MDX on Hadoop

A case study on OLAP for Big Data

Version 4
Abstract

Online Analytical Processing (OLAP) is a method used for analyzing data within business intelligence and data mining, using n-dimensional hyper cubes. These cubes stores the aggregates of multiple dimensions of the data, and can traditionally be computed from a dimensional relational model in SQL databases, known as a star schema. Multidimensional expressions are a type of queries commonly used by BI tools to query OLAP cubes. This thesis investigates ways to conduct one-line OLAP like queries against a dimensional relational model, based in a Hadoop cluster. In the evaluation, Hive-on-Spark and Hive-on-Tez and various formats have been compared. The most significant conclusions are that Hive-on-Tez delivers better performance than Hive-on-Spark, and that the ORC format seems to be the best performing format. It could not be demonstrated that less than 20-second performance could be achieved for all queries with the given setup and dataset or that order of input data significantly affects the performance of the ORC format. Scaling seems fairly linear for a cluster of 3 nodes. It also could not be demonstrated that Hive indexes or bucketing improves performance.

Acknowledgements

I would like to thank Dr. Jim Dowling for his support and counseling which has helped make this thesis become reality. I would also like to thank Kenneth Wrife, CEO at Omicron Ceti for giving me the opportunity to conduct my thesis project at such a wonderful company, and also for his support and input.

Lastly I would also like to thank my parents for all the years that they have supported me during my studies at the Royal Institute of Technology.

Jakob Stengård, Midsummer’s eve 2015.
# Table of contents

**CHAPTER 1 INTRODUCTION** ........................................................................................................... 1  
1.1 PROBLEM ........................................................................................................................................ 2  
1.1.1 Background .............................................................................................................................. 2  
1.1.2 Definition .................................................................................................................................. 3  
1.1.3 Limitations ............................................................................................................................... 3  
1.2 ETHICAL CONSIDERATIONS FOR BIG DATA .......................................................................... 3  
1.3 SUSTAINABILITY ........................................................................................................................... 4  
1.4 OUTLINE ....................................................................................................................................... 4  

**CHAPTER 2 AN INTRODUCTION TO OLAP** .................................................................................. 5  
2.1 OLAP CUBES ................................................................................................................................... 5  
2.1.1 Common approaches to the OLAP problem ............................................................................ 5  
2.2 DATA MODELS AND SCHEMAS FOR MULTIDIMENSIONAL DATABASES .................................. 7  
2.3 MONDRIAN ...................................................................................................................................... 7  
2.4 APACHE KYLIN .............................................................................................................................. 7  

**CHAPTER 3 THE HADOOP ECOSYSTEM** ...................................................................................... 8  
3.1 APACHE TEZ .................................................................................................................................. 8  
3.2 APACHE SPARK ............................................................................................................................ 8  
3.3 APACHE HIVE ............................................................................................................................... 9  
3.3.1 File formats ............................................................................................................................. 9  
3.3.2 Dividing tables ......................................................................................................................... 11  
3.3.3 Joins ......................................................................................................................................... 12  
3.3.4 Indexes ..................................................................................................................................... 15  
3.3.5 Concurrency of queries .......................................................................................................... 16  
3.3.6 Materialized views .................................................................................................................. 16  
3.3.7 Vectorization ............................................................................................................................ 16  
3.4 HADOOP VS. TRADITIONAL DATABASES ................................................................................... 16  

**CHAPTER 4 OLAP IN A BIG DATA CONTEXT** ............................................................................ 18  
4.1 THE CURSE OF DIMENSIONALITY ............................................................................................ 18  
4.2 INDEXES ONLY TAKE YOU SO FAR ......................................................................................... 18  
4.3 HADOOP IS NOT A DATABASE .................................................................................................... 19  

**CHAPTER 5 SYSTEM MODEL** ...................................................................................................... 20  

**CHAPTER 6 EVALUATION** .......................................................................................................... 23  
6.1 GOAL ............................................................................................................................................. 23  
6.2 TEST ENVIRONMENT ..................................................................................................................... 23  
6.2.1 Machine specs ....................................................................................................................... 23  
6.2.2 Configuration .......................................................................................................................... 24  
6.2.3 Hadoop software versions ....................................................................................................... 24  
6.2.4 Benchmarking software .......................................................................................................... 24  
6.2.5 Test data ................................................................................................................................... 24  
6.3 RUNS ............................................................................................................................................ 29  
6.4 STEPS TAKEN TO AVOID CACHING ......................................................................................... 29  
6.5 DIFFERENCES BETWEEN SPARK AND TEZ ON YARN .......................................................... 29  
6.6 INDEXING ..................................................................................................................................... 30  
6.7 BUCKETING .................................................................................................................................. 30  

**CHAPTER 7 RESULTS** .................................................................................................................... 32  
7.1 FORMATS, PARTITIONING AND TEZ VS. SPARK .................................................................... 32
7.1.1 Effects of vectorization on CPU usage................................................................. 36
7.2 SCALABILITY ........................................................................................................... 37
7.3 BUCKETS ............................................................................................................... 38
7.4 INDEXES ............................................................................................................... 40
7.5 ORDER OF INPUT DATA ...................................................................................... 40

CHAPTER 8 CONCLUSIONS .................................................................................... 41
8.1 DISCUSSION ........................................................................................................ 42
8.2 FURTHER WORK.................................................................................................. 43

CHAPTER 9 ADVICE TO OMICRON ..................................................................... 44
9.1 PURE ROLAP SOLUTIONS ............................................................................ 44
9.2 HOLAP WITH KYLIN ...................................................................................... 44
9.3 DRUID ................................................................................................................. 45

APPENDIX A ............................................................................................................. 46

BIBLIOGRAPHY ........................................................................................................ 47
**DAG**

Directed acyclic graph is a directed graph with no directed cycles. It is formed by a collection of vertices and directed edges, each edge connecting one vertex to another in such a way that there is no sequence of edges from a vertex $v$ that eventually leads back to $v$ again [1].

**Dimensional relational model**

Refers to the datamodel used in ROLAP, with datacubes implemented in some type of multidimensional schema in a relational database.

**Dimension table**

A table used to store dimensions in a star schema. Described in chapter 2.

**ETL**

Extract, transform and load, refers to a process within data warehousing where data is extracted from some data source, transformed into the proper format, and loaded into the final target.

**Fact table**

A table used for storing data in star schemas. Star schemas are described in chapter 2.

**HAIL**

Hadoop aggressive indexing library

**HDFS**

Hadoop distributed file system is the primary distributed file system used with Hadoop. It is designed to run on commodity hardware. HDFS was designed to be highly fault-tolerant and provide high throughput to applications. [2] It does this by sacrificing random access. It is not possible to modify data written to a file in HDFS. Instead an entirely new file is usually written, and data is copied. Appending to existing files is possible in later versions of HDFS. [3]

**Hive-QL**

Hive query language. An SQL dialect used by Apache Hive. Hive-QL constitutes a subset of the official standards combined with some application specific extensions.

**HOLAP**

Hybrid online analytical processing is a hybrid between MOLAP and ROALP where the data is stored using a dimensional relational model, however aggregations of the data are also precomputed and stored in for example, a specialized array format.

**MDX**

Multidimensional expressions is a query language for OLAP databases, much like SQL is a query language for relational databases. It is also a calculation language, with syntax similar to spreadsheet formulas.

**MOLAP**

Multidimensional online analytical processing is the traditional way of doing OLAP where data cubes are pre-computed and stored in a specially designed format.
OLAP

**Online analytical processing** refers to array-oriented database applications that enable users (analysts, managers and executives) to view, navigate through, manipulate, and analyze multidimensional databases [4]. OLAP itself does not specify any particular technology, but is rather a concept. In this report I mainly use it contrast to OLTP. The main point with OLAP is the *online* part, which means that queries are answered swiftly, so that the data can be analyzed in an interactive manner.

OLTP

**Online transaction processing** is a class of information systems that facilitate and manage transaction-oriented applications, typically for data entry and retrieval transaction processing. In contrast to OLAP, OLTP refers to processing in which the system responds immediately to smaller, non-analytical queries.

ORC

**Optimized row columnar** is a file format used in Apache Hive.

RDBMS

**Relational database management system**

RDD

**Resilient distributed dataset** is an abstraction in Apache Spark that represents an immutable, partitioned collection of data that can be operated on in parallel [5].

ROLAP

**Relational online analytical processing** is a concept that involves using a dimensional relational model to perform OLAP. ROLAP solutions commonly store data in a Data Warehouse, located in an ordinary relational database.

SQL

**Structured query language** is a special purpose query language used to query data stored in relational databases.

Star schema

A schema for storing OLAP data in a relational database. Described in chapter 2.

TPC

**Transaction processing performance council** is a non-profit organization which provides a number of benchmarks which are commonly used in the industry. Some notable members of this organization are Oracle, IBM, Redhat and Microsoft.

XML

**Extensible markup language**
Chapter 1  Introduction

In the recent years, as storage media has become cheaper, it has become feasible for businesses to store more and more information about their users, customers, products and measurements. The concept itself is nothing new, however the scale of the operation is. SINTEF, a Norwegian research institute, stated 2013 that 90% of the world’s data had been generated over the last 2 years [6].

With so much data being collected, the problem of storing, indexing and making sense of it all has become increasingly relevant. Again, this is nothing new. Data mining emerged during the 1980s and has been around since then [7, p. xxi]; however, the tools and applications for handling these enormous amounts of data have recently emerged within a field plainly referred to as "Big Data". New software tools and storage systems for use within this field have been developed during the last 10 years, such as Google’s Big Table and Apache Hadoop.

The advent of these new technologies and the challenges that these large quantities of data imposes, new ways of applying existing methods within data mining are also necessary in order to make sense of all this data. Data mining, the computational process of discovering patterns in large data sets, is also centuries old, and an extensively studied field.

This thesis investigates a technique for analyzing data within Business Intelligence (BI), called OLAP data cubes. Data cubes are models for working with data in stored in a so-called dimensional relational model, which is commonly used with BI-Tools. Queries on a cube are normally done using either SQL or MultiDimensional eXpressions, MDX. The goal of this thesis has been to find a way to do online MDX queries on Hadoop clusters using open source software as a basis. The idea has been to evaluate existing SQL solutions for Hadoop and see how and if they could be optimized to the point where doing real time MDX with normal BI-Tools is possible.

The thesis has been conducted at Omicron Ceti AB, which is a consultant company that among other things provides solutions within Business Intelligence. The current way of working with Big Data at Omicron is to use an ETL tool to extract part of the data out of the Hadoop cluster and into a DBMS, such as MySQL. This database is then used as a backing store for a ROLAP engine called Mondrian, which provides caching and MDX support.

The ETL approach has the downside that it requires data to be moved out of the Hadoop cluster, and only a small subset of the data can be viewed at a time. What would be more interesting would be if we could browse all the data directly on a Hadoop cluster interactively and not need to limit ourselves to a subset, or move the data outside of the cluster. There are obvious advantages of using for example MapReduce to build the multidimensional model or compute cube aggregates of a cube, instead of pulling data through an ETL tool, into a MySQL database. The latter clearly has scalability issues and it does not make use of the storage capacity or processing power available in the Hadoop cluster.
1.1 Problem

1.1.1 Background

What has been previously been investigated at Omicron Ceti was the use of Pentaho Bi-Server with Hive as the backend in Hadoop. Pentaho Bi-server is a package consisting of the OLAP server Mondrian and the OLAP visualization framework JPivot. However, the attempts made then were not very successful. The problem was that queries simply took too long time, and Mondrian timed out before results could be delivered. It was therefore far from an On-line system. The goal was to be able to run any type of query on the data, which makes aggregation tables and pre-computed cubes impractical. The purpose of this project was to look further, and see to what extent Hive could perform, or if we would need to consider another solution.

There are various products, which provide OLAP functionality for Big Data. The problem is that most of them are proprietary and tuned for a special purpose. One example is LinkedIn’s Avatrria [8], which was designed especially to generate many small OLAP cubes in LinkedIn’s own database, Voldemort. Voldemort and Avatria are proprietary, and the source code for these engines have not been released. E-Bay recently initiated an open source project for an OLAP engine based in Hadoop. This project goes under the name Kylin, and is an Apache incubator project. Kylin is an OLAP engine based in Hadoop with the goal of providing an SQL interface and multi-dimensional analysis on Hadoop supporting extremely large datasets. The webpage claims “Users can interact with Hadoop data via Kylin at sub-second latency, better than Hive queries for the same dataset” [9]. Kylin operates on dataass stored in Hive.

An advantage of using Hive is that Hive has been available for a while and is a de facto tool that exists on most Hadoop installations. Thus, a solution based on Hive is more portable and flexible. This is something that the Kylin developers also realized, since their choice has been to use Hive and Hbase as a backend for Kylin. Hbase is a bigtable database for Hadoop, which is also quite common on Hadoop clusters. Fortunately both Hive and Hbase store data in the distributed and fault tolerant file system HDFS, and that MapReduce jobs can be used to move data between Hive and Hbase.

In addition to Hive, the number of solutions for SQL on Hadoop is starting to approach the number of flavors of Unix. Thus, it is not possible to compare them all thoroughly within the scope of this report. Some interesting frameworks that do exist are Impala, Presto and Apache Tajo, which are all solutions for data warehousing on Hadoop. All of these claims to be faster than Hive, however most of them, if not all compare themselves against Hive on MapReduce, which is naturally slow due to the batch and fault-tolerant nature of MapReduce.

One problem with Hive is its limited support for indexes. Hive currently does not support B-tree indexes, and overall support for indexing seems to be limited as I state in my conclusions. Hive indexes are described in more detail in chapter 3.

There have been other approaches to indexing for Hadoop, such as Hadoop++ and it is successor Hadoop Aggressive Indexing Library (HAIL). HAIL works by providing a modified upload pipeline for HDFS, which creates different clustered indexes on each data block replica. The authors demonstrate that HAIL runs up to 68 times faster than Hadoop (map reduce) [10]. HAIL is mainly an improvement upon HDFS, with it is own API, which means applications must be rewritten in order to utilize the benefits of HAIL. Hive currently does not support HAIL.
1.1.2 Definition

Research question:

How can we achieve on-line (20 seconds or faster) MDX queries on a large (1 billion rows or larger fact table) dimensional relational database schema based in Hive?

The goal of this thesis has been to investigate the following:

- **Hive-on-Tez vs. Hive-on-Spark**
  Hive on Tez and Hive-on-Spark promises much, but what do they actually deliver?

- **File formats**
  What file formats are best for analytical queries?

- **Partitioning**
  How does paritioning affect performance?

- **Bucketing**
  Will this improve performance of where queries? Is this like dynamic partitions? What about the performance of Joins on bucketed tables?

- **Indexing**
  What indexing opportunities are available in Hive today? Performance?

- **Does the order which data is inserted into tables have any effect on query performance?**

- **How does it scale?** Can we predict how many machines we would need in order to achieve satisfying performance?

- **Kylin**
  Kylin is an aggregation engine for OLAP based in Hadoop. Is it a possible candidate for solving the problem?

1.1.3 Limitations

It is beyond the scope of this thesis to investigate techniques and algorithms for constructing cubes and aggregate tables. The study also does not aim to compare all available OLAP tools, techniques or SQL on Hadoop solutions. The choice was made to primarily focus on Hive, and the evaluation of Hive-on-Tez and Hive-on-Spark, as well as Kylin in the context of the problem.

1.2 Ethical considerations for Big Data

The ethical considerations of Big Data are becoming more and more relevant as the amount of data being collected grows. Thus I thought it would be worth to mention something about the subject. Today a lot of data is being collected automatically without the user’s knowledge, which puts users at an
ethical disadvantage. [11]. Collected data is used for marketing, to identify threats to national security, as a basis for political campaigns and more. Even though data might be anonymous, there might still be ethical concerns involving drawing conclusions about very specific groups of people. Also, there is always the risk of wrong conclusions being reached, and patterns being identified by algorithms where no correlation really exists.

1.3 Sustainability

Big Data has applications for measuring sustainability for example, within transportation planning, through the use of indicators. According to Cottril & Derrible, on problem with sustainability indicators is that they rely on data that is often erroneous or incomplete. Big Data could help developing more comprehensive indicators, or make current data more accurate [12].

1.4 Outline

- **Chapter 2**
  Introduction to OLAP, what it is and how it’s commonly implemented.

- **Chapter 3**
  Description of the basics of Hadoop and relevant parts Hive that the reader might need to know about.

- **Chapter 4**
  Description of OLAP in a big data context, what the challenges of performing OLAP in such a context might be.

- **Chapter 5**
  Description of the system model of the system tested.

- **Chapter 6**
  Details of the experiment setup and how the evaluation was done.

- **Chapter 7**
  Presentation of the results of the evaluation.

- **Chapter 8**
  Interpretation of the results, discussion, conclusion and future work.

- **Chapter 9**
  Advice to Omicron Ceti.
Chapter 2  An introduction to OLAP

2.1  OLAP cubes

OLAP Cubes are a way to present large sets of aggregated information. OLAP cubes can be thought of as extensions to the two-dimensional array of a spreadsheet. For example a company might wish to analyze some financial data by product, by time-period, by city, by type of revenue and cost, and by comparing actual data with a budget. These additional methods of analyzing the data are known as dimensions [13]. Users can perform a number of operations on the data cubes. Examples of OLAP operations are slicing, dicing, drill-down, roll-up and pivot [4].

For a simple example, assume that we have a two-dimensional spreadsheet, containing the total sales of different departments during the month. It is easy to see how one might add a third dimension to his spreadsheet; let us say we want to view it by month. We now have a cube with sales as the x-axis, departments as the y-axis and month as the z-axis. In order to examine a particular department, we might want to perform a drill-down and expand one of the departments in order to view the sales of each product for that particular department, or we might want to look at a different slice of the cube, in order to view a different month. Maybe we want to remove some data and look at a smaller sub-cube with the same dimensions, also called dicing.

One can see how this way of representing data might be of use for business analysts.

2.1.1  Common approaches to the OLAP problem

The traditional way of doing OLAP is by using precomputed cubes. The cubes contain all possible answers for a range of queries. This method is usually called MOLAP, however, it can also be referred to just as OLAP. These cubes are stored in array based multidimensional storage engines. The advantage of using data cubes is that it allows fast indexing to summarized data [7, p. 136].

Another approach is using an ordinary database to store the data and compute the answers to queries either on the fly, or with the help of materialized views or aggregation tables. This approach is called ROLAP (Relational OLAP). ROLAP technology tends to have greater scalability than MOLAP technology [7, p. 135].

There also exist hybrid OLAP, which combine MOLAP and ROLAP for better scalability and performance. For example, a HOLAP severer may allow large volumes of detail data to be stored in a relational database, while aggregations are kept in a separate MOLAP store [7, p. 136].
Figure 1. Examples of OLAP operations. Inspired by figure 3.10 in Data Mining: Concepts and techniques. [7, p. 124]
2.2 Data models and schemas for multidimensional databases

In traditional relational databases, the most common data model is probably the entity-relationship model. Such data models are appropriate for online transaction processing OLTP. However, a data warehouse requires a concise, subject oriented schema that facilitates online data analysis [7, p. 114].

When doing ROLAP on relational databases, the data that forms the cube is normally stored according to a dimensional relational model. Some common implementations of this type of relational models are the star schema, the snowflake schema and the fact constellation schema.

Star and snowflake schemas

Star schemas consist of a fact table with several dimension tables laid out around it. The fact table is generally large and has foreign key relationships against the smaller dimension tables. The snowflake schema is a variant of the star schema where some dimension tables are normalized and broken up into several tables [7, p. 114].

Normally, these schemas are de-normalized and do not conform to a particular normal form, something that is usually considered bad design for entity-relationship models. This means that these schemas contain a substantial amount of redundancy, although, redundancy is reduced in snowflake schemas. De-normalization is basically sacrificing disk space for speed under the assumption that joins are expensive, which is usually the case. For this reason, star schemas are more popular than snowflake schemas.

Fact constellation schemas

Also called galaxy schemas, these are star schemas with several fact tables that share some dimension tables between them [7, p. 116].

2.3 Mondrian

Mondrian is an open source OLAP server and the core component of Pentaho Bi-Server. It utilizes the MDX query language to query cubes defined by an XML schema. The data itself is stored in an ordinary relational database such as MySQL. Mondrian is thus a ROLAP server. Another feature of Mondrian is that it provides caching. This cache can be used in a distributed manner and shared among several instances of Mondrian. Caching of result sets of queries greatly enhances user experience, and allows “speed of thought” interaction with the dataset. The problem with caching is naturally that it gets outdated when data is changing. For this, Mondrian provides an (rather crude) API to enable flushing of the cache. Mondrian runs on Apache Tomcat.

2.4 Apache Kylin

Kylin is an Open Source OLAP engine based in Hadoop. The idea of Kylin is to precompute aggregates for a predefined OLAP Cube based on a multidimensional model in Hive, and then store the aggregates in HBase, which is a kind of HOLAP like approach. Kylin is still under development at the time of writing. The Kylin project is driven by E-bay, and the intention seems to be to use it as a backend for Tableau.

Kylin’s roadmap contains some interesting features for the future, such as a support for spark and support for streaming cubes. The idea of streaming cubes is to update aggregates in the cube on the fly as new data arrives. Thus, complete re-computation of the cube will not be necessary, and the cube becomes more like and aggregate cache, which is always up to date.
Chapter 3  The Hadoop ecosystem

Hadoop is a set of algorithms and frameworks used for distributed storage and distributed processing of very large datasets (Big Data). One of the advantages of Hadoop is the fact that it is designed to be used with clusters of machines built from commodity hardware, which makes deploying, maintaining and expanding an Hadoop cluster quite cost efficient. Hadoop is also available as a cloud service from various providers. This enables users to conduct tests and scale up fast without having to manage, upgrade or invest in physical machines.

The core components of Hadoop systems are

- HDFS, a distributed file system with replication
- Apache Common, common libraries shared between Hadoop modules
- MapReduce, a framework for distributed computation
- Yarn, a resource manager and application scheduler for Hadoop

Normally, Hadoop systems also run Zookeeper instances, which provides information, naming, distributed synchronization, and group services.

Hadoop was built with fault tolerance in mind and the HDFS file system as well as MapReduce was designed to be able to withstand hardware failures during operation. Thanks to HDFS replication and MapReduce write back to disk during operation, data loss can be avoided, and jobs can be restarted in the advent of for example, hard disk crashes on some limited number of machines.

It is important to note that Hadoop itself is not a database. Hadoop can be seen as a framework for running distributed applications. Traditionally, this was in the form of map reduce jobs, but with the advent of yarn, map reduce was broken out of Hadoop and became just another application that can be run on a Hadoop cluster.

3.1  Apache Tez

In 2014 Hortonworks launched the “stinger initiative” in order to speed up Hive. Hortonworks claims this initiative in total delivered 100x faster performance than Hive 0.10 [14].

Tez was developed as part of this initiative. Tez is a replacement for the traditional MapReduce, and uses optimized Directed Acyclic Graphs, which reduces the number of jobs, and steps that are required for completing a task. Tez is, with the introduction of yarn, just another application that can be run on Hadoop alongside MapReduce. It also provides sessions and reusable containers, which when used together with Hiveserver2, greatly reduces query latency. The use of sessions and reusable containers removes the need to schedule a new application on the job queue for each query, like with traditional MapReduce.

3.2  Apache Spark

Apache spark is a computing framework that can be run on yarn just like Tez. Spark uses an in-memory approach for doing computations, which according to (official Spark communication) the spark webpage is “100x faster than map reduce in memory, and 10x faster on disk” [15]. It does not say however, in which cases this holds. In any case, it is apparent that for iterative algorithms such as those used in machine learning, in memory computation can be beneficial. MapReduce with its fault tolerant
write back to disk is not that well suited for these types of iterative algorithms. Just like Tez, Spark also uses DAG based scheduling between data processing steps run on so called RDDs [16].

It is possible to use spark as a backend engine for hive. This is something that is being developed as part of the Stinger Next initiative, the continuation of the stinger initiative [14]. The Stinger Next initiative aims to provide sub-second query performance for Hive, however they do not specify for what sizes of datasets, which makes the statement quite ambiguous to say the least.

3.3 Apache Hive
Hive is an engine for doing Hive-QL (a form of limited SQL) queries on Hadoop. Tables in Hive can be created from regular text files with data stored in HDFS, and then converted into more effective formats. Originally, Hive only translated queries into MapReduce jobs. These jobs usually take minutes or longer to complete, which make them mostly suitable for batch processing. This is also one of the reasons why there is such an abundance of ways of interacting with Hadoop using SQL (Cloudera Impala and others), everybody is being “faster than Hive”. Today however, Hive can run both with Tez and Spark as query engines, which greatly increases Hive’s performance.

3.3.1 File formats
There exist a number of different file formats, which can be used to store Hive tables. These are listed below.

Textfiles
Simple row oriented CSV like files with no built in indexing. The files can be compressed using Bzip, Snappy or Gzip. According to Hive Essentials these files cannot be split during input processing, and thus results in one big map job being run for the entire file [17, p. 130].

Sequencefiles
Sequence files are a binary storage format for key-value pairs. They are more compact than plain text files. Just like the text files, sequence files are row oriented. Sequence files can be compressed on record level or block level. However this must be enabled in Hive by setting the following parameters:

```
set hive.exec.compress.output=true;
set io.seqfile.compression.type=BLOCK;
```

[17, p. 130]

Sequence files also have a sync marker written into the body of the file to allow for seeking to random points in the file. For example, in the case of block compression, this sync marker will be written before every block in the file. This sync also provides split ability for sequence files. [16]

RCFiles
Record Columnar File, is a column oriented storage format. Instead of storing each row in a horizontal fashion, RCFiles divide the rows into row groups, where the data is then is organized with rows going vertically and the values for columns going horizontally. This means that when the file is read in sequence, all the values for each column come after each other within each row group. Thus, if the reader of the format knows it is only looking for a particular column, all the data for that column can be read in sequence for each row group, and the rest of the file skipped. Thus, columns stores are more efficient when a query only requires a subset of columns. Each row group of an RCFile can be compressed.
**Avro**

Avro is based on JSON or Avro IDL. Avro can be compressed. The main focus for Avro is to provide portability, not speed. [16].

**ORC files**

Optimized Row Columnar, ORC was designed to overcome the limitations of the other hive formats. [18]. ORC files consist of stripes (several files), which in turn are divided into streams and row groups. A column can be stored as one or more streams. "For example, an integer column is represented as two streams PRESENT, which uses one with a bit per value recording if the value is non-null, and DATA, which records the non-null values." [18]

ORC is optimized for use with large datasets, and has a default stripe size of 256 MB compared to the 4 MB that is default for the RCFiles [17, p. 131]. The purpose of this is to enable large efficient reads of the format. The general idea of stripes is to enable data to be processed (and possibly updated) independently by different tasks.

ORC files have an index embedded into the format. The index entries are located in the beginning of each stripe, and stores basic statistics, such as MIN, MAX, SUM, and COUNT for columns. These statistics are not used to answer queries directly. They are only loaded when either predicate push-down is being used or when the reader seeks for a particular column [18]. The purpose of the index is to enable predicate push-down to determine if a chunk of rows contains a certain value, or if it can be skipped. How large these skip chunks are depends on the row index stride, which is set to 10,000 rows by default.

Bloom filter data is also embedded in the format. Bloom filtering is a kind of hash-set filtering, which allows a smaller hash area to be used, at the expense of having false positives with a certain probability [19]. It can be used by predicate push-down to determine if a predicate key-value may be or is definitely not in a set of values, which helps pruning row groups when searching through the table. Bloom filtering must be enabled separately for columns. It is not known whether this is actually used by predicate push-down today.

In addition ORC files also have a lot of compression features. Not only can row groups be compressed with GZIP, Snappy or BZIP2, but the ORC writer also uses dictionaries for storing strings, and run length encoding for integers among other things. In the evaluation it was observed that a Textfile with 24.6 Gb of integers became just 3.6 Gb using ORC with default settings and GZIP compression. Using pure GZIP compression on the same text file yields a 7 Gb file.

The main drawback of the ORC format is that it was designed specifically for Hive, and so is not a general purpose format.

**Parquet**

Row columnar format similar to ORC in it is design. Parquet is more widely supported in the Hadoop echo system than ORC, which is mainly used in Hive and Pig [17, p. 131]. The advantages of parquet are mostly similar to those of ORC [16].
3.3.2 Dividing tables

Partitions
Partitions can be used to divide a table into different segments, based on the value of a column that the table is partitioned on. Choosing a good column to partition on is hard. It needs to be something that is often or always included in the where clause of queries in order to be effective. It also needs to be column of low cardinality that allows us to constrain table scans to a small portion of the data without creating too much fragmentation. In the evaluation, an artificial “year” column for partitions were used. This is handy since almost all multidimensional models include a time dimension.

Buckets
In addition to partitions, tables can also be divided into buckets. In HDFS partitions for tables are implemented, as folders while buckets are implemented files. This means that a table can be both partitioned and bucketed at the same time. Tables are bucketed on a certain column, and values are distributed in between the number of specified buckets using a hash and/or a modulo function.

When creating bucketed and sorted table one need to specify the following:

`CLUSTERED BY (key) SORTED BY (key ASC) INTO [NUMBER OF] BUCKETS`

Then before insert into a table, set the following options:

```
set hive.enforce.bucketing=true;
set hive.enforce.sorting=true;
```
3.3.3 Joins

Common join
Also referred to as “shuffle join”. This is the default type of join, and also the slowest.

A common join is performed in a traditional MapReduce fashion. Mappers read from the tables to be joined and emits the join keys and join value pairs to intermediate files. A shuffle stage is then performed where pairs are sorted and merged. The reducers take these sorted results as input and does the actual join work [20]. The shuffle stage is expensive since it needs to sort and merge [20]. For an illustration of the common join, see figure 2.

![Diagram of common join](image)

**Figure 2: Common Join.** Inspired by slides by Tang et al. (Facebook) [21]
**Map join**
This type of join can be applied when a larger table is joined with one that is small enough to fit in memory. The small table is read into an in-memory hash table, which is broadcasted to all mapper nodes. [21]. Each mapper then streams through the big table and joins each record from the hash table.

This approach eliminates the expensive shuffle and reduces stages from the common join, however it only works when one of the tables is small enough, as defined by the small table setting in Hive.

The following settings need to be set in order for the map-join to be performed:

```
set hive.auto.convert.join=true;
set hive.mapjoin.smalltable.filesize= [a decent value]
```

**Figure 3:** Map Join. Inspired by slides by Tang et al. (Facebook) [21].
Bucket Joins

When both tables that are to be joined are bucketed on the join key, and the number of buckets in each table is a multiple of each other, special bucket joins can be used. When doing these joins only the buckets that match the join key in each table need to be touched [16], which should improve performance when joining large tables.

Hash bucketing ensures that all matching values for a join reside on the same nodes, and when the tables are also sorted, all matching values will reside on the same area of the disk, which makes joins easier [22].

There are a number of different bucket join types:

- **Sort merge bucket joins**
  
  This join requires that both tables are bucketed and sorted on the join key. In this case common joins are performed on each bucket in turn [17], but the sorting in the shuffle stage of MapReduce can possibly be skipped since the buckets are already sorted on the join key.

- **Bucket map-joins**
  
  This join works like a map-join, but for bucketed tables. Mapper tasks are spawned based on the big table, and buckets from small tables are replicated onto the mapper nodes. Only the matching buckets need to be replicated [21].

- **Sort merge bucket map-joins**
  
  This is basically a combination of the SMB join and the bucket map-join.

  With this type the small tables can be read on demand, which means that we do not need to hold them in memory. This join type also supports outer joins [21]. The join boils down to just merging the already sorted tables, allowing this operation to be faster than an ordinary map-join [23].

  In order to perform a SMB map-join, the following settings need to be set in Hive.

  
  ```
  set hive.auto.convert.sortmerge.join=true;
  set hive.optimize.bucketmapjoin=true;
  set hive.optimize.bucketmapjoin.sortedmerge=true;
  set hive.auto.convert.join=true;
  ```

  In addition, it is possible to specify a policy for big table selection, though the option

  ```
  hive.auto.convert.sortmerge.join.bigtable.selection.policy
  ```

  Possible values are:

  ```
  org.apache.hadoop.hive.ql.optimizer.AvgPartitionSizeBasedBigTableSelectorForAutoSMJ (default)
  org.apache.hadoop.hive.ql.optimizer.LeftmostBigTableSelectorForAutoSMJ
  org.apache.hadoop.hive.ql.optimizer.TableSizeBasedBigTableSelectorForAutoSMJ
  ```

  If AvgPartitionSizeBasedBigTableSelectorForAutoSMJ is used then tables need to be partitioned in order for SMB joins to be used. This can be quite counterintuitive.
### 3.3.4 Indexes

There exists support for two types of indexes in Hive, *Bitmap* and *Compact*. B-tree indexes are to my knowledge not supported. Note that these indexes are separate from the indexes present in for example ORC files, and are created using the SQL “CREATE INDEX” syntax. Indexes are still an experimental feature of Hive.

The compact index is stored in a table with the following structure *(value, _bucketname, _offsets)*. Value in this case refers to a value in the column being indexed on. *_bucketname* is a string pointing to the file storing the block. *_offsets* refers to the block-ids or “offsets” in the file where the value can be found.

Bitmap uses a table like *(value, _bucketname, _offset, _bitmaps)*. For the bitmap index, a single offset is given in *_offset*_.*_bitmaps* is an uncompressed bitmap encoding of the bitmap for this column value, bucketname, and row offset. The bits in the bitmap correspond to rows in the block. A bit is set if that row has the value of the values in the columns being indexed, and unset if not. If a value does not appear in a block at all, the value is not present in the bitmap [24].

Note that every unique value of the column will be stored in the index, and thus, these indexes are best suited for columns with low cardinality.
To enable the use of indexes in Hive, the following need to be specified:

```
set hive.optimize.index.filter=true;
```

### 3.3.5 Concurrency of queries

For online MDX queries, it is desirable to be able to process multiple queries at the same time. Mondrian by default posts multiple queries to Hive over JDBC, and all of these are not long running. Also, if Hive is to be used with multiple clients at once, concurrency of queries is more or less required.

Using the default configuration, it is only possible to run one query at the time in a batch manner. This means that large long running queries will block smaller ones. However, Hiveserver2 can be configured to use multiple job queues. These are set up in capacity scheduler, and assigned a percentage of the cluster resources. Using this setup, you can let Hiveserver2 spawn one Tez application on each queue, and keep these sessions running between queries. This way, the latency of starting up Tez sessions can be eliminated. However, using too many queues limits the resources of each queue. Even though queues can scale up their percentage of the cluster resources dynamically, it is recommended to avoid creating new queues. Instead, one can configure Hiveserver2 to queue up a maximum number Tez sessions on each queue. In this setup, the timeout of Tez should be reduced to a few seconds at most.

Hortonworks has a guide “tuning for interactive Hive queries”, which lists different configurations for various numbers of users.

### 3.3.6 Materialized views

Materialized views are currently not available for Hive, but they are planned to the second half of 2015. This could provide benefits for dynamic OLAP queries on Hive, by utilizing pre-computed cached materialized views for complex aggregation queries.

### 3.3.7 Vectorization

Vectorized query execution a feature in Hive that greatly reduces CPU usage. The standard query execution system processes one row at a time, which results in significant meta-data interpretation and long code paths in the inner loop. Vectorized query execution instead loads blocks of 1024 rows at a time into a set of column vectors. These vectors can then be processed in a tight loop with few branches and additional method calls, which can be compiled into relatively few instructions, which uses fewer clock cycles, on average, by effectively utilizing the processor pipeline and cache memory.

Vectorized execution currently requires data to be stored in the ORC format.

### 3.4 Hadoop vs. traditional databases

Hadoop solves a different problem than traditional RDBMs. Hadoop is in general more suited for running batch jobs for reporting on large unstructured datasets (such as log files). Originally, Hadoop was not built with interactive queries in mind. However people have been trying to tweak it more towards analytical on-line data processing during the last years. Hadoop will probably always be a back office system. It is not designed for handling large amounts of users.

**Other SQL query engines for Hadoop**

Facebook, who were involved in the original design of Hadoop, Hive and map reduce, later created Presto for use for analytical queries. They realized early that maps reduce and batch processing was not going to be a one size fits all. In addition to this, there is Tajo and Impala, which are somewhat...
similar in operation to Presto. All these engines query data on HDFS, but bypass the rest of the Hadoop frameworks and infrastructures.

**Big table databases**

Moving closer to the database world, we have Cassandra and HBase. These are actual databases and lie somewhere in-between the OLTP world and the reporting world of Hadoop. What they have in common that they are both based on google’s big table paper. They are key-value stores and are No-SQL databases (although, SQL can be used for Hbase through Phoenix). Cassandra has its own query language called CQL. HBase has an API and runs on top of HDFS and it integrates well with Hadoop in general.

One must remember that HBase and Cassandra are not relational databases; they are better viewed as large distributed disk-backed hash tables. Thus, the most efficient usage pattern for these databases is retrieving values by key.

**Document databases**

These are in general also distributed and no-sql. Here we find databases such as MongoDB and CouchDB. They do not have any schemas by default (although some schema like features are available in the form of extensions). In mongo, data is stored in JSON, and the entire database uses a Javascript engine. It also works as a distributed key-value store. These databases have very good performance for retrieving entries by id (key), and can handle a large number of concurrent small requests. They do not guarantee ACID, and they do not have transactions.

**Relational databases**

Here you find traditional databases such as Mysql and Postgres. These databases have been used and optimized over a long period of time, and in general, they have quite good read performance. The problem with these is that it is hard to scale them up when data grows. Many setups require manual sharding. One exception is Mysql Cluster, which was built for automatic sharding through partitioning of tables. Mysql Cluster uses NDB (Network Database) as it is underlying engine.
Chapter 4  OLAP in a Big Data context

There are many challenges involved in doing OLAP on large datasets in a distributed environment. Some of these problems relate to classical OLAP in general, and some arise due to the nature of distributed systems.

4.1 The curse of dimensionality

This is a classical OLAP problem, which affects basically all products that try to pre-compute cubes. A data cube can be viewed as a collection of aggregations along a set of dimensions. In SQL terms, these aggregations can be referred to as group-by’s, and each of these group-by’s can be represented by a cuboid. Each dimension can also have a hierarchy of levels, such as “day < month < quarter < year” for the time dimension.

For a cube with n dimensions we end up with:

\[ \text{Total number of cuboids} = \prod_{i=1}^{n} (L_i + 1) \]

Where \( L_i \) is the number of levels associated with dimension \( i \).

[7, pp. 137-139]

The authors of Data Mining concepts and techniques write:

\[ \text{As the number of conceptual hierarchies or cardinality increases, the storage space required for many group-by's will grossly exceed the fixed size of the input relations.} \]

[7, p. 140]

This illustrates the unfeasibility of materializing all cuboids for large datasets where the cardinality of dimensions is high. Thus, the only feasible way to perform precomputed OLAP on these datasets is by using some form of partial materialization. We can either limit the number of dimensions, their cardinality, or choose to only perform the computation for a limited part of the fact table(s). The result of this is that we pose constrains on what queries we can answer.

4.2 Indexes only take you so far

The reason why B-tree indexes are not available in Hadoop is probably not just because they are hard to implement. B-tree indexes take up a lot of space, since the nodes contain all the row id’s for every key-value, and this limits their usefulness for big data. Tree indexes are also best when you only want to retrieve a few rows (OLTP applications).

Using the book analogy, reading the index of a book does not help you much if you, for example, are going to count all the words in the book. Likewise, using an index will not help you compute aggregates over tables unless you can use them to either limit your scans, or if the index contains partial aggregates for columns.
4.3 Hadoop is not a database

It is a distributed system framework built for computations. Hadoop and Hive lacks some features that ordinary database systems have such as:

- Tree indexes
- Join indexes
- Materialized Views

Materialized Views are commonly used as a caching mechanism in ROLAP systems, where they are set up as aggregate tables, containing precomputed results of common queries, so it’s unfortunate that they are not available. In addition to this we have the batch manner in which Hadoop executes queries and the fact that data is distributed and replicated for fault tolerance, unlike common SQL databases where all data relies on the same machine. HDFS is also designed for storing large files and for streaming access, not random access.

Joins

Large joins have the potential of being extremely expensive in Hadoop if large amounts of data end up needing to be transferred over the network. The same goes for large result sets. Fortunately, when doing ROLAP, dimension tables are generally small. When tables are small enough to fit in memory and be easily transferred over the network, we can perform efficient maps-side joins.

When doing aggregations, we generally perform full table scans on fact-tables. Full table scans are IO-bound operations that depend on the number/speed of disks, and the number of machines in the cluster. There is a slight difference between doing an aggregation and performing a where search for one specific item though, and that is that every machine produces a result, not just the machine that happens to find the item. In order to compute the final result of a group-by, these intermediate results need to be combined in a reduce step. Fortunately, the intermediate results in these cases are small, and therefore the greatest cost comes from starting up and performing the reduce-step itself.

Projections

When using column-oriented formats, the more columns we involve in a query, the slower it becomes since we touch more data during our scans. On the flip side, this means that adding a lot of columns need not hurt performance as long as the columns length is kept constant. ORC for example, is also good at compressing low-cardinality columns, and low-cardinality columns can be efficiently indexed using bitmap indexes in Hive.

Another way to limit the amount of data that full table scans touches, is to make them non full-table by utilizing partitioning, or possibly, skipping rows by dividing or sorting columns. As previously described, ORC can skip rows in columns by utilizing column statistics for row groups.
Chapter 5  System model

Figure 5 shows the deployment diagram of the system. Pentaho Bi Server consists of Mondrian and a web frontend, running JPivot as a plugin. Mondrian communicates through JDBC to Hiveserver2 on
the Hadoop masternode, Hive then schedules Tez or Spark jobs to run on yarn in the Hadoop cluster. When running Kylin, a Hbase master server and 3 region servers need to be launched as well, in addition to the Kylin server which runs on its own Tomcat instance.

Note that this is not a recommended setup for a larger Hadoop cluster. Normally when running clusters for production, a dedicated master node is used. The masternode then only runs a HDFS namenode and yarn resource manager. In addition, a secondary name node and more than one zookeeper instance would probably be desirable. One problem in production systems is scaling of the name node. The most common solution for this is using HighAvailability (HA) name nodes, but this solution still only allows scaling to two name nodes [28].

The greatest benefit from using a HA setup seems to be automated failover in case of namenode failure.

---

Figure 6, based on illustrations on the Hive Confluence wiki [42], and Yarn walkthrough by Murthy (Hortonworks) [41]

Figure 6 shows a detailed view of Hive to the left and Hadoop Yarn running MapReduce2 on the right. Note that this is based on a picture of Hive from 2011 with Map Reduce 1. It’s probably safe to assume that Hive works somewhat similar today. The components are interchangeable. For example, when running Tez, a TezCompiler instance is used.
It is interesting to note that Hive can read directly from HDFS. This is done in two cases. For simple select queries, Hive does not always submit an application to Yarn, it reads the files on HDFS directly. The second case is when fetching results from MapReduce. In map reduce, intermediate and final results are written to HDFS which distributes and replicates them. This is obviously, a source of overhead, and also a part of the strategy for achieving fault tolerance in MapReduce. The major difference between Tez and MapReduce here is that Tez has the ability to stream intermediate results in-between different vertexes, and thus it sacrifices fault tolerance for performance. Multiple tasks are created per logical vertex in order to perform computations in parallel [29]. Also, Tez can combine tasks that would have had to be ran as different jobs with map reduce.
Chapter 6  Evaluation

6.1  Goal
The goal of this project was to look at a subset of the available SQL-on-Hadoop solutions and see if we could reach 20s for typical OLAP queries on a fact table with one billion rows. It was decided to limit the study to Hive-on-Tez, Hive-on-Spark and a simple evaluation of Kylin.

The decision was made to go about this problem by constructing something similar to a benchmark.

The following tests were set up:

- Formats
- Tez vs. Spark
- Partitioning
- Scalability
- Buckets
- Indexing

Unfortunately, it was not possible to get numbers for Kylin, because at the time of evaluation, it was still very unstable.

6.2  Test environment
A Hadoop cluster was deployed on 3 machines, one machine called “master” and two machines called “slave1” and “slave2”. Master runs a hdfs namenode and datanode, yarn resource manager and nodemanager, zookeeper and mapred history server.

Each of the slaves run a HDFS datanode and a yarn nodemanager.

6.2.1  Machine specs
Master:

CPU: 4 Cores, 8 hypertherads, i7-4790S, 3.20 Ghz

RAM: 8 GB

Disks: 1

Each of the slaves:

CPU: 4 Core i5-4460, 3.20 Ghz

RAM: 8 Gb

Disks: 1
6.2.2 Configuration

- **Yarn**
  
  The configuration settings listed in Appendix A was used.
  
  This configuration yielded three yarn containers on each node. In total each machine had four real cores. One core and two GBs of ram was reserved for the Hadoop services on each node. Swapping was turned off for all nodes.

- **Tez**
  
  Only one Tez job was run at a time. The configuration settings from Appendix A were used.

- **Spark**
  
  The settings in Appendix A was derived from Cloudera’s recommendations for spark [30] and from manually testing with different queries. It seemed that spark executors rarely utilized more than one CPU core, even if they were assigned two. For this reason the choice was made to decrease the amount of memory per executor instance and instead increase the number of instances to match the number of yarn containers in the cluster. This appeared to make simple queries run faster.

6.2.3 Hadoop software versions

As Hadoop distribution, Hortonworks HDP 2.2 was installed on all machines. To enable Hive-on-spark, a specially built version of Hive 1.2.0 needed to be used. In addition to this, spark 1.3.0 was configured and built without Hive jars. For the Tez tests, the default Tez version 0.5.2, included in HDP2.2 was used.

6.2.4 Benchmarking software

For running queries, a self-written java program was used that connected to Hiveserver2 using JDBC. The program was made so that it timed queries from the point they were sent to the point where the first column of getResultSet() had returned something. The reason for the later is because some queries would return immediately without a call to getResultSet().

For measuring cluster utilization, Ganglia were set up on each node in the cluster.

6.2.5 Test data

For test data, the standard benchmarks TPC-DS and TPC-H were considered. There is a variant of TPC-H called TPC-H*d, which was adapted for analytical queries by Cuzzocrea and Moussa [31]. This was set up using Mondrian files found on a web page allegedly published by Moussa [32], however, the problem with these, where that they were not adapted for the quantity of data defined in the problem definition. Mondrian got “result set too large” for larger amounts of data using these cube definitions. Therefore it was concluded that these tests were not representative for the use cases examined by this study.

Also, a variant of the TPC-H benchmark is already performed on regular basis by professional organizations such as Amplab [33], however, it would be a challenge to try to replicate it and make the results comparable using the given test environment.
Rather than modifying the TPC-H or DS even further (Hive currently does not support running TPC without rewriting the queries considerably), the decision was made to use a minimal test dataset in order to be able to perform more specifically targeted tests. It is also hard to know how relevant the TPC performance will be for the kind of queries that Mondrian performs.

**Diagram over test data tables**

![Diagram of data tables](image)

**Figure 7**

**Measures (aggregated values):**

- Profit (Sum income – Sum costs)
  
  Computed on the fly by Mondrian

- Sum(Costs)

- Sum(Income)

**Partitions:**

Partitions where used in some of the tests. The fact table was then partitioned on the year column.
Dataset sizes:

<table>
<thead>
<tr>
<th></th>
<th>TEXTFILE</th>
<th>SEQUENCEFILE</th>
<th>ORC (Snappy)</th>
<th>PARQUET</th>
<th>RCFILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of partitions</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Fact table partition size for year 2012:</td>
<td>4.17Gb</td>
<td>6.28Gb</td>
<td>0.90Gb</td>
<td>1.43Gb</td>
<td>2.62Gb</td>
</tr>
<tr>
<td>Fact table partition size for year 2015:</td>
<td>4.33Gb</td>
<td>6.43Gb</td>
<td>0.90Gb</td>
<td>1.43Gb</td>
<td>2.61Gb</td>
</tr>
<tr>
<td>Fact table total size:</td>
<td>28Gb</td>
<td>39Gb</td>
<td>4.8Gb</td>
<td>8.0Gb</td>
<td>17Gb</td>
</tr>
<tr>
<td>Time dimension table size:</td>
<td>28kb</td>
<td>52kb</td>
<td>0.7kb</td>
<td>10kb</td>
<td>15kb</td>
</tr>
<tr>
<td>Div dimension table size:</td>
<td>0.8kb</td>
<td>1.2kb</td>
<td>1.0kb</td>
<td>1.2kb</td>
<td>0.8kb</td>
</tr>
</tbody>
</table>

For a fact table with 1000 000 000 rows. Numbers are in kibibytes and gibibytes respectively.

OLAP Queries:

Mondrian generates SQL queries from MDX queries. One MDX query might result in one or more SQL Queries depending on what currently is in Mondrian’s cache. Thus results that have been previously collected can be, and are in practice, combined with the results of new SQL queries in order to generate a result set. Thus, its not really possible to say with certainty exactly what queries will be generated for each MDX query. The same MDX query might also be implemented in several different ways in SQL.

For example the following MDX query applied in JPivot:

```
select NON EMPTY {[Measures].[Costs], [Measures].[Profit], [Measures].[Income]} ON COLUMNS,
NON EMPTY {([Time].[2012].[2].[3], [Division].[All Divisions])} ON ROWS from [testcube]
```

yields the following sequence of SQL requests by Mondrian:

```
select time.year as c0
from default.test_time_dim time
where time.year = 2012 group by time.year
order by ISNULL(c0) ASC, c0 ASC;
```

```
select time.month as c0
from default.test_time_dim time
where (time.year = 2012) and time.month = 2
order by time.month
order by ISNULL(c0) ASC, c0 ASC;
```

```
select time.day as c0 from default.test_time_dim time
where (time.month = 2 and time.year = 2012) and time.day = 3
roup by time.day
```
order by ISNULL(c0) ASC, c0 ASC;

select count(distinct year)
from default.test_time_dim;

select count(distinct month)
from default.test_time_dim;

select count(distinct day)
from default.test_time_dim:

select time.year as c0, time.month as c1, time.day as c2, sum(fact.costs) as m0, sum(fact.income) as m1
from default.test_time_dim time, default.test_fact fact
where fact.time_id = time.time_id and time.year = 2012 and time.month = 2 and time.day = 3
group by time.year, time.month, time.day;

What exact queries are sent depends on what Mondrian has in its cache.

The following SQL queries were selected for benchmarking:

- **agg1**
  
  ```
  select sum(income), sum(costs)
  from test_fact;
  ```

  JPivot usually does some kind of simple aggregation when it starts up (because that is the default view).

  What exact aggregation is done depends on the cube definition. This query was included just to symbolize a simple aggregation touching only one table.

- **agg2**
  
  ```
  select sum(id), sum(time_id), sum(income), sum(costs)
  from test_fact;
  ```

  Similar to agg1, but with aggregation on twice as many columns. The purpose of this is to get an idea how the number of aggregations affect query performance. The hypothesis was that this would have little or no effect on query performance.

- **mondrian_query1, (mq1)**
  
  ```
  select time.year as c0, time.month as c1, sum(fact.income) as m0
  from default.test_time_dim time, default.test_fact fact
  where fact.time_id = time.time_id and time.year = 2015
  group by time.year, time.month;
  ```

  Simple group by query on the dataset generated by Mondrian. This query computes a sum aggregation over the income field and does an implicit join with the time dimension table. For this query, the entire fact table will be scanned since the query does not make use of the partitioned column in the where clause.

- **mondrian_query1_modified (mq1_m)**
  
  ```
  select time.year as c0, time.month as c1, sum(fact.income) as m0
  ```
from default.test_time_dim time, default.test_fact fact
where fact.year=2015 and fact.time_id = time.time_id

group by time.year, time.month;

This is a hand modified version of mondrian_query1, which only touches one partition of the fact table when predicate pushdown is used. This should make a huge difference in performance when using partitioned tables.

- mondrian_query2 (mq2)

```sql
select time.year as c0, time.month as c1, time.day as c2, d.name as c3, sum(fact.costs) as m0,
        sum(fact.income) as m1
from default.test_time_dim time, default.test_fact fact,
     default.test_div_dim d
where fact.time_id = time.time_id and time.year = 2012 and time.month = 1 and time.day = 1 and
    fact.division_id = d.division_id
group by time.year, time.month, time.day, d.name;
```

A more complex group by involving joins of all tables. Again, it does not make use of partitions.

- mondrian_query3 (mq3)

```sql
select time.year as c0, time.month as c1, time.day as c2, d.name as c3
from default.test_time_dim time, default.test_fact fact, default.test_div_dim d
where fact.time_id = time.time_id and fact.division_id = d.division_id and (time.day = 3 and
                      time.month = 2 and time.year = 2012)
order by ISNULL(c0) ASC, c0 ASC, ISNULL(c1) ASC, c1 ASC, ISNULL(c2) ASC, c2 ASC,
        ISNULL(c3) ASC, c3 ASC;
```

Group by and join of all tables without aggregations and an ordered result set.

- count

```sql
select count(*) from test_fact
where year=2015;
```

Count of the number of rows for year 2015.
Properties for queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Tables touched</th>
<th>Partitions touched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agg1</td>
<td>1</td>
<td>all</td>
</tr>
<tr>
<td>Agg2</td>
<td>1</td>
<td>All</td>
</tr>
<tr>
<td>Mq1</td>
<td>2</td>
<td>All</td>
</tr>
<tr>
<td>Mq1_mod</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Mq2</td>
<td>3</td>
<td>all</td>
</tr>
<tr>
<td>Mq3</td>
<td>3</td>
<td>all</td>
</tr>
<tr>
<td>count</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

6.3 Runs

The queries where executed in runs of 30, once for each format, and with a partitioned and a non-partitioned fact table. The reason for this low sample size is because larger numbers took too long time when testing with a large fact table (1 billion rows).

When testing the ORC format, vectorization was enabled using `hive.vectorized.execution.enabled = true;` before each run. For all jobs `hive.auto.convert.join.noconditionaltask = true;` was issued before each run.

6.4 Steps taken to avoid caching

It was observed that repeated queries resulted in much faster responses after a few runs. This seems to be due to some kind of caching on the client or server. In order to prevent this, and get more consistent test results over runs, the JDBC connection to the database was closed and OS buffers where flushed on all machines in between runs using a shell script which cleared the buffers over SSH. After this modification to the benchmark, repeated queries had roughly the same latency, and the standard deviation decreased for all queries. This means that whatever latencies or other external forces that affected the query performance during this test were fairly constant, at least between runs of the same query.

6.5 Differences between Spark and Tez on Yarn

Spark and Tez behave a bit differently when run on yarn. Tez uses sessions, and can be configured with a long keep-alive timeout, as previously described. As it turns out, the Hive-on-Spark development has not gotten this far yet. In the test environment, a new spark instance was scheduled on the Yarn job queue every time a new connection was made to Hiveserver2. This introduces a startup delay for Spark, however it was verified that this delay was not measured in the tests. The tests measure the time from when the query is sent, to the point where the first column of the resulted is delivered. Before this, a query with settings is sent. Thus, Spark has already responded to the settings query, when the timer is started and the measured query is sent, thus, we know that spark is ready to receive queries at this
point. However, one could argue that a “warm” test would be more realistic. When spark is restarted every time, any type of in memory caching becomes impossible, and this effectively ensures that everything has to be read from disk every time.

One of the reasons why things currently are this was is probably the fact that hive-on-spark is planned to run on LLAP (Live Long and Process) Yarn containers in the future [14]. LLAP is currently in development.

6.6 Indexing
Indexing was tested using compact indexing in Hive with the following options enabled:

\[
\text{set hive.optimize.index.filter=true;}\
\]
Enable automatic use of indexes.

\[
\text{set hive.optimize.index.autoupdate=true;}\
\]
Update stale indexes automatically.

\[
\text{set hive.optimize.index.filter.compact.minsize=0;}\
\]
The minimum size in bytes a table needs to have for indexes to be used.

It is unknown whether compact.minsize also affects bitmap indexes or not. It was set to 0 in order to use the index at all times, if available.

6.7 Bucketing
Bucketing was tested using clustering and sorting of the time_id column on both the fact table and the time dimension table. 32 buckets were used for the fact table and 1 for the time_dim table. The tables were sorted ascending on time_id.

The following (non-default) options where set:

\[
\text{set hive.auto.convert.sortmerge.join=true;}\
\]
Automatically preform sort-merge joins.

\[
\text{set hive.auto.convert.sortmerge.join.bigtable.selection.policy = org.apache.hadoop.hive.ql.optimizer.TableSizeBasedBigTableSelectorForAutoSMJ;}\
\]
This controls how Hive selects the big table for sort-merge joins. This option defaults to partition based big table selection. The result of that is that SMB joins won’t be performed if the tables are not partitioned.

\[
\text{set hive.optimize.bucketmapjoin = true;}\
\]
Try bucket map-joins.

\[
\text{set hive.optimize.bucketmapjoin.sortedmerge = true;}\
\]
Try sorted bucket merge map-joins.

\[
\text{set hive.input.format = org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;}\
\]
Sets input format. The default is CombineInputFormat.
set hive.enforce.bucketing=true;

Bucketing is enforced when inserting into tables.

set hive.enforce.sorting=true;

Sorting is enforced when inserting into tables.

set hive.convert.join.bucket.mapjoin.tez = true;

Automatically convert map-joins into bucket mapjoins when Tez is used as engine for Hive.

Attempts to produce SMB joins were also done with a different dataset with two large tables with two columns; key and value. Random values for keys and values were used.
Chapter 7  Results

7.1  Formats, Partitioning and Tez vs. Spark

The results below are for a fact table size of one billion rows. The time in seconds is a mean of 30 runs. The black bars at the top of the graphs represents the standard deviation.

Figure 8 shows the difference between the mean query times for each format, using mondrian_query1_modified on a fact table without partitions. The query therefore results in a full table scan over 1 billion rows.

Figure 9 shows the same query on a partitioned fact table. Note that the query was modified to take advantage of the partitions. We also see here that the difference in time for ORC between using and not using partitions for this query are in the 20 s interval, while the difference is a lot higher for the simpler Textfile and Sequencefile formats.
ORC with vectorization enabled

Figure 10 and 11 shows the ORC format with vectorization enabled. Vectorization gives ORC a huge advantage over the other formats. Spark seems to be slightly slower than Tez in the majority of cases. This could possibly be attributed to the fact that Spark was restarted every time we ran queries in this evaluation. Thus, the spark timings are “cold”. Possibly they could be better if warm timings was used.

Only slight performance gains can be seen from partitioning. The Tez timings for Mondrian_query1_modified (mq1_m) is 31.2 seconds (sd 1.42) for the partitioned dataset and 36.3 seconds (sd 1.69) without partitions.

The count query is still the only query that is faster than 20 seconds. With Tez, it achieves 17.5 seconds (sd 1.48) unpartitioned with and 16.7 seconds (sd 1.35) with partitions.

We note that the less complex query mq1 which only involves one join, performs worse than mq2 and mq3 which involve 2 joins.
ORC with vectorization disabled

Figure 12 and 13 shows the ORC format with vectorization disabled. The performance for mq1_m without partitions was measured to a mean of 63 seconds (sd 1.63) and 44.2 (sd 1.52) with partitions.
Figure 14 and 15 shows the Parquet format with and without partitions. There is a major difference between parquet and ORC, and that is that Hive does not support vectorization for Parquet.

It is worth noting that the standard deviation is quite high for Spark in some of these tests. If the margin of error is set to 2 standard deviations, it is not possible to say that Spark or Tez is faster here, since the difference is too small and the margin of error too vide.

Figure 16 and 17 shows the RCFile format. Like ORC it is column oriented. Tez seems to be much faster when used with partitions for this format.
7.1.1 Effects of vectorization on CPU usage

**Figure 18**

![CPU usage with vectorization](image)

**Figure 19**

![CPU usage without vectorization](image)
Figure 18 and 19 show the CPU usage during an entire test run of the ORC format with all queries. In figure 18, vectorization is enabled, and in figure 19 it is disabled. The measurements in the figures were taken every 15 seconds. We see that Tez and Spark seem to consume an almost equal peak amount of CPU time in both cases. However, there is great difference in the CPU utilization between using vectorization and not using vectorization.

The agg1 and agg2 queries are run first in the test, with vectorization, we see that these queries consume less CPU time than the later queries which involve joins.

7.2 Scalability

Figure 20 shows the same dataset (30 million rows) against the number of nodes scaled up from 1 to 3. Figure 21 shows an expanding dataset, starting with 10 million rows and increasing the number of rows with 10 million for each new node added. We see that the time it takes to process this increasing dataset is more or less constant as nodes are added. The query used in the test was mq1, and the format was ORC. Note that vectorization was not enabled here, which probably makes queries slightly slower. This should not affect scale up though.
7.3 Buckets

It was observed that Tez does not produce SMB joins even with the options specified in chapter 5. For MapReduce it could be observed that an SMBJoinOperator was used in the query plan for the same data and with the same options. Tez (the version used in hdp2.2) plans an ordinary map-join with the same options enabled.

Figure 22

Figure 22 shows the measurements for the various formats with bucketing enabled. *Note that the scale is different from previous graphs.* Parquet could not be tested since queries kept failing with `java.lang.IllegalArgumentException` when using buckets. Measurements were done on Tez. No partitions were used here.
Figure 23 shows the ORC format with buckets. The performance of mq1_m was measured to a mean of 43.8s (sd: 5.35). This is worse than the performance obtained without buckets. Measurements were done on Tez.
7.4 Indexes

It was observed that Tez 0.5 does not utilize the indexes created with “CREATE INDEX” in Hive.

The following query was tried:

    Select * from test_fact where id=50000000

on a fact table of 1 billion rows, in TEXTFILE format, which had an bitmap index built on it, Tez did not involve the index in the query stage plan. However, when the same query was run with MapReduce, MapReduce did a scan of the index table in stage1. In addition to this, the building of bitmap indexes for ORC, PARQUET or the RCFILE format did not succeed for a 1 billion row fact table. Index rebuilds kept failing with null pointer exceptions. Same thing happened when the index was built with MapReduce.

According to the documentation `hive.optimize.index.filter=true;` should also be set in order to enable format specific indexes with predicate pushdown [34]. This was tried, however, no improvements in timings were observed.

7.5 Order of input data

The performance of the ORC format seems to be independent on the order of input data. A series of 100 million sorted integers was inserted into a table. The performance was measured over 100 where queries and then this was compared to a shuffled version of the input data with the same query. The result was that no significant difference in performance could be observed between the two.
Chapter 8  Conclusions

- It could not be demonstrated that less than 20 second performance could be achieved for all queries with the given setup and dataset, however it was demonstrated that less than 50 seconds could be achieved for all queries.

- The ORC format appears to be the best performing format out of the formats tested.

- In the majority of cases, Hive-on-Tez delivers better performance than Hive-On-Spark.

- We have shown that partitions contribute to better performance when used correctly with predicate pushdown.

- It could not be observed that order of input data significantly affects the performance of the ORC format.

- Scaling is linear for 1-3 nodes.

- Any performance gains from manual indexes could not be observed.

- Any performance gains from bucketing could not be observed.

- Kylin is not production ready at the time of writing.
8.1 Discussion

As stated in the conclusions, it could not be demonstrated that a performance better than 20 seconds could be achieved for all queries with the given setup. However, one must bear in mind that the machines used were not the best, but rather the most affordable. An upgrade to for example SSD hard drives could very well change the outcome of the evaluation. It is also worth noting that the amount of data used in the test is still quite small. If we assume linear scaling, as suggested by the results, we would have to deploy 6 equivalent machines to achieve the same performance for 2 billion rows. That is still no more than 10 gigabytes of data using the ORC format, or about 46 gigabytes using the Textfile format. However, if we always limit ourselves to scanning small partitions, and make sure queries are correctly formatted to enable Hive to do predicate pushdown, there should not be a problem to do this for a much larger dataset as long as partitions themselves are small enough. Adding more columns to any of the columnar formats should not affect performance that much either. After all, the point of columnar formats is to only scan the columns that we actually use. If queries only touch a few of the columns in the table, then performance should not be severely impacted.

Limiting full table scans is a major challenge in all database systems, and this is also the goal of indexing and partitioning. The problem with OLAP style queries is that most of them requires a full table scans, or at least full partition scans. The use of indexes will most likely make little improvement on the performance of these queries. If our results depend on aggregations over the entire dataset, then only pre-computation or caching of these aggregations can probably help us improve performance if we cannot scale the cluster horizontally.

We saw in the results that in some instances mq1 preformed worse than mq2 and mq3. We suggest that the reason behind this is because Hive is able to prune more rows for these queries than for mq1, which makes the table scan of the fact table faster. The effect seems to be amplified by the use of partitions.

Hive only supports vectorization for the ORC format. As stated in the results, vectorization doubled the performance for the Mondrian_query1_modified query. This suggests that, while we do have to worry about the size of the datasets we're scanning, there is still huge performance gains to be achieved from optimizing the query engine.

The performance of Spark was slightly worse than the performance for Tez. This is consistent with other evaluations done by Hortonworks [35]. It is worth noting that the development of Hive-on-Spark has not yet reached its final release. However, unlike in the Hortonworks evaluation, this evaluation did not show that Spark consumes more CPU time than Tez.

The use of buckets did not deliver any performance improvements; we rather saw a slight performance decrease. Unlike MapReduce Tez does not seem to perform bucket joins judging from the query plans, despite the fact that bucket join optimizations have been enabled. If this is intentional, or if it just means that bucket join optimizations are not implemented in the version of Tez being tested, is left unsaid.

Potentially, bucketing could improve performance on star schemas where a small portion of the fact table is joined with a dimension table. In that case, only the data contained in the buckets matching the join key would need to be touched. A possible problem with bucketing however, is that it might interfere with ORC striping and the size of those stripes, and this could impair performance for the ORC format.
Indexes in Hive seem to still be a quite experimental feature. As mentioned in the results section, building of indexes failed for all formats except TEXTFILE. Tez does not seem to use the indexes created with “CREATE INDEX” either.

Kylin was investigated during the initial part of the project. However, it was abandoned when the building of simple cubes did not work correctly on the test setup. Support for aggregation types was also limited. It was concluded that Kylin was not production ready at the time.

As of June 2015, Kylin only supports MOLAP. This means that building large cubes is subject to the curse of dimensionality as mentioned in chapter 4. Thus, Kylin is currently not an alternative if you want to be able to answer any kind of MDX query on large data set. However, HOLAP functionality is on the roadmap, and is planned for 2016 [36, p. 32]. The problem with Kylin today is that it is still quite immature as a product. There are a lot of bugs and flaws in the engine itself that needs to be addressed. It is not possible to use Tez to build cubes, only MapReduce is currently supported. However, Spark support is on the roadmap.

Other than Kylin, hope for OLAP on Hadoop lies with the promised Materialized views for Hive in the Stinger next initiative.

Hive-on-Spark still has some problems that need to be resolved as well. During my tests spark queries could rare cases fail with null pointer exceptions for not apparent reason. As stated before, spark is probably nor supposed to be run with the current Yarn job queue, but rather in a separate LLAP process. This will probably help spark to better utilize memory for caching, not just for computation.

8.2 Further work

Besides contributing to improving the Tez, Hive-on-Spark and Kylin projects there are some other things that could possibly be done:

- Evaluating other SQL on Hadoop products, such as Impala, Tajo, or Presto for ROLAP applications.
- Possibly, integrating Mondrian with Kylin, having them use function together and running on the same tomcat instance.
- Developing MDX / XMLA support for Kylin, so that Kylin can be used with standard Bi-Tools without the need for Mondrian.
- Make Mondrian partition aware. Make sure that you can configure it so that it takes advantage of partitions.
- Benchmark Kylin.
- Preform an evaluation on more nodes.
- Improve Mondrian’s cache invalidation API.
Chapter 9  Advice to Omicron

9.1  Pure ROLAP solutions

If the decision is made to continue pursuing a pure ROLAP solution on Hadoop I would suggest looking at some other products: Impala, Tajo or Pesto and see how they compare to Hive-on-Tez using the ORC format with vectorization enabled. Possibly, these products are optimized for performance rather than being general solutions for SQL on Hadoop. What these tools really do is helping you get more out of your hardware.

The problem with the pure ROLAP approach is that it requires you to limit the amount of data you scan on each machine in order to receive adequate performance, and there is really two ways of thing this:

- **Projections on the dataset**
  
  This encompasses things like partitioning and selecting a smaller part of the dataset in order to limit the amount of data being scanned.

- **Buying more hardware**

  If you want to scan 3 billion rows at the same amount of time as was done in this study, you are obviously going to need the equivalent of 10 machines, similar to those used here. That is, if we assume linear scaling, and from the observations made in this study, it seems to be safe to do that for a small amount of machines. Note that these amounts of data are still far from being “Big Data”. Just like the official Hadoop documentation, I would also recommend using a dedicated namenode-host.

  *Remember that when working with row-columnar formats, adding more columns to a table does not necessarily impact performance, while adding rows certainly does. This means that raw performance should be measured over number of rows scanned, not on dataset size.*

When using the ORC format I would recommend to use Snappy compression, this saves space, however judging from my informal observations, turning it on does not seem to affect the runtime of queries. This is not an official result of this thesis, but I would make sure to have good grounds for turning compression off and not just doing it because some documentation on the Internet said it improves performance.

Bucketing of tables should probably be avoided, it does not seem to deliver much value right now, also, it seems to slow things down, and I you can see in the results.

9.2  HOLAP with Kylin

As previously mentioned Kylin has HOLAP functionality on the roadmap for 2016 [36, p. 32]. The idea is to be able to materialize partial-cubes on the fly as queries come in. This could make Kylin a lot more useful, and help avoid the effects of the “curse of dimensionality”. I would recommend going back and testing Kylin at a later time.

The benefit of Kylin is that pre-computation minimizes the amount of hardware needed in order to achieve On-Line performance for queries.
9.3 Druid

Another interesting project that I have not mentioned previously in the report is Druid. Durid is a mature open source OLAP engine for Hadoop that focuses on real-time performance. Real-time here means that aggregates are updated as soon as new data arrives. The Kylin team is planning a similar feature called "streaming cubes". This eliminates the need to rebuild the cube periodically, which is of course a huge advantage. The main drawback (or advantage, depending on the way you look at it) is that Durid does not support SQL or MDX. Instead it has its own query API based in JSON [37].

Druid could be an alternative when you want to build a web-based frontend with aggregates that are updated in real time. It is probably not the most convenient solution for traditional Bi-Tools however.
Appendix A

Configuration settings for HDP 2.2

Using cores=4 memory=8GB disks=1 hbase=False:Profile: cores=4 memory=6144MB reserved=2GB usableMem=6GB disks=1
Num Container=3
Container Ram=2048MB
Used Ram=6GB
Unused Ram=2GB

 ***** mapred-site.xml *****
mapreduce.map.memory.mb=2048
mapreduce.map.java.opts=-Xmx1536m
mapreduce.reduce.memory.mb=2048
mapreduce.reduce.java.opts=-Xmx1536m
mapreduce.task.io.sort.mb=768

 ***** yarn-site.xml *****
yarn.scheduler.minimum-allocation-mb=2048
yarn.scheduler.maximum-allocation-mb=6144
yarn.nodemanager.resource.memory-mb=6144
yarn.app.mapreduce.am.resource.mb=2048
yarn.app.mapreduce.am.command-opts=-Xmx1536m

 ***** tez-site.xml *****
tez.am.resource.memory.mb=2048
tez.am.java.opts=-Xmx1536m

 ***** hive-site.xml *****
hive.tez.container.size=2048
hive.tez.java.opts=-Xmx1536m
hive.auto.convert.join.noconditionaltask.size=402653000

Spark settings

spark.executor.memory=1664m
spark.yarn.executor.memory.overhead=384m
spark.executor.instances=9
spark.executor-cores=1
spark.master=yarn-cluster
spark.serializer = org.apache.spark.serializer.KryoSerializer
Bibliography


