UPPSALA
UNIVERSITET

IT 14 074
Examensarbete 15 hp
December 2014

Analyzing the impact of data compression in Hive

Niklas Andersen

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Analyzing the impact of data compression in Hive

Niklas Andersen

Executing expensive queries over many large tables can be prohibitively time consuming in conventional relational databases. Hadoop and its data warehouse Hive is a powerful alternative for large scale data processing.

Conventionally, data is stored in Hive without compression. There is value in storing the data with compression, if the overhead of compression does not negatively impact the query processing time.

This paper describes through experiments using imports, transformations and exports of Hive data in various file formats and with different compression techniques how this can be achieved.

Handledare: Erik Zeitler
Ämnesgranskare: Silvia Stefanova
Examinator: Olle Gällmo
IT 14 074
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

With 90% of all data in the world generated over the last two years[1], relational database management systems have had difficulties in keeping up with data of high volume, velocity and variety.

Hadoop[2] has an approach to handle complex processing of large scale data. Hive[3] is a data warehouse used for querying and analyzing data stored in Hadoops distributed file system (HDFS[4]). Running complex queries with Hive can be demanding both on time and disk space consumption. The purpose of this project is to understand to what extent data compression could alleviate disk space usage during Hive queries in Hadoop, and also to consider the trade-offs in execution time.

Hive is capable of working with a variety of files including textfiles, sequencefiles (Binary key-value pairs), RC-files (Row Columnar files), ORC-files (Optimized Row Columnar files) and Parquet (another columnar-storage format). Hive allows various file compression schemes to be applied on each of these file formats, and this report aims to explore possible time and disk space effective combinations of file formats and compression techniques.

To answer the questions at hand, a series of experiments using different data set sizes, file formats and compression algorithms were performed.
First, data was imported from a relational database (RDBMS[5]) to HDFS. Time taken to import the data was measured, as well as the disk space consumption needed to store this data in HDFS. Second, execution time for running a set of queries on this data in Hive was measured. The results of these queries were stored as new Hive tables in HDFS, whose disk space consumption was also measured.
Third, the time taken to export the results of the queries to another RDBMS was measured. The importing and exporting of data was important for understanding the overhead in time of compressing and decompressing data in various file formats.

Chapter 1 of this report describes the outlines of this project, while chapter 2 is meant to give readers a quick overview of central database concepts and Hadoop tools used in this project. Chapter 3 describes the experiments carried out, as well as their results. Chapter 4 finally sums up the conclusions drawn from these experiments, and chapter 5 looks at other contributions on the subject and how they relate to this thesis, as well as what work could be conducted in the future.

2 Background

To get an understanding of the problem at hand, as well as the main tools and concepts used throughout this report, this chapter describes what Klarna is and how their flow of relational data is handled with Hadoop. It explains the file formats and compression algorithms used, and it also explains and shows the relevance of TPC-H to this report.

2.1 Klarna

Klarna offers payment solutions for e-commerce where the customer receives the order first and pays the invoice afterwards. To provide relevant informationfor customer credit assessment, Klarna regularly executes various data pipelines. One of these data pipelines is called the Risk Realtime Database (RRDB) transformations, which is executed using Hive. The job imports data from RDBMS's into Hive, where it performs queries on the data before exporting the query results to the RRDB (which is also an RDBMS). These queries require a lot of disk space, and it is useful to understand to what extent compression can alleviate disk consumption without negatively affecting the performance of the queries.

2.2 RDBMS

A relational database management system is a model for storing and processing data. This model is built on first-order predicate logic, meaning that data is represented as records (rows, tuples) and grouped into relations.

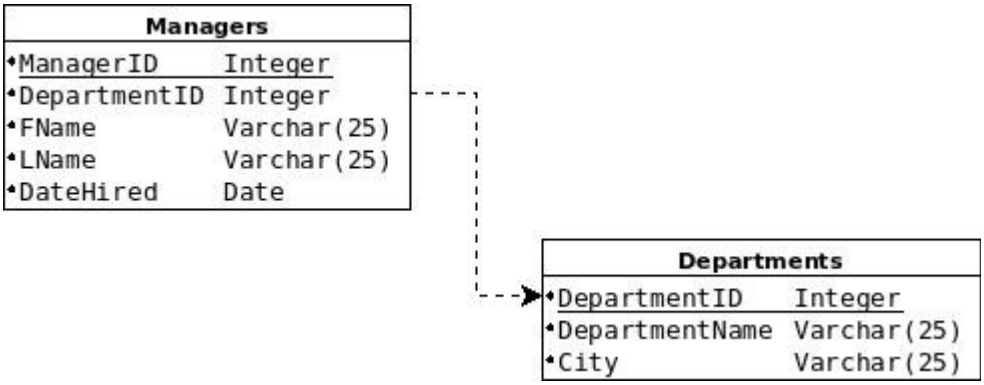

Figure 1: Two tables with primary keys underlined and Mangers.DepartmentID being a foreign key referring to Departments.DepartmentID

A table in an RDBMS is a collection of data organized in terms of columns and records, with each table having a primary key functioning as a unique identifier for any record. Tables are then related to one another through foreign keys pointing from one field of a record to the primary key of another table (see figure 1).

This model enables a declarative method for quickly accessing and safely inserting, deleting and updating data, and the most commonly used language for doing so today is SQL (Structured Query Language).

The RDBMS model implements the ACID[7] concept, meaning that it provides *atomicity* (each transaction (modification of data) is either carried out in full, or not at all), *consistency* (all data has its properties and data types checked beforehand so that only valid data can be written into the database), *isolation* (guarantees to not introduce race conditions) and *durability* (transactions committed to the database will not be lost). A precondition for this model is that the data is structured to fit into the predetermined tables where each column has a specific data type (integer, varchar etc). If the data does not meet these criteria it is

rejected before being written to disk, and this is known as *schema on write*.

Scaling up an RDBMS can be achieved through either vertical scaling, meaning that the server machine is either upgraded (RAM, disk space, number of cores etc) or replaced, or horizontal scaling, meaning that data is partitioned, preferably by some evenly distributed column, and spread out over multiple servers (for example, a phone book taking all names beginning with A-M and storing it on one server, and taking all names beginning with N-Z and storing it on another one). Vertical scaling usually equals high costs and system downtime when upgrading, and once the threshold for what a machine can manage in terms of resources has been reached, it simply can't vertically scale further. Horizontal scaling means that relations might be severed and has to be reconstructed at the client side, taking away much of the purpose of an RDBMS in the first place. Also, the consistency demand on an RDBMS means that an update on any node must be replicated to all other nodes or not enforced at all, demanding additional overhead and locking of tables. For example: an update query using multiple JOINs on tables distributed across a number of servers might have to put a lock on all tables used until the update is complete. For complex and/or frequent queries, this can have a significant impact on availability.

2.3 Hadoop

Apache Hadoop is an open-source system for storage and processing of data. Hadoop was designed to meet the new needs of big data and is suited for storing and processing large unstructured or semi-structured files of data. Where the RDBMS model fits well with data that is often updated, Hadoop fits better with data that is stored once and read multiple times as it does not support transactions. The two main components of Hadoop are the distributed file system (HDFS) and the MapReduce processing framework, which the following paragraphs introduces.

HDFS, as the name suggests, is a distributed file system aiming to be fault-tolerant and implemented on commodity hardware with a focus on scalability and high throughput rather than low latency. To reduce network congestion, computations are mostly done locally on the machine storing the actual data, instead of first transferring the data to a specific compute-node.
HDFS relies on a master-slave model with a cluster of nodes having one master (the NameNode) and multiple slaves (DataNodes). Imports are split up and stored in blocks as flat files on the DataNodes. As compared to RDBMSs, data is not stored in tables, and no updates are available. It is not until the data is read (for example with Hive) that any schema is applied to it. This is known as *schema on read* and has the effect that it is easy to load data into the file system. Imports to the HDFS are done directly from data sources to the DataNodes, and the book keeping of how all files and their respective blocks are stored is automatically done by the NameNode. With the files stored in HDFS often being of large sizes, excessive book keeping is avoided. If many small files are used, then a container format (like sequencefiles) can be used to group files together. Each node uses periodical heartbeats to inform the NameNode that it is functioning properly. Each node has a number of backups (current default replication factor is 3) so that no data is lost if a data node fails. To scale up HDFS horizontally, more data nodes are added to the cluster.

The MapReduce model is well suited for processing large datasets in parallel. It consists of a *map* phase which processes, sorts and filters datasets, and a *reduce* phase which summarizes the results of the Map phase.
The '*Hello world!'* of Hadoop is a simple word count program. The MapReduce framework would take one or multiple text files as input. For example, imagine two text files containing one sentence each being processed with MapReduce:

Textfile 1:
"word count example"

Textfile 2:
“let’s count every word”

Each map task would take a share of text and output a key/value pair for each word within the text, where the key would be the word and the value would be 1. Assume in this case that we use two mappers (one for each file), which would give the output:
<word, 1>
<count, 1>
<example, 1>

<let’s, 1>
<count, 1>
<every, 1>
<word, 1>

These key/value pairs would then be passed on to reducers, which sum up the result for each word, which in turn could give a final output like this:

<word, 2>
<example, 1>
<count, 2>
<let’s, 1>
<every, 1>

When multiple reducers are used, the output from the mappers are combined so that all occurrences of the same word arrive at the same reducer.

2.4 Hive

To avoid having end users having to learn MapReduce programming in order to access and analyze the data in HDFS, Hive was introduced to the Hadoop framework in 2007. Hive is a data warehouse infrastructure providing a query language, Hive QL (a subset of the simple and well established SQL), enabling well known database structures like tables, rows and columns, and data types like integers, strings and structs.

When a query is executed in Hive QL, the Hive compiler parses the query, type checks and performs semantic analysis and optimization before translating the query into a directed acyclic graph of MapReduce jobs (with the exception of meta data and raw data requests). As queries are transformed to fit the MapReduce framework, simple queries like ‘SELECT COUNT(*)’ from a small table takes a lot longer to complete than they would on an RDBMS, but when long running queries are executed, this overhead becomes negligible.

2.5 File formats

Hadoop is capable of working with a variety of file formats, and the following formats are the ones used in the experiments of this thesis.
Hadoop stores, if not otherwise specified, data as **textfiles** in HDFS.
Hadoop is designed for large files, and **sequencefiles** can be used as a container for merging multiple small files into a larger sequencefile, thereby reducing NameNode overhead of keeping track of meta data for each small file. Sequencefiles are flat files of key-value pairs stored in binary format, and is frequently used in Hadoop with temporary map task outputs using this format.
An **RC-file** (row-columnar file) is designed for using the MapReduce framework. A table is first partitioned horizontally before storing the resulting tables by column instead of by row.

This is suitable for tables where values of the same column are often similar or repeated and/or when queries are interested only in one or a small subset of the columns of a table. Compression has a high impact on disk space when values are similar, and queries interested in aggregate functions or only a small portion of a tables columns, like for example ‘SELECT AVERAGE(salary) FROM employee’, would speed up execution time significantly since instead of going through each row of the employee table only to discard all unwanted columns, only one single column's worth of data (the 'salary' column) would have to be read from disk.

2.6 Sqoop

Sqoop[6] is a tool used for transferring data between RDBMSs and Hadoop. Imported data is stored in HDFS (with the option for data in text format to be directly imported or exported to and from Hive).

When an import of a table is executed with Sqoop, a MapReduce job is executed to fetch the maximum and minimum value of the primary key column (if not otherwise specified) of the table. This is to be able to compute how to fairly evenly split the table up horizontally before transferring the data in parallel into HDFS. Much in the same fashion, exports from Hive to an RDBMS are also carried out with parallel reads from HDFS before inserting the rows into the destination database.

When importing a table in a format other than text, the imported table will be a Java-object where each column is a separate attribute. Therefore, when importing tables as for example sequence- or RC-files with Sqoop, the class files have to be kept and handled manually in order for Hive to be able to interpret the data. HCatalog described next solves the problem of manually handling the class files for interpreting these file formats.

2.7 HCatalog

HCatalog is an abstraction layer that makes the Hive MetaStore available to other parts of Hadoop (like MapReduce and Pig), enabling Hadoop users to work with the tables in Hive without having to know the availability, place or format of the data[8].

2.8 Compression

There are three different levels of compression (apart from the individual settings of for example the gzip codec). Either no compression at all, RECORD-level compression where each row of a table is compressed separately, and BLOCK-level compression where each HDFS-block is compressed separately. BLOCK-level compression gives a better compression rate (as the codec dictionary is able to grow larger), while RECORD-level compression on text files are splittable and thus can be read by multiple mappers at once. In the experiments we used two different kinds of compression algorithms,one which aims to be fast (Snappy) and one which aims more to compress well (gzip).
Snappy (earlier named Zippy) is an open source lossless codec library. The focus of the algorithm is to be quick, with the trade-off that the compression ratio is lower than many other compression libraries. The Snappy version used by Hadoop is a little different than original Snappy, as Hadoop uses more meta data which can not be read by the original version. When used with the Hadoop framework, it is recommended to use Snappy-compression on container formats (like sequencefiles), since Snappy-compressed textfiles using block-compression are not splittable, and therefore can not be processed in parallel.
gzip (GNU zip) uses the DEFLATE algorithm. As with Snappy-compression, gzip-compressed textfiles with block-compression are not splittable, and thus gzip should be used on container formats. The speed of the compression process can be altered with the flags -1 thru -9, where -1 means optimize for speed and -9 means optimize for compression. Throughout this thesis, the default level (-6) was used.

MapReduce decides by looking at a file extension whether or not to decompress a file when reading it from HDFS.

2.9 TPC-H

The TPC Benchmark H is a well known and understood ad-hoc/decision support benchmark from the Transaction Processing Performance Council, and so there is a general interest in using TPC-H when comparing different databases. It features an RDBMS schema, a database generator for populating this schema, and a suite of queries used for measuring database performance. The schema and queries are and designed to be business-oriented, easy to implement and to offer a high degree of complexity.

3 Processing and measurements

To examine the impact of compression in Hive, several experiments that simulate the process of the Klarna RRDB transformations were conducted. While working on the development cluster at Klarna, production data was not possible to use. To synthesize production like data would be a highly complex and time consuming task.

In the first experiment, disk space and time consumption was measured using Snappy-compression. A devised schema with complex queries (multiple joins) was used and large enough result sets were produced which are comparable to Klarna RRDB transformations. In the second experiment, Snappy-compression was used with schema and queries from the TPC-H Benchmark to make to make the experiments easily reproducible.
Lastly, snappy compression was compared with gzip compression using the TPC-H schema.

Text files were used in the experiments as this is the default file format used with Hadoop, moreover they serve well as a format to compare the other formats against.
The reason for using the sequencefile format is that text files when used with block-compression are not splittable, while block-compressed sequencefiles are.
RC-files were chosen as this is a format designed for tabular data in Hadoop. The reason for not using ORCfiles (Optimized Row Columnar files, a newer version of RC-files which supports indexes among other things) is that the current Hive version (0.11) deployed at the Klarna development cluster did not support this format.

Hadoop compresses data at record level or block level. In record level each record is compressed independently, where as in block level each HDFS block is compressed together. If a file size grows beyond a certain limit, the compression ratio does not show any meaningful improvement beyond this size. Thus, for the purpose of these experiments there were no need to scale the source data set sizes beyond 10GB.

Snappy compression on text files were introduced later on in the experiment, and exports were not possible to execute until some time into the project, and therefore these numbers are missing from some of the results of the 1GB source data set tests of the full output experiment.

3.1 Experimental set-up

The following software versions has been used throughout this project:
- Hadoop 2.0.0-cdh4.4.0 (with 10GB Ethernet, using CentOS release 6.5 (final))
- Hive 0.11.0-cdh5.0.0-beta-1
- Sqoop 1.4.3-cdh4.4.0
- psql (PostgreSQL) 9.3.4

The cluster consisted of 5 nodes, each with two Intel E5520 quad-core 2.27GHz processors and 19.6TB disk space.

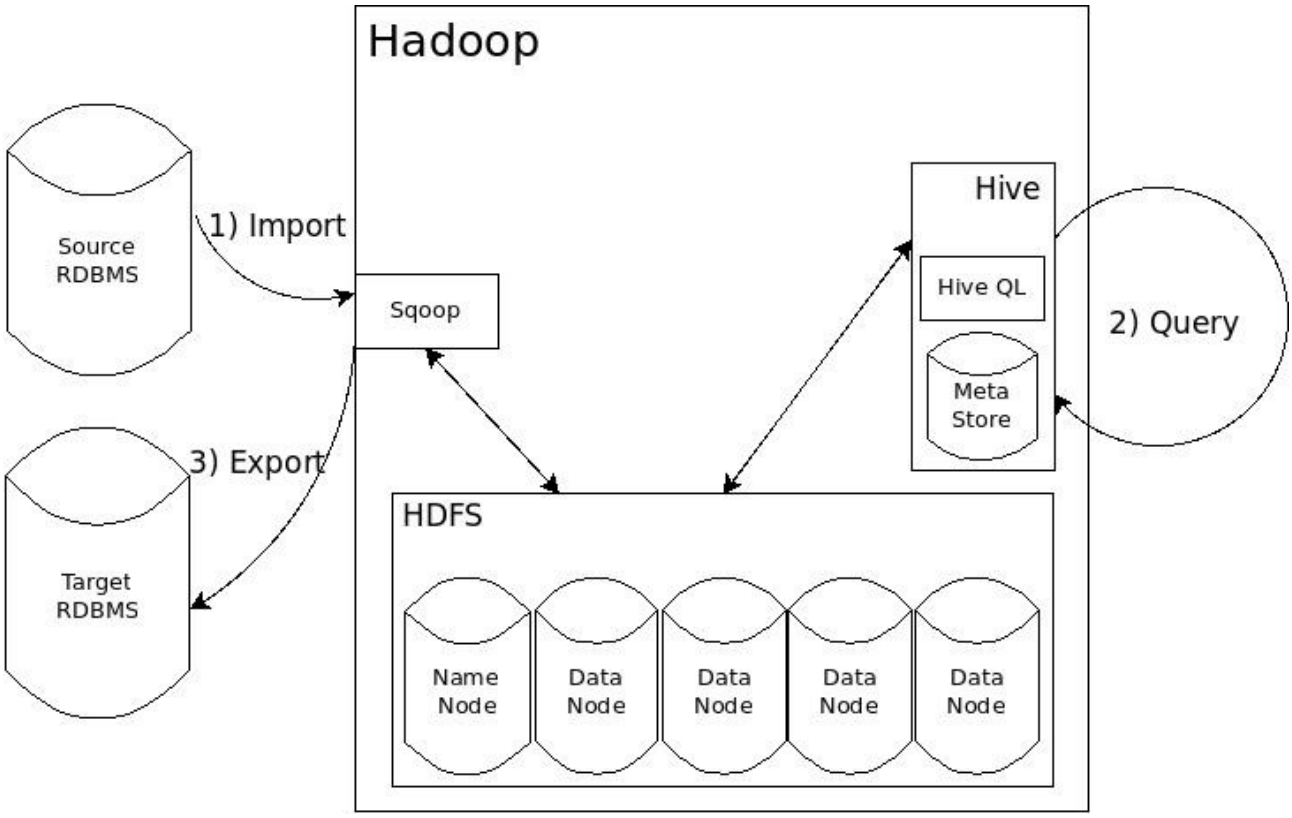

Figure 2: Flow chart showing the three steps of the experiments. No direct communication between Hive and Sqoop demands manual handling of tables and class-files

Figure 2 shows the three steps of the experiments.
1) Data was loaded from the source RDBMS into HDFS with Sqoop. Sqoop triggers several map tasks in parallel, each task using a JDBC connector to import data from the source RDBMS table to HDFS, while the meta data (table and column names) is stored in Hives metastore. At this step, import time and disk space used in HDFS for storing the imported data were measured.
2) The queries were executed on the imported data with Hive and the results were stored in HDFS as new Hive tables. At this step, the time to run the queries and store the results was measured.
3) The stored result tables from step 2 were exported to a target RDBMS using Sqoop, and at this step export time was measured.

For statistical validity, each test was run three times, and the results show the average result of each test.

Source data set sizes used were 1, 5 and 10GB.

3.2 Full output experiment

To enable complex enough Hive queries to be relevant in comparisons with the Klarna RRDB queries, a schema was created consisting of five tables connected via foreign keys. The purpose was to enable queries consisting of four joins, which are highly time consuming.

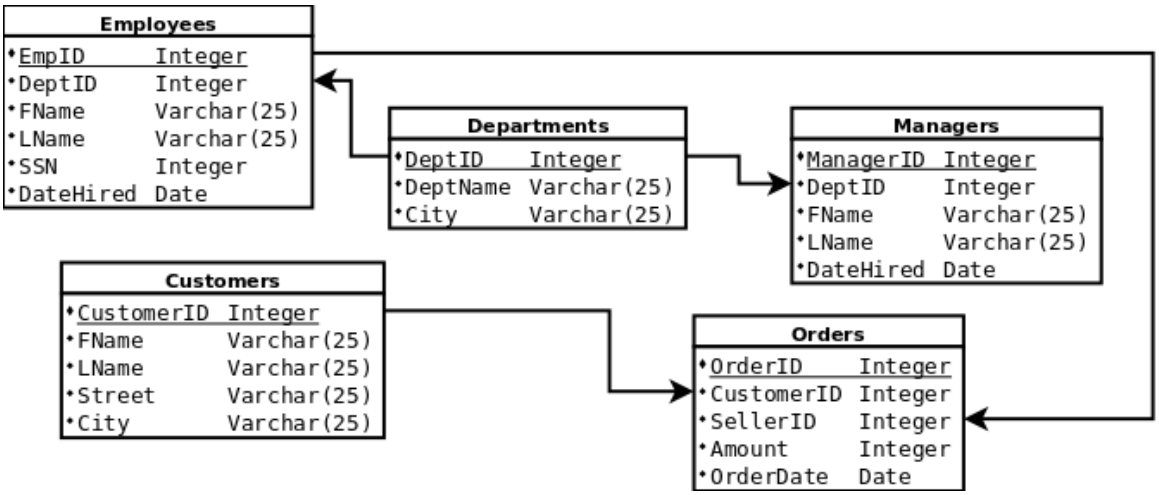

Figure 3: Database schema for the full output experiment

Figure 3 shows the tables of the database and their foreign keys.

Data was generated by developing a small Java program creating csv-files (comma-separated values files), and loaded into the source RDBMS using the PostgreSQL bulk loader.

The source table imports into HDFS were carried out in two different formats in four different settings. Text files and sequencefiles, with and without Snappy-compression.
The results of the queries were also stored in these four different settings, making up a total of 16 different settings (Appendix A, table 3).
Source database sizes of 1, 5 and 10GB were used.

When using Sqoop import, Java-code is automatically generated for mapping the RDBMS-data to HDFS-data. When importing data as text, this Java-code is not needed as the values are separated by delimiters and thus can be read directly by Hive. However, when using binary formats like Sequencefile or RC-file, each table has s a Java-class with each column being an attribute to this class, and this information must be passed to Hive manually.

3.2.1 Results, Full output experiment

Described here are the results of the full output experiments. Figure 4 show the times measured for import from an RDBMS to HDFS, query execution time, and export time for the result tables. Figure 5 displays the disk space usage for imported data as well as for the results of the queries.

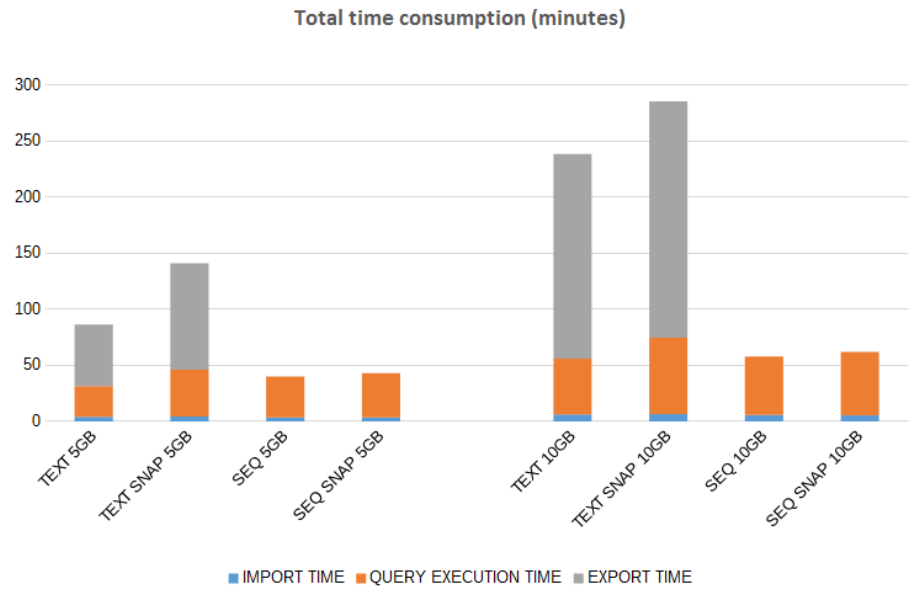

Figure 4: Diagram showing the total time consumption for import, query execution and export of 5GB and 10GB data source set

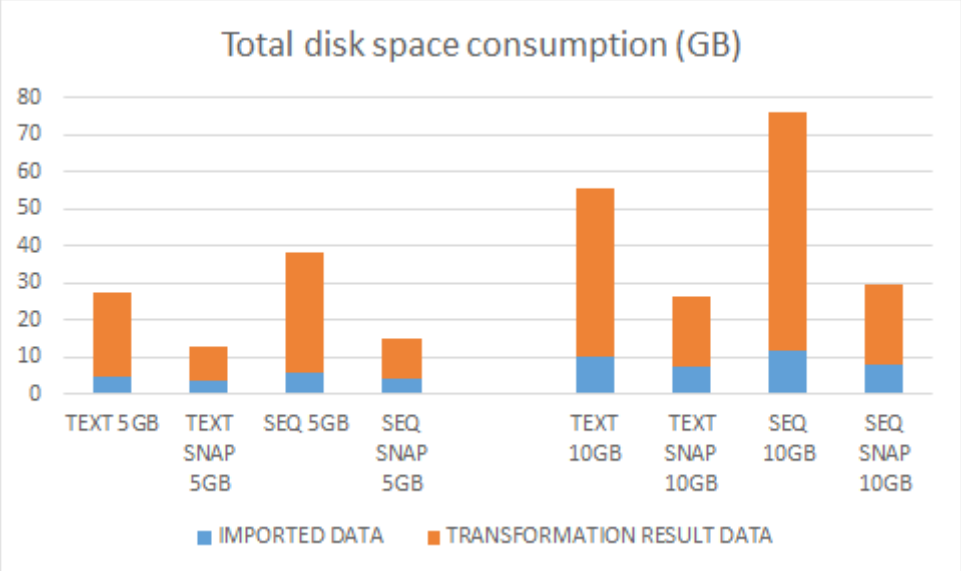

Figure 5: Diagram displaying total disk space consumption of imported and query result data

The measurement results show that there is no significant difference in import time for compressed and uncompressed text files (figure 4), while for the 5GB source data sets there's a significant overhead in query execution (55%) and export time (74%) when using compressed text as compared with using uncompressed text. With the 10GB source data sets, this overhead is reduced down to about 15-20%. The full process time of importing, querying and exporting data with compressed text show a total of about 20% overhead as compared to plain text (figure 4). Disk usage is about 50% lower for both compressed text and compressed sequencefiles as compared to uncompressed sequence files (figure 5).

The Snappy-compressed sequencefiles are about 35% smaller than their uncompressed counterparts on import and thus slightly faster to import as less data needs to be written to HDFS comparing to uncompressed formats. This difference is barely noticeable though, as compared to the time usage of the whole process (figure 5). The property of compressed data being smaller and thus faster to write to disk makes the overhead of compression negligible after just a few GB of data. A reason for the compression effect on disk space usage not being better in this experiment could be a) the compression level used was record-compression instead of block-compression, and b) the data populating the source database was highly random (making repeated patterns less frequent, thus being less suitable for compression). Record level compression was enabled to make the text file compression splittable.

The query process time measurements (figure 4) show that the 10GB sequencefiles are just slightly slower than 10GB text files to process, and Snappy-compressed sequencefiles have about a 10-15% overhead in time consumption as compared to the uncompressed text-files while needing 20% less disk space as compared to uncompressed sequence files.

As the actual data sets to be exported to the destination RDBMS are identical once exported, the compressed files have some overhead in decompressing and thus is slower than the uncompressed files to export. This overhead was reduced when files grew beyond 5GB (figure 4).

All in all, as files grew towards 10GB and beyond, Snappy-compressed sequencefiles demanded about 30-65% less disk space as compared to uncompressed sequencefiles, and the trade-off was that Hive queries were about 10% slower to execute.
For exports, Snappy-compressed text files were less than 50% the size of uncompressed text files, and used about 20% additional export time.

3.3 TPC-H experiment

These experiments were conducted using text-, sequence- and RC-files, all with and without Snappy-compression.

The schema and queries used were the TPC-H schema (figure 6). The TPC-H query suite contains a number of queries, however we have selected queries 3, 9, 13 and 14 (appendix E) based on their similarities with Klarna's RRDB transformation queries.

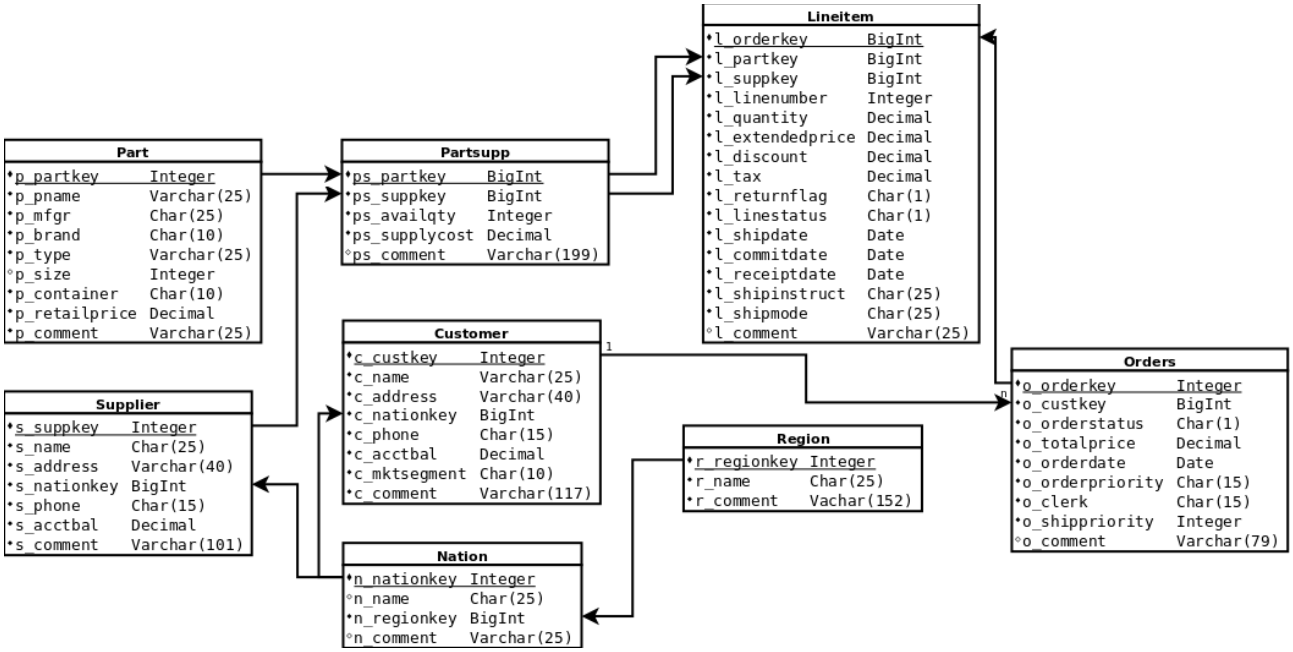

Figure 6: The TPC-H schema.

Some of the TPC-H queries, in particular those containing nested loops, uses a syntax which is not compatible with PostgreSQL, and the TPC-H database generator DBGEN which used for generating data to populate the database with has no option for working together with this particular RDBMS. Since we were only interested in populating the PostgreSQL database and running the queries in Hive, we avoided the problem by choosing the DEBGEN setting to 'Oracle', and then converting the generated data into CSV files and then loading them into PostgreSQL.

Imports into HDFS in this experiment were done using HCatalog. It provides abstraction over Hive to enable other Hadoop applications to use Hive schemas, regardless of file format and where the data is stored. This eliminated the manual storing and passing of Java class-files during Sqoop import and running Hive queries as performed in the full output experiment.

As PostgreSQL, Hive and HCatalog do not use identical data types, all decimal values in the TPC-H tables had to be converted to doubles with Sqoop to Hive import. The precision of the decimals were two digits, and so the results were identical for all queries except query number 14, which uses a division function. This gave a slight difference on the result. The small difference on query results had no significant effect on the experiment.

The Hive query language HiveQL is an extended subset of SQL, meaning HiveQL is not identical to SQL and therefore some small changes had to be applied as HiveQL does not support implicit join notation: All joins written using FROM … WHERE clauses had to be rewritten using the explicit JOIN ... ON syntax (Appendix D).

3.3.1 Results, TPC-H experiment

Figure 7 shows import time from an RDBMS to HDFS, query execution time (which includes the time taken to store the query results as new Hive tables in HDFS, which Hadoop does by default), as well as export time from HDFS to an RDBMS. Figure 8 shows total disk space consumption of imported and query result data. The query result data is very small (Appendix B, table 14), and thus is not shown in figure 8.

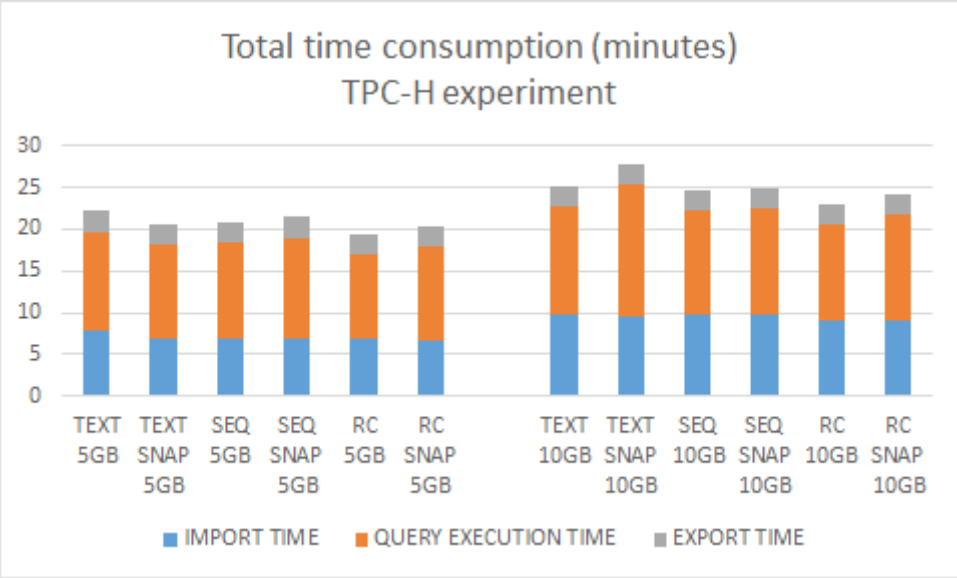

Figure 7: Diagram showing total time consumption for the TPC-H experiment, using 5GB and 10GB source data

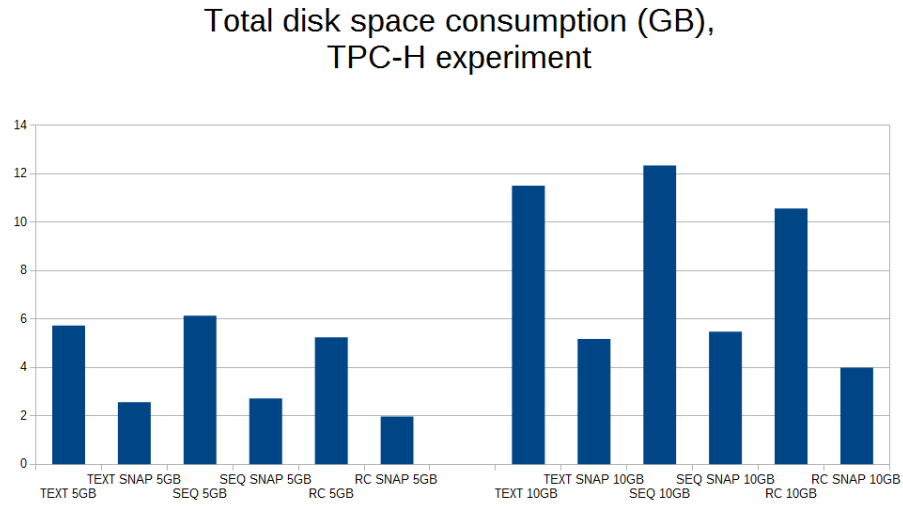

Figure 8: Diagram showing total disk space consumption for the TPC-H experiment, using 5GB and 10GB source data

The disk space measurement results show that using Snappy-compressed RC-files was quicker to import than both the other Snappy-compressed formats (figure 7). This was most likely due to the better compression ratio, causing less data to be written to HDFS. The Snappy-compressed RC-file gave the best compression ratio (less than 2/5 that of any of the uncompressed formats) as well as the least disk space usage (figure 8).

Measurements on query execution times show little to no difference between uncompressed and Snappy-compressed text files until the source data set reaches 10GB (figure 7), where the Snappy-compressed text tend to demand more execution time.

Neither compressed sequencefiles or compressed RC-files show any overhead in transformation time as compared to both compressed and uncompressed text-files (figure 7 and appendix B, tables 8,9,11,13), and the RC-file format is both the fastest to import as well as the least disk consuming format of this experiment.
The difference in query execution time between compressed and uncompressed source data is less than 10% for both sequencefiles and RC-files for the 10GB data sets.
A surprising result is that queries using source data as plain (uncompressed) text and also storing the results as plain text (appendix B, tables 6-13) were the slowest on all three data set sizes, despite no additional compressing or decompressing having to be carried out for these queries. The difference in time, however, is marginal, and one reason could be that as compressed data is smaller than uncompressed data, less data has to be written to HDFS when using compressed formats. Another reason could be that writing text to HDFS simply takes slightly longer than writing the other formats.

Worth to notice is that despite setting the compression-level to block for all formats, Hadoop still used record-level compression for text-files, thereby enabling splittability.

The scaling in these tests with regards to time consumption went very well (figure 8) due to more mappers being used when the data sets became larger and therefore split up across several HDFS blocks enabling further parallellisation.

As TPC-H queries are very selective, the resulting tables are quite small and thus no conclusions can be drawn from the marginal differences in disk space usage and export time of the different file formats (tables 14-15). Even though the result table size were very small there still was a small overhead in creating output in any other formats than the plain text format.

3.4 gzip experiment

To get a comparison with another compression method, gzip was used for the same schema, data sets and queries as used in the TPC-H experiments. Data sets were imported as gzip-compressed sequencefiles and gzip-compressed RC-files. To get statistics for the most interesting transformations, we ran the queries from these two formats to produce uncompressed text, as well as from uncompressed text to gzip text, sequence gzip and RC-gzip file formats (table 18).

The compression-level used with gzip was the default setting (-6), which aims slightly more for compression ratio than for speed.

3.4.1 Results, gzip experiment

Below are diagrams from the test runs with import time in figure 9, transformation time in figure 10 and disk space usage in figure 11.

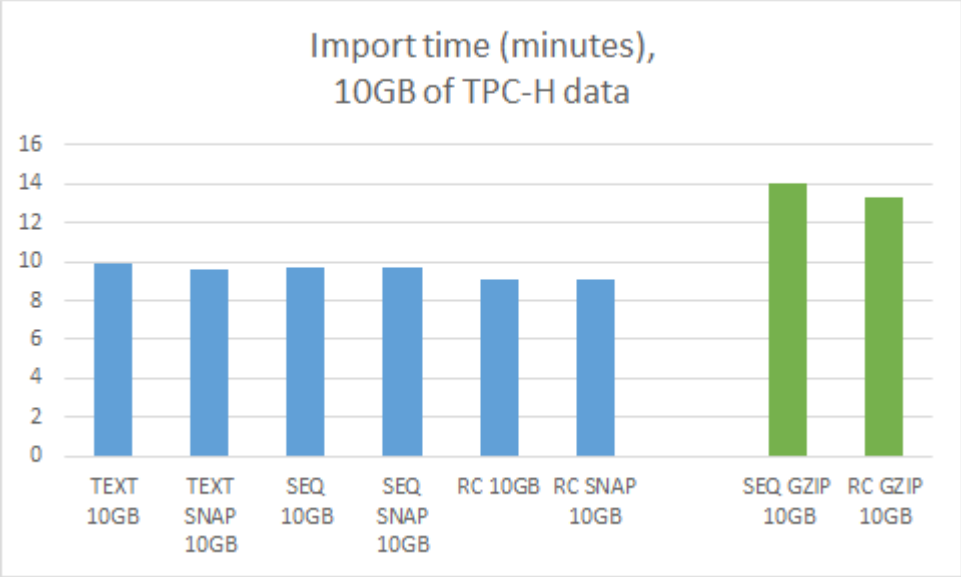

Figure 9: Diagram showing import time of 10GB of TPC-H data to HDFS with gzip-compressed formats to the right

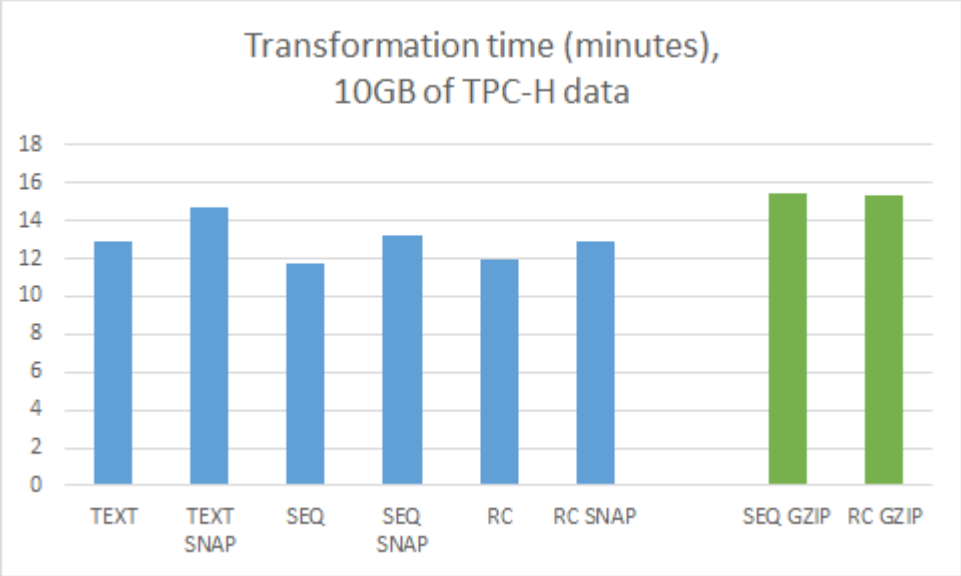

Figure 10: Diagram showing query execution time for 10GB of TPC-H data with results stored as uncompressed text. gzip-compressed formats to the right

Import times are fairly equal between gzip-compressed sequencefiles and gzip-compressed RC-files (figure 9, Appendix C, table 16), but the disk space usage is almost 30% lower for the RC-files (figure 11). In comparison to uncompressed text, gzip is significantly slower to import (about 30-40% additional time), but also significantly lower on disk usage (about 70-80% lower).

The transformation times from gzip-compressed sequencefile and gzip-compressed RC-file are almost identical, but as compared with the Snappy-compressed formats the gzip-compressed data sets are a little slower to query due to slower decompression.

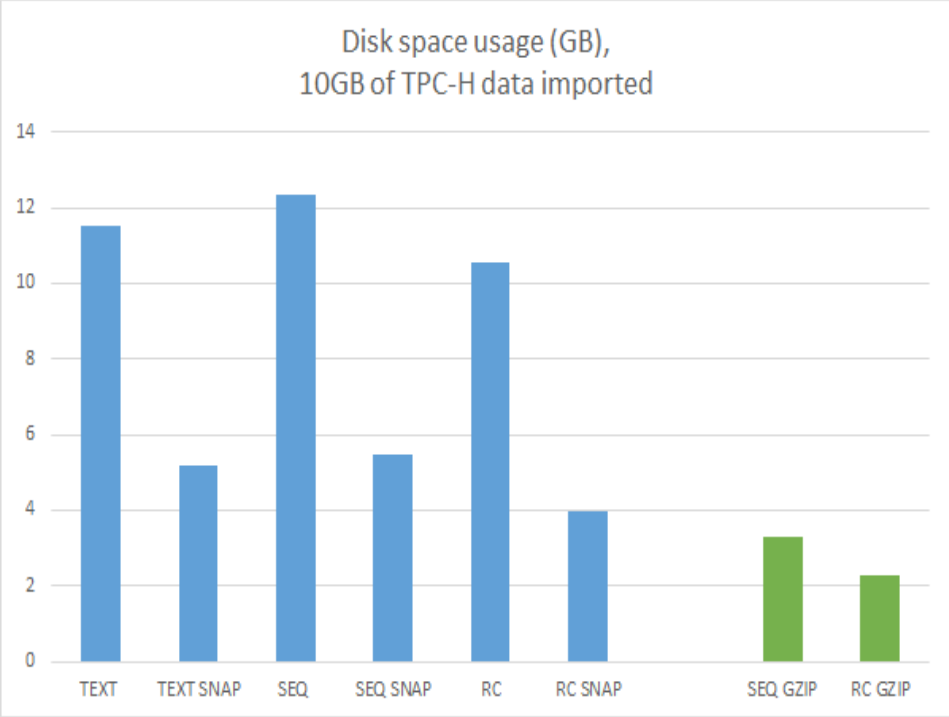

Figure 11: Diagram showing disk space usage for 10GB of imported TPC-H data with gzip-compressed formats to the right

As in the TPC-H experiment, with the answers from TPC-H queries being highly selective, the result tables were too small to draw any statistical conclusions from with regards to disk space consumption (table 14) or export time.

4 Conclusions

With data sets growing over a few GB, the trade-off in time when using Snappy-compression together with sequencefiles quickly becomes small with about 10% time overhead as compared to using plain text files. The savings on disk space on the other hand, is between 20-65% (depending on the randomness of the data as well as the compression-level used).

With tabular data, RC-files combined with Snappy compression gives even better compression rates and slightly quicker imports as compared to Snappy-compressed sequencefiles.

With gzip compression, the compression rates are even better than for the Snappy-compressed ones, but with considerably slower execution times.

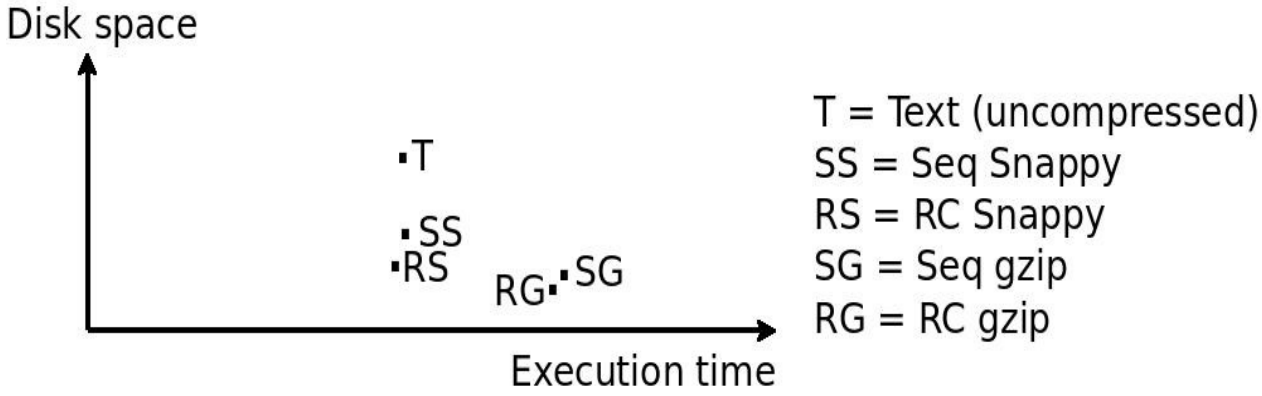

Figure 12 Image comparing total time consumption (x-axis) and total disk space usage (y-axis) for importing, querying and exporting 10GB of TPC-H source data in various formats

The values in figure 12 comes from importing, querying and exporting 10GB of TPC-H data from an RDBMS with Sqoop in various file formats, with and without Snappy- and gzip-compression, and shows the differences in the time consumption and disk space usage trade-off. The highest point shows uncompressed text taking up more disk space than the compressed formats. All snappy compressed formats were quick to import as compared to uncompressed text, with RC-snappy compressed being the quickest. Gzip-compressed files gave the best compression, while taking significantly longer to import than the Snappy-compressed formats.

With TPC-H queries giving small result tables, it is difficult to draw conclusions from their disk space usage or export times. However, querying data with Hive includes reading from HDFS which offers some clues about decompression times. Figure 10 displays query execution time with source data in HDFS in various formats, and all result outputs using the same format (text), and it shows that decompressing gzip-compressed formats is noticeably slower than decompressing Snappy-compressed formats. We can also see that decompressing Snappy-compressed RC-files takes about the same time as reading uncompressed text from HDFS, which in turn gives reason to believe that Snappy-compressed sequencefiles and RC-files would not demand a significant increase in export time as compared to text files. As decompression is faster than compression, and Snappy-compressed text was slower to import than both other Snappy-compressed formats, this also gives reason to believe that export of Snappy-compressed sequencefiles or RC-files would not give more overhead than the export of Snappy-compressed text files (figure 5).

It can be concluded from the results that when data sets are large, which is expected in Hadoop, and time is of importance, Snappy-compression on RC-files is to be the preferred file format and compression method for this use-case. If time is not an issue, say for long-time storage with few reads, then gzip compression on RC-files might be preferred as it gives the

best compression ratio. Furthermore, if the data is not relational or if the data files are small in comparison to the HDFS block size, then sequencefiles are be to preferred over RC-files.

5 Related and future work

Running the TPC-H Benchmark on Hive[9] explains how to translate the TPC-H queries from SQL to Hive QL. In comparison to [9], this thesis does not try to give a benchmark result on the full TPC-H query set, but uses this benchmark as a tool to run an easily replicated set of business-oriented queries in order to compare the differences in disk space usage and execution time when using different file formats and compression methods with Hive.
Data Compression In Hadoop[10] gives a good introduction to the subject, though it is unclear what data and queries having been used. [10] also uses compression for Hives intermediate outputs, something that was not in the scope of these experiments.

What could be useful in the future would be to run these experiments with ORC- and Parquet-files with Snappy, gzip and Bzip2 compression, including compression for Hives intermediate outputs, to further compare execution times and disk space usage.
As the TPC-H queries are highly selective thus rendering only small result tables for export, it would be useful to measure export times with larger data sets than used in our TPC-H experiments. Also, experiments using BLOCK-level compression on all formats but text (as this is the recommended format for using for example sequencefiles[11]), could be useful to get a clearer understanding of the export time of different file formats.

Acknowledgements

Continuous guidance and technical support by Ehsan Haq at Klarna during this project has been greatly appreciated, as well as the assistance provided by Erik Zeitler (Klarna), Sylvia Stefanova (Uppsala University) and all the other staff members at Klarnas Odin group.

Appendix A – Detailed results, full output experiments

1. Import time

	Text	Text Snap	Seq	Seq Snap
1GB source	2.56 min	N/A	2.04 min	2.1 min
5GB source	4.25 min	4.3 min	3.69 min	3.63 min
10GB source	6.2 min	6.5 min	5.71 min	5.44 min

Table 1: Import time from 1, 5 and 10GB source data set

Disk space of imported data

	Text	Text Snap	Seq	Seq Snap
1GB source	1GB	N/A	1.1GB	0.8GB
5GB source	5GB	3.7GB	6GB	4.1GB
10GB source	10GB	7.5GB	12GB	8.2GB

Table 2: Disk space used after import of 1, 5 and 10GB data set

Transformation time

	Text to text	Text to text Snap	Text to seq	Text to seq Snap	Text Snap to text	Text Snap to text Snap	Text Snap to seq	Text Snap to seq Snap	Seq to text	Seq to text Snap	Seq to seq	Seq to seq Snap	Seq Snap to text	Seq Snap to text Snap	Seq Snap to seq	Seq Snap to seq Snap
1GB	14.98	N/A	15.74	14.06	N/A	N/A	N/A	N/A	15.67	16.89	16.21	16.14	16.08	N/A	16.32	16
5GB	27.26	35.18	35.17	33.97	42.33	41.96	41.22	41.53	36	35.91	36.4	35.09	39.58	38.5	39.35	39.56
10GB	49.7	48.56	50.04	49.43	58	68.25	69.57	69.68	51.06	49.89	52.12	50.63	54.72	53.83	55.21	56.59

Table 3: Time taken to transform data from and to various formats.
Left column displays source data set size and time is measured in minutes

2. Disk space of result tables

Text	Text Snap	Seq	Seq Snap
4.66GB	1.87GB	6.4GB	2.32GB
22.7GB	9.3GB	32.1GB	10.8GB
45.36GB	18.66GB	64.13GB	21.6GB

Table 4: Disk space used for transformation results stored in HDFS

Export time

Text size	Text export time	Text Snap size	Text Snap export time
4.66GB	21.78 min	1.87GB	20.47 min
22.7GB	54.89 min	9.3GB	94.9 min
45.36GB	182.6 min	18.66GB	210.66 min

Table 5: Data set size and their respective export time as uncompressed and compressed text

Appendix B – Detailed results, TPC-H experiments

Import time

	Text	Text Snap	Seq	Seq Snap	RC	RC Snap
1GB	5.28 min	5.38 min	5.5 min	5.95 min	5.85 min	5.25 min
5GB	7.87 min	6.97 min	7.02 min	7 min	6.83 min	6.75 min
10GB	9.87 min	9.61 min	9.71 min	9.69 min	9.05 min	9.1 min

Table 6: Import time from 1, 5 and 10GB source data set

Disk space of imported data

	Text	Text Snap	Seq	Seq Snap	RC	RC Snap
1GB	1.13GB	0.5GB	1.21GB	0.53GB	1.05GB	0.39GB
5GB	5.73GB	2.57GB	6.14GB	2.72GB	5.25GB	1.98GB
10GB	11.51GB	5.18GB	12.35GB	5.48GB	10.57GB	3.99GB

Table 7: Disk space used after import of 1, 5 and 10GB data set

Query time

TEXT	Text	Text Snap	Seq	Seq Snap	RC	RC Snap
1GB	10.34 min	9.75 min	9.98 min	9.36 min	9.66 min	9.59 min
5GB	11.81 min	11.12 min	11.18 min	11.13 min	11.2 min	11.01 min
10GB	12.87 min	12.2 min	11.82 min	12.07 min	12.04 min	12.1 min

Table 8: Transformation times from a source data set in text format

TEXTSNAP	Text	Text Snap	Seq	Seq Snap	RC	RC Snap
1GB	9.93 min	9.71 min	9.57 min	9.61 min	9.42 min	9.52 min
5GB	12.08 min	11.26 min	11.37 min	11.36 min	11.43 min	11.25 min
10GB	14.65 min	15.78 min	14.62 min	14.84 min	14.85 min	14.71 min

Table 9: Transformation times from a source data set in Snappy-compressed text format

SEQ	Text	Text Snap	Seq	Seq Snap	RC	RC Snap
1GB	10.52 min	10.07 min	10.01 min	10.37 min	9.77 min	9.88 min
5GB	11.74 min	11.52 min	11.37 min	11.37 min	11.34 min	11.23 min
10GB	11.38 min	12.46 min	12.48 min	12.45 min	12.59 min	12.53 min

Table 10: Transformation times from a source data set in sequencefile format

SEQSNAP	Text	Text Snap	Seq	Seq Snap	RC	RC Snap
1GB	10.91 min	10.61 min	10.25 min	10.57 min	10.43 min	10.47 min
5GB	12.46 min	12.24 min	12.09 min	12.02 min	12.08 min	11.9 min
10GB	13.2 min	13.09 min	12.75 min	12.92 min	12.7 min	12.84 min

Table 11: Transformation times from a source data set in Snappy-compressed sequencefile format

RC	Text	Text Snap	Seq	Seq Snap	RC	RC Snap
1GB	10.3 min	9.19 min	9.58 min	9.56 min	9.53 min	9.68 min
5GB	11.61 min	10.65 min	10.51 min	10.5 min	10.06 min	10.45 min
10GB	11.91 min	11.85 min	11.66 min	11.49 min	11.58 min	11.88 min

Table 12: Transformation times from a source data set in RC-file format

RCSNAP	Text	Text Snap	Seq	Seq Snap	RC	RC Snap
1GB	10.09 min	9.43 min	9.44 min	9.75 min	9.6 min	9.44 min
5GB	12.03 min	11.97 min	11.85 min	11.38 min	11.75 min	11.13 min
10GB	12.94 min	12.79 min	12.66 min	12.73 min	12.64 min	12.66 min

Table 13: Transformation times from a source data set in Snappy-compressed RC-file format

Disk space of result tables

Text	Text Snap	Seq	Seq Snap	RC	RC Snap
9603 bytes	4289 bytes	12787 bytes	5346 bytes	7027 bytes	2949 bytes
9722 bytes	4372 bytes	12903 bytes	5424 bytes	7046 bytes	2957 bytes
9691 bytes	4371 bytes	12903 bytes	5448 bytes	7063 bytes	2985 bytes

Table 14: Disk space used for transformation results stored in HDFS

Export time

Text size	Text export time	Text Snap size	Text Snap export time	Seq size	Seq export time	Seq Snap size	Seq Snap export time	RC size	RC export time	RC Snap size	RC Snap export time
9603 bytes	2.25 min	4289 bytes	2.28 min	12787 bytes	2.26 min	5346 bytes		7027 bytes	2.45 min	2949 bytes	3 min
9722 bytes	2.56 min	4372 bytes	2.42 min	12903 bytes	2.46 min	5424 bytes	2.42 min	7046 bytes	2.57 min	2957 bytes	2.54 min
9691 bytes	2.46 min	4371 bytes	2.38 min	12903 bytes	2.4 min	5448 bytes	2.25 min	7063 bytes	2.31 min	2985 bytes	2.44 min

Table 15: Data set size and their respective export time for different file formats

Appendix C – Detailed results, gzip experiments

Import time

	SEQ SNAP	RC SNAP
1GB	5.86 min	6.09 min
5GB	9.27 min	8.89 min
10GB	14.06 min	13.28 min

Table 16: Import time for 1, 5 and 10GB data sets

Disk space of imported data

	SEQ SNAP	RC SNAP
1GB	0.32	0.22
5GB	1.63	1.13
10GB	3.28	2.29

Table 17: Disk space used for storing imported data in HDFS

Transformation time

	Text to text gzip	Text to seq gzip	Text to RC gzip	Text gzip to text	Seq gzip to text
1GB	9.2 min	8.97 min	9.17 min	9.44 min	8.85 min
5GB	11.43 min	10.96 min	10.86 min	13.38 min	13.17 min
10GB	12.75 min	12.36 min	12.49 min	15.42 min	15.35 min

Table 18: Time taken to transform data from and to various formats.
Left column displays source data set size and time is measured in minutes

Disk space of result tables

	Text gzip	Seq gzip	RC gzip
1GB	3658 bytes	3714 bytes	2669 bytes
5GB	2736 bytes	3775 bytes	2674 bytes
10GB	2719 bytes	3769 bytes	2687 bytes

Table 19: Disk space of query results stored in HDFS

Export times

N/A

Appendix D – Queries for full output experiment

```
1) SELECT  m.FName AS managerFname, d.DeptName as departmentName, e.FName AS
employeeFname, c.FName as customerFname
 FROM Managers m
 JOIN Departments d on d.deptID = m.DeptID
 JOIN Employees e on e.DeptID = d.DeptID
 JOIN Orders o on o.SellerID = e.EmpID
 JOIN Customers c on c.CustomerID = o.CustomerID
 WHERE c.City <> d.City;

2) SELECT  m.FName AS managerFname, d.DeptName AS departmentName, e.FName AS
employeeFname, c.FName AS customerFname
 FROM Managers m
 JOIN Departments d on d.deptID = m.DeptID
 JOIN Employees e on e.DeptID = d.DeptID
 JOIN Orders o on o.SellerID = e.EmpID
 JOIN Customers c on c.CustomerID = o.CustomerID
 WHERE c.City like '%ab%';

3) SELECT c.CustomerID AS customerid, o.OrderID AS orderid, e.EmpID AS employeeid
FROM Customers c
JOIN Orders o on o.CustomerID = c.CustomerID
JOIN Employees e on e.EmpID = o.SellerID
JOIN Departments d on d.DeptID = e.DeptID
JOIN Managers m on m.DeptID = d.DeptID
WHERE m.ManagerID < 50;

4) SELECT sum(o.Amount) AS ordersAmountSum, m.FName AS managerFname, e.DeptID AS
employeeDeptId, COUNT(c.CustomerID) AS customerIdCount
 FROM Orders o
 JOIN Employees e on o.SellerID = e.EmpID
 JOIN Customers c on c.CustomerID = o.CustomerID
 JOIN Departments d on d.DeptID = e.DeptID
 JOIN Managers m on m.DeptID = d.DeptID
 WHERE d.DeptID >10
 GROUP BY m.FName, e.DeptID;
```

Appendix E – Queries, TPC-H experiment

Shipping Priority Query (Q3)

```
SELECT l_orderkey, SUM(l_extendedprice * (1 - l_discount))
AS revenue, o_orderdate, o_shippriority
FROM customer, orders, lineitem
WHERE
c_mktsegment = ':1'
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate < date ':2'
AND l_shipdate > date ':2'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY
revenue DESC, o_orderdate
LIMIT 10;
```

rewritten as

```
SELECT l.l_orderkey, SUM(l.l_extendedprice * (1 - l.l_discount))
AS revenue, o.o_orderdate, o.o_shippriority
FROM customer c JOIN orders o
ON c.c_mktsegment = '${hiveconf:Q3MKTSEGMENT}'
AND c.c_custkey = o.o_custkey
AND o.o_orderdate < '${hiveconf:Q3DATE}'
JOIN lineitem l ON
l.l_orderkey = o.o_orderkey
AND l.l_shipdate > '${hiveconf:Q3DATE}'
GROUP BY
l.l_orderkey,
o.o_orderdate,
o.o_shippriority
ORDER BY
revenue DESC,
o_orderdate
LIMIT 10;
```

Product Type Profit Measure Query (Q9)

```
SELECT nation, o_year, SUM(amount)
AS sum_profit
FROM (
select n_name AS nation,
EXTRACT(year FROM o_orderdate) AS o_year,
l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity AS amount
FROM part, supplier, lineitem, partsupp, orders, nation
WHERE s_suppkey = l_suppkey
AND ps_suppkey = l_suppkey
AND ps_partkey = l_partkey
AND p_partkey = l_partkey
AND o_orderkey = l_orderkey
AND s_nationkey = n_nationkey
AND p_name LIKE '%:1%')
AS profit
GROUP BY nation, o_year
ORDER BY nation, o_year DESC;
```

rewritten as

```
SELECT nation, o_year, SUM(amount) as sum_profit
FROM (
SELECT n_name AS nation,
year(o_orderdate) AS o_year,
l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity AS amount
FROM orders o JOIN (
SELECT l_extendedprice, l_discount, l_quantity, l_orderkey, n_name, ps_supplycost
FROM part p JOIN (
SELECT l_extendedprice, l_discount, l_quantity, l_partkey, l_orderkey, n_name, ps_supplycost
FROM partsupp ps JOIN (
SELECT l_suppkey, l_extendedprice, l_discount, l_quantity, l_partkey, l_orderkey, n_name
```

FROM (
SELECT s_suppkey, n_name
FROM nation n
JOIN supplier s ON n.n_nationkey = s.s_nationkey) s1
JOIN lineitem l ON s1.s_suppkey = l.l_suppkey) l1 ON ps.ps_suppkey = l1.l_suppkey AND ps.ps_partkey = l1.l_partkey) l2 ON p.p_name LIKE '%:1%' AND p.p_partkey = l2.l_partkey) l3 on o.o_orderkey = l3.l_orderkey) profit
GROUP BY nation, o_year
ORDER BY nation, o_year desc;

Customer Distribution Query (Q13)

SELECT c_count, count(*) as custdist
FROM(
SELECT c_custkey, count(o_orderkey)
FROM customer LEFT OUTER JOIN orders
ON c_custkey = o_custkey
AND o_comment NOT LIKE '%:1%:2%'
GROUP BY c_custkey)
AS c_orders (c_custkey, c_count)
GROUP BY c_count
ORDER B custdist DESC,
c_count DESC;

rewritten as

SELECT c_count, count(*) AS custdist
FROM (
SELECT c_custkey, count(o_orderkey) AS c_count
FROM customer c LEFT OUTER JOIN orders o
ON c.c_custkey = o.o_custkey AND NOT o.o_comment LIKE '%${hiveconf:Q13STRING1}%${hiveconf:Q13STRING2}%'
GROUP BY c_custkey), c_orders
GROUP BY c_count
ORDER BY custdist DESC, c_count DESC;
Promotion Effect Query (Q14)

*SELECT 100.00 * SUM(*
CASE WHEN p_type LIKE 'PROMO%'
*THEN l_extendedprice * (1 – l_discount)*
ELSE 0
*END) / SUM(l_extendedprice * (1 - l_discount)) AS promo_revenue*
FROM lineitem,part WHERE l_partkey = p_partkey
AND l_shipdate >= date ':1'
AND l_shipdate < date ':1' + interval '1' month;

rewritten as

*SELECT 100.00 * SUM(*
CASE WHEN p_type LIKE 'PROMO%'
THEN l_extendedprice(1-l_discount)*
ELSE 0.0
*END) / SUM(l_extendedprice * (1-l_discount)) AS promo_revenue*
FROM lineitem l JOIN part p ON l.l_partkey = p.p_partkey

AND l.l_shipdate >= '${hiveconf:Q14DATE}'
AND l.l_shipdate < '${hiveconf:Q14DATE2}';

Parameters used for all queries:
Q3MKTSEGMENT='AUTOMOBILE'
Q3DATE='1995-02-08'
Q9STRING='red'
Q13STRING1='idea'
Q13STRING2='quick'
Q14DATE='1996-01-01'
Q14DATE2='1996-02-01'

[1] http://www.sciencedaily.com/releases/2013/05/130522085217.htm (2014-08-08)
[2] http://hadoop.apache.org/ (2014-08-08)
[3] http://hive.apache.org/ (2014-08-08)
[4] http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf (2014-08-08)
[5] http://www.databasedir.com/what-is-rdbms/ (2014-08-08)
[6] http://sqoop.apache.org/ (2014-08-08)
[7] http://databases.about.com/od/specificproducts/a/acid.htm (2014-08-08)
[8] Edward Capriolo, Dean Wampler, Jason Rutherglen - Programming Hive, ISBN 978-1-4493-1933-5 (p.255)
[9] https://issues.apache.org/jira/secure/attachment/12416257/TPC-H_on_Hive_2009-08-11.pdf (2014-08-08)
[10] http://comphadoop.weebly.com/ (2014-08-08)
[11] Tom White - Hadoop: The definitive guide, 3rd edition, ISBN 978-1-4493-1152-0 (p. 91)