06:42	07:41	07:04
	4	01:07	01:13	01:09
	5	02:25	02:28	02:26

RDBMS Star Schema	Query	Min. Time (MM:SS)	Max Time (MM:SS)	Average (MM:SS)
	1	00:01	00:03	00:02
	2	00:02	00:06	00:04
	3	39:33	45:40	45:32
	4	00:02	00:46	00:17
	5	00:01	00:03	00:02

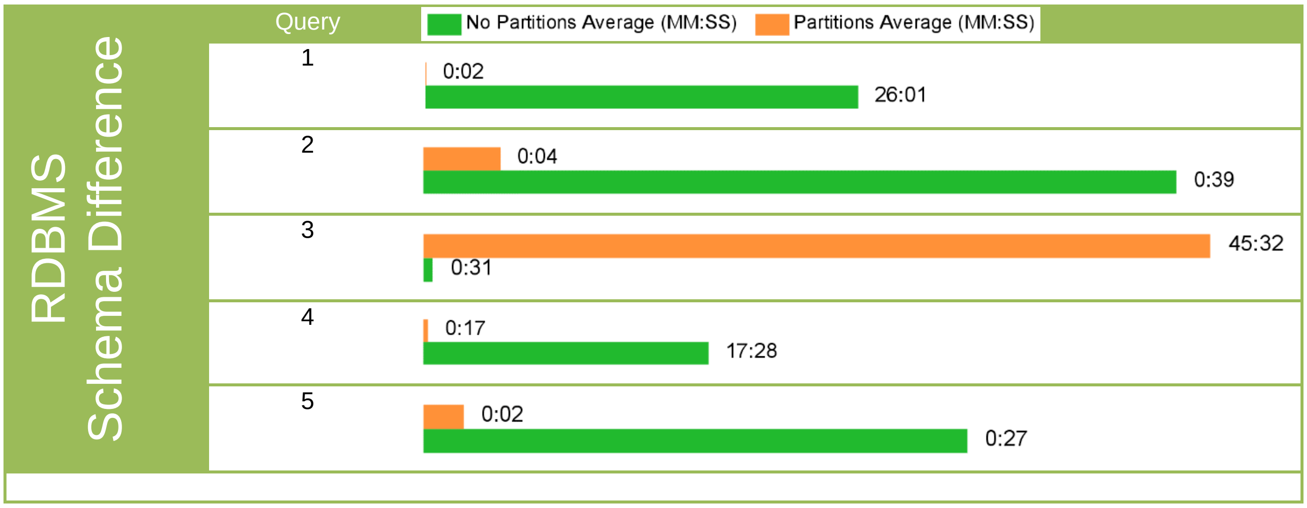
Hadoop Schema Difference

Query	No Partitions Average (MM:SS)	Partitioned Average (H:MM:SS)	Improvement (No Partitions to Partitioned)
1	01:22	00:55	00:27
2	01:29	01:06	00:23
3	05:59	07:04	(01:05)
4	01:34	01:09	00:25
5	04:43	02:26	02:17



RDBMS Schema Difference

Query	No Partitions Average (MM:SS)	Partitioned Average (MM:SS)	Improvement (No Partitions to Partitioned)
1	26:01	00:02	25:59
2	00:39	00:04	00:35
3	00:31	45:32	(45:01)
4	17:28	00:17	17:11
5	00:27	00:02	00:25



Charts for queries 2, 3, and 5 have been rescaled

Partitioning significantly improved all queries except for the third query. Query 3 was slightly slower in Hive and significantly slower in the RDBMS.

Experiment 5: Impala

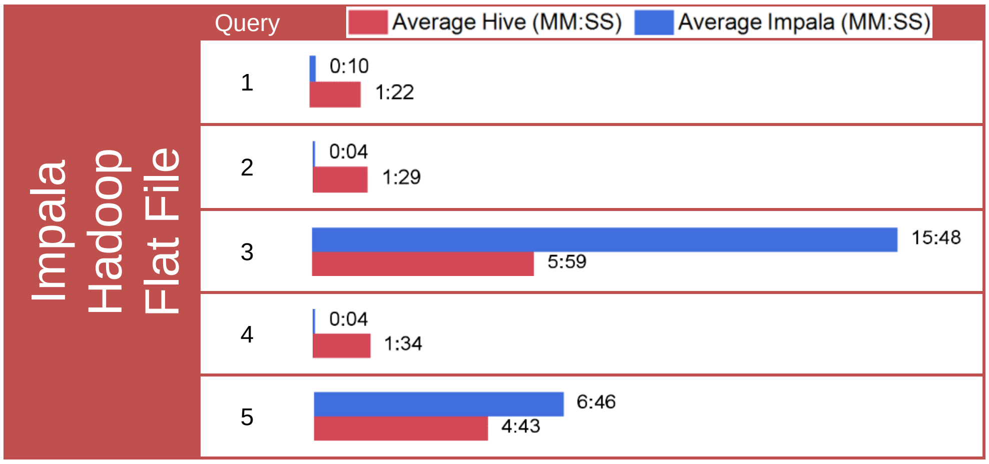
In experiment 5, we ran the queries using Impala on the Hive compressed sequence file table with compression and indexes. Impala bypasses MapReduce to use its own distributed query access engine.

Results for Experiment 5

Impala Hadoop Flat File	Query	Time (MM:SS)
	1	00:10
	2	00:04
	3	15:48
	4	00:04
	5	06:46

Analysis of Experiment 5

Impala Hadoop Flat File	Query	Average Hive (MM:SS)	Impala Time (MM:SS)	Improvement (Hive to Impala)
	1	01:22	00:10	01:12
	2	01:29	00:04	01:25
	3	05:59	15:48	(09:49)
	4	01:34	00:04	01:30
	5	04:43	06:46	(02:03)



The results for Impala were mixed. Three queries ran significantly faster, and two queries ran longer.

Interpretation of Results

The results of the first experiment were surprising. When we began the tests, we fully expected that the flat file structure would perform better than the star schema structure in the Hadoop and Hive environment. In the following table, we provide information that helps explain the differences in the amounts of time processing the queries. For example, the amount of memory required is significantly higher in the flat table structure for the query. Moreover, the number of mappers and reducers needed to run the query was significantly higher for the flat table structure. Altering system settings, such as TaskTracker heap sizes, showed benefits in the denormalized table structure. However, the goal of the experiment was to work with the default system settings in Cloudera Hadoop 4.2 and investigate the effects of structural changes on the data.

Unique Visitors per Page for Safari

	DENORMALIZED	*NORMALIZED*	*DIFF*
Virtual Memory (GB)	7,452	2,927	4,525
Heap (GB)	3,912	1,512	2,400
Read (GB)	507	329	178
Table Size (GB)	1,002	328	674
Execution Plan	3967 maps/999 reduce	1279 maps/352 reduce	
Time (minutes)	42	14	28

Our second experiment showed the performance increase that emerged from transitioning from text files to sequence files in Hive. This performance improvement was expected. However, the magnitude of the improvement was not. The queries ran about ten times faster when the data was stored in compressed sequential files than when the data was stored in uncompressed text files. The compressed sequence file optimizes disk space usage and I/O bandwidth performance by using binary encoding and splittable compression. This proved to be the single biggest factor with regard to data structures in Hive. For this experiment, we used block compression with SnappyCodec.

In our third experiment, we added indexes to the fact table in the RDBMS and to the flat table in Hive. As expected, the indexes generally improved the performance of the queries. The one exception was the third query, where adding the indexes did not show any improvement in Hive. The Hive Explain Plan helps explain why this is happening. In the highlighted section of the Hive Explain Plan, we see that there are no indexes used in the predicate of the query. Given the characteristics of the data, this makes sense because almost all of the values of DOMAIN_NM were the support site itself. The referring domain was predominantly www.google.com.

Hive Explain Plan:

```
STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-2 depends on stages: Stage-1
  Stage-3 depends on stages: Stage-2
  Stage-0 is a root stage
STAGE PLANS:
  Stage: Stage-1
    Map Reduce
      Alias -> Map Operator Tree:
        visits_pp_ph_summary:page_click_flat_seq_ns
          TableScan
            alias: page_click_flat_seq_ns
            Filter Operator
              predicate:
                  expr: ((domain_nm = ‘support.foo.com') and   (referrer_domain_nm = 'www.google.com'))
                  type: boolean
```

Experiment 4 added partitioning to the table structures. Both the fact table and the flat table were partitioned by date in the DETAIL_DT column. Partitioning tables changes how Hive structures the data storage. In addition to the directory for each table, Hive creates subdirectories reflecting the partitioning structure. When a query is executed on Hive, the query does not need to scan the entire directory for the query. Rather, partition elimination enables the query to go directly to the subdirectory or subdirectories where that data is located to retrieve the results. Because many of our queries used DETAIL_DT in the WHERE clause of the query, execution time improved. The same improvement was seen in the RDBMS, which was able to use partition elimination for most of the queries. In the case of the third query, the predicate does not include DETAIL_DT. In this case, having partitions actually hurt query performance because the query needed to examine each partition individually to locate the relevant rows. The decrease in performance was significant in the RDBMS.

Experiment 5 explored Impala and gauged the performance. Impala is a query engine that provides more SQL functionality in the Hive environment. Impala does not use MapReduce to process the data. Rather, it provides direct access to the data in HDFS through its own proprietary engine.

Overall, three of the queries ran significantly faster in Impala. Two of the queries were worse in Impala. Interestingly, in our query for the top referrers, we needed to add a LIMIT clause following the ORDER BY clause because this is currently a requirement for Impala queries. Similar to the issue in query 3 in the Hive environment, query 3 was slow because a full table scan of all of the partitions was required to retrieve the data.

Conclusions

Through our experiments, we have shown that structuring data properly in Hive is as important as in an RDBMS. The decision to store data in a sequence file format alone accounted for a performance improvement of more than 1,000%. The judicious use of indexes and partitions resulted in significant performance gains by reducing the amount of data processed.

For data architects working in the Hive environment, the good news is that many of the same techniques such as indexing that are used in a traditional RDBMS environment are applicable. For those of us who are familiar with MPP databases, the concept of partitioned data across nodes is very familiar.

The key takeaway is that we need to understand our data and the underlying technology in Hadoop to effectively tune our data structures. Simply creating a flat table or star schema does not result in optimized structures. We need to understand how our data is distributed, and we need to create data structures that work well for the access patterns of our environment. Being able to decipher MapReduce job logs as well as run explain plans are key skills to effectively model data in Hive. We need to be aware that tuning for some queries might have an adverse impact on other queries as we saw with partitioning.

Future experimentation should look into the performance enhancements offered with other Hive file formats, such as RCFile, which organizes data by column rather than by row. Another data modeling test could examine how well collection data types in Hive work compared to traditional data types for storing data. As big data technology continues to advance, the features that are available for structuring data will continue to improve, and further options for improving data structures will become available.

Appendix

Queries Used in Testing Flat Tables

```
1. select top_directory, count(*) as unique_visits
     from (select distinct visitor_id,
                  split(requested_file, '[\\/]')[1] as top_directory
             from page_click_flat
            where domain_nm = 'support.sas.com' and
                  flash_enabled='1' and
                  weekofyear(detail_tm) = 48 and
                  year(detail_tm) = 2012
          ) directory_summary
   group by top_directory
   order by unique_visits;
2. select domain_nm, requested_file, count(*) as unique_visitors, month
     from (select distinct domain_nm, requested_file, visitor_id,
                  month(detail_tm) as month
             from page_click_flat
            where domain_nm = 'support.sas.com' and
                  referrer_domain_nm = 'www.google.com'
          ) visits_pp_ph_summary
   group by domain_nm, requested_file, month
   order by domain_nm, requested_file, unique_visitors desc, month asc;
3. select query_string_txt, count(*) as count
     from page_click_flat
    where query_string_txt <> '' and
          domain_nm='support.sas.com' and
          year(detail_tm) = '2012'
   group by query_string_txt
   order by count desc;
4. select domain_nm, requested_file, count(*) as unique_visitors
    from (select distinct domain_nm, requested_file, visitor_id
            from page_click_flat
           where domain_nm='support.sas.com' and
                 browser_nm like '%Safari%' and
                 weekofyear(detail_tm) = 48 and
                 year(detail_tm) = 2012
         ) uv_summary
   group by domain_nm, requested_file
   order by unique_visitors desc;
5. select domain_nm, requested_file, count(*) as unique_visits
     from (select distinct domain_nm, requested_file, visitor_id
             from page_click_flat
            where domain_nm='support.sas.com' and
                  weekofyear(detail_tm) = 48 and
                  year(detail_tm) = 2012 and
                  seconds_spent_on_page_cnt > 10;
          ) visits_summary
   group by domain_nm, requested_file
   order by unique_visits desc;
```

References

[1] "Understanding Hadoop Clusters and the Network." Available at http://bradhedlund.com. Accessed on June 1, 2013.

[2] Sammer, E. 2012. *Hadoop Operations*. Sebastopol, CA: O'Reilly Media.

[3] "HDFS High Availability Using the Quorum Journal Manager." Apache Software Foundation. Available at http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailabilityWithQJM.html. Accessed on June 5, 2013.

[4] "Hadoop HDFS over HTTP - Documentation Sets 2.0.4-alpha." Apache Software Foundation. Available at http://hadoop.apache.org/docs/r2.0.4-alpha/hadoop-hdfs-httpfs/index.html. Accessed on June 5, 2013.

[5] "Yahoo! Hadoop Tutorial." Yahoo! Developer Network. Available at http://developer.yahoo.com/hadoop/tutorial/. Accessed on June 4, 2013.

[6] "Configuring the Hive Metastore." Cloudera, Inc. Available at http://www.cloudera.com/content/cloudera-content/cloudera-docs/CDH4/4.2.0/CDH4-Installation-Guide/cdh4ig_topic_18_4.html. Accessed on June 18, 2013.

[7] Kestelyn, J. "Introducing Parquet: Efficient Columnar Storage for Apache Hadoop." Available at http://blog.cloudera.com/blog/2013/03/introducing-parquet-columnar-storage-for-apache-hadoop/. Accessed on August 2, 2013.

[8] Capriolo, E., D. Wampler, and J. Rutherglen. 2012. *Programming Hive*. Sebastopol, CA: O'Reilly Media.