CONSEQUENCES OF CONVERTING A DATA WAREHOUSE BASED ON A STAR-SCHEMA TO A COLUMN-ORIENTED-NOSQL-DATABASE

Master Degree Project in Data Science
One year Level ECTS
Spring term 2018

Rebecca Bodegård Gustafsson

Supervisor: Mikael Berndtsson
Examiner: Manfred Jeusfeld
Abstract

Data warehouses based on the relational model has been a popular technology for many years, because they are very reliable due to their ACID-properties (Atomicity, Consistency, Isolation, and Durability). However, the new demands on databases today due to increasing amounts of data and data structures changing do mean that the relational model might not always be the optimal choice. NoSQL is the name of a group of databases that are less bound by schemas and are therefore more scalable and easier to make changes in. They are also adapted for massive parallel processing and are therefore suited for handling large amounts of data. Out of all of the NoSQL databases column-databases are the most like the relational model since it also consists of tables. This study has therefore converted a relational data warehouse based on a STAR-schema to a column-oriented-NoSQL-database and evaluated the implementation by comparing query-times between the relational data warehouse and the column-oriented-NoSQL-database. Scrambled economical data from a business in Sweden has been used to do the conversion and test it by asking a few usual queries. The results show that the mapping works but the query-time in the NoSQL-database is significantly longer.
# Table of content

1. Introduction 1

2. Background 4
   2.1 Relational databases 4
   2.2 Data warehouses and the STAR-schema 5
   2.3 NoSQL databases 7
      2.3.1 Key-value stores 7
      2.3.2 Document databases 8
      2.3.3 Column databases 8
      2.3.4 Graph databases 9
   2.4 Data warehouse evolution 10
   2.6 Apache HBase 11
   2.7 Ways to query HBase using SQL-like language 12
   2.8 Previous research on converting relational data warehouses to NoSQL 13

4. Method/research design 15

5. Implementation 17
   5.1 The relational data warehouse 17
      5.1.1 The STAR-schema 17
      5.1.2 Implementation of the relational data warehouse 18
   5.2 Mapping styles 19
   5.3 Transformation of STAR-schema to column-oriented-NoSQL-structure 20
   5.4 Implementation of HBase 21
   5.5 Queries 22

6. Analysis and results 24
   6.1 Evaluating the queries 24
   6.2 Evaluating the implementation 26

7. Conclusion 29
   7.1 Contributions and conclusion 29
   7.2 Future work 30

References 31

Appendix 1. Queries and results relational data warehouse 34
Appendix 2. Queries and results Apache Hive/Apache HBase 37
Tables

Table 1. The differences between operational databases and data warehouses 5
Table 2. Example of a key-value-store 7
Table 3. Example of an HBase table 11
Table 4. HBase vs RDBMS (Tutorialspoint, 2018) 11
Table 5. Translation of the dimensions from Swedish to English 18
Table 6. Structure to be implemented in HBase 20
Table 7. Example of output from the questions 22
Table 8. The results from the first type of questions 24
Table 9. The results from the second type of questions 25
Table 10. A comparison of the results to previous studies 28

Figures

Figure 1. Example of a relational model 4
Figure 2. Example of a STAR-schema 6
Figure 3. STAR-schema of the economical model 17
Figure 4. The first type of query in HiveQL 22
Figure 5. The first type of query in SQL 22
Figure 6. The second type of query in HiveQL 23
Figure 7. The second type of query in SQL 23
Figure 8. HBase compared to RDBMS for the first type of queries 24
Figure 9. HBase compared to RDBMS for the second type of queries 25
1. Introduction

For many years relational databases (RDBMS) have been the dominating way to store data (Padron-McCarthy & Rysch, 2005). They are often used in data warehousing which means that data from many different systems are combined and stored in a large relational database. The combination of the data is then used to draw new insights. Data warehouses are usually structured after a so called STAR-schema where there are fact-tables that are connected to dimensional tables using foreign keys (Ponniah, 2011).

Relational data warehouses are very reliable because of their ACID-properties (Atomicity, Consistency, Isolation, and Durability). They are, however, not easy to scale out or make changes in (Chung, Lin, Chen, Jiang & Chung, 2014; El Alami & Bahaj, 2016). The changing of a column-type or the adding of another column might mean quite a lot of work even though the change is minor, especially when working with a lot of data.

Another problem with relational data warehouses is when data should be divided between shards to enable massive parallel processing (MPP), which becomes more and more important as the amount of data increases (El Alami & Bahaj, 2016). There are some database-solutions on the market that are relational but still enable MPP (Postgres-XL, 2018). These solution are, however, still schema bound so the issue with the schema changing is still a problem.

Previous research on data warehouse evolution does not take into account that the requirement in terms of the scope and structure of data and meta data has increased. Most of the research that have been conducted focus on the relational model which is not convenient to support historical evolution of the logical structure of the data warehouse schema (Subotic, Poscic & Jovanovic, 2014).

NoSQL is the name of a group of databases less bound by schemas and are therefore more scalable and easier to make changes in. They are also adapted for MPP and handles unstructured data better than relational databases (El Alami & Bahaj, 2016).

The new requirements of databases in order to adapt to changing data and the need for massive parallel processing means that NoSQL is a technology on the rising. The fact that a lot of companies have their data in relational data warehouses means that they might want to look into moving to a NoSQL solution (Chevalier, El Malki, Kopliku, Teste and Tournier, 2015b). The problem then will be how to map their current STAR-schema to the new NoSQL-structure.
Out of all the NoSQL databases column-databases are the most similar to the relational model since it also consists of tables. It is also good for sorting and conversion, operations which are often performed in a data warehouse (Redmond & Wilson, 2012; Lee & Zheng, 2015).

Some research have been conducted on conversion between relational databases and NoSQL-databases (El Alami & Bahaj, 2016; Chung et al., 2014; Kuderu & Kumari, 2016), but this research has mostly focused on production databases and not data warehouses. Data warehouses have other requirements than operational databases which means that this research might not be applicable (Ponniah, 2011).

There are also some tools on the market to migrate data from relational databases to a column-database (HBase), but they have a problem dealing with foreign key dependencies (Zhao, Li, Li & Lin, 2014).

There are a few papers focusing on transforming relational (multidimensional) data warehouses to NoSQL (Chevalier, El Malki, Kopliku, Teste & Tournier, 2015a; Chevalier et al., 2015b; Yangui, Nabli & Gargouri, 2016). The guidelines in the previous research have, however, only been tested in controlled environments using experiments. This study therefore aims to map a STAR-schema to a NoSQL-structure and implement it in a less controlled environment. The study will also attempt to compare the relational data warehouse to the NoSQL implementation, which previous research have not done. The research question that this study aims to answers is:

*What are the consequences of converting a data warehouse based on a STAR-schema to a column-oriented-NoSQL-database?*

The study tried to map a STAR-schema of a relational data warehouse into a column-oriented-NoSQL-database by first mapping the conceptual STAR-schema into a logical-column-oriented-structure. The logical structure was then implemented in a column-oriented-database called HBase. HBase was chosen because it is a popular technology (based on Google’s bigtable) and is similar to relational databases, because it consists of tables (Yangui, Nabli & Gargouri, 2016).

The scope of the implementation was to convert one fact-table and its dimensions, containing economical data. An economical model was chosen because it is very common to start with economical data when implementing a data warehouse. Economical data is often quite structured and doesn’t require that much cleaning before it can be used. This project therefore focuses on transforming a STAR-schema for an economical relational model in a data warehouse.
To evaluate the result of the mapping and implementation of HBase a few important queries in the relational data warehouse were asked in both the HBase-implementation and the relational data warehouse to compare the response-time. Also the process of transforming the STAR-schema and implementing of the logical model in HBase was taken into account in the evaluation.

The results of the study show that the STAR-schema can be mapped to the logical column-oriented structure but the queries take a lot longer in the column-oriented data warehouse than in the relational data warehouse. The query-times for the column-oriented data warehouse seem to compare to the previous studies that have used similar mapping methods (Dehdouh, Bentayeb, Boussaid & Kabachi, 2015; Chevalier et al., 2015b). The study shows that the consequences of converting a relational data warehouse based on a STAR-schema to a column-oriented-NoSQL-database is longer query-times.
2. Background
This chapter explains how relational databases as well as the different categories of NoSQL-databases work. It also presents previous research on mapping relational data warehouses to column-oriented databases and the research area of data warehouse evolution.

2.1 Relational databases
The relational model is the dominating model of today's database management systems. It stores data in several different tables which consist of rows and columns. These tables are connected by unique keys (primary key, pk) on each row (Lee & Zheng, 2015; Redmond & Wilson, 2012). These tables are normalised to reduce redundancy and minimised storage space. Normalisation is about finding out the relationships in each table. After this process rows in one table can be linked to rows in other tables by a unique key that is called a foreign key (fk). See an example of the relational model in figure 1. Relational databases can be queried using sequel query language (SQL) which allows JOIN-operations to combine the different tables (Lee & Zheng, 2015).

![Figure 1. Example of a relational model](image)

Key features of databases based on the relational model (RDBMS) are Atomicity, Consistency, Isolation, and Durability (ACID).
2.2 Data warehouses and the STAR-schema

Data warehouses are databases that combine data from different sources and use this to draw new insights. A data warehouse is used to provide strategic information and therefore needs to support complex queries. The data stored is often summarised and derived to more easily support complex queries (Ponniah, 2011). Table 1 summarises differences between data warehouses and operational databases.

<table>
<thead>
<tr>
<th></th>
<th>Operational</th>
<th>Data warehouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data content</td>
<td>Current values</td>
<td>Archived, derived, summarised</td>
</tr>
<tr>
<td>Data structure</td>
<td>Optimised for</td>
<td>Optimised for complex queries</td>
</tr>
<tr>
<td></td>
<td>transactions</td>
<td></td>
</tr>
<tr>
<td>Access frequency</td>
<td>High</td>
<td>Medium to low</td>
</tr>
<tr>
<td>Access type</td>
<td>Read, update, delete</td>
<td>Read</td>
</tr>
<tr>
<td>Usage</td>
<td>Predictable, repetitive</td>
<td>Ad hoc, random, heuristic</td>
</tr>
<tr>
<td>Response time</td>
<td>Sub-seconds</td>
<td>Several seconds to minutes</td>
</tr>
<tr>
<td>Users</td>
<td>Large number</td>
<td>Relatively small number</td>
</tr>
</tbody>
</table>

Table 1. The differences between operational databases and data warehouses

When storing data in a data warehouse using the relational model a common concept is the STAR-schema. We have a fact-table that contains all of the measures that we want to aggregate, analyse etc. This fact-table also contains foreign keys to what is called dimensions. Dimensions are properties that we might want to aggregate on or in other ways use to analyse the measures in the fact-table. The STAR-schema is basically a relational model with a one-to-many relationship between each dimensional table and the fact table (Ponniah, 2011). In figure 2 OrderTable is the fact-table and OrderDate, ProductTable and CustomerTable are dimensions. NumberOfArticles and TotalCost are measures in the fact-table while ProductID, CustomerID and OrderDate are references to the dimensional tables.
STAR-schemas are great because they are easy for users to understand and suitable for asking queries. The drawback of the model is, however, if one or more of the dimensions are very large. When this is the case it might be better to go with a snowflake schema. What this model does is normalise the dimension. In the example in figure 2 this could mean that the product table would be split up into a categorical dimension and one product dimension in order to reduce redundancy. However, the STAR-schema is usually preferable because its advantages usually outweighs its drawbacks (Ponniah, 2011).
2.3 NoSQL databases

NoSQL stands for Not only SQL and refers to distributed, non-relational database-systems that are not primarily built on tables and generally don’t use SQL for data manipulation. These systems are built to scale horizontally to handle parallel read/write operations over many servers (Moniruzzaman & Hossain, 2013; Cattell, 2010). Key features of a relational database are ACID (Atomicity, Consistency, Isolation, and Durability), but NoSQL-stores instead work with BASE (Basically Available, Soft state, Eventually consistent) (Cattell, 2010; Lee & Zheng, 2015). By giving up some of the ACID-features like consistency the idea is that the NoSQL-store can work faster and scale up more easily (to improve availability).

NoSQL-databases can be classified into four categories (Lee & Zheng, 2015):

1. Key-value databases
2. Document databases (or stores)
3. Column-databases
4. Graph-databases

In the following sections these four categories of NoSQL databases will be presented.

2.3.1 Key-value stores

Key-value databases store items as alpha-numeric keys with associated values. The values (attributes) can be strings or more complex structures like sets and lists. Usually searches can only be performed against the keys and they need to be exact matches (Moniruzzaman & Hossain, 2013; Chen, Mao, Zhang, & Leung 2014; Lee & Zheng, 2015).

<table>
<thead>
<tr>
<th>Key</th>
<th>Values</th>
</tr>
</thead>
</table>
| 1   | Name: AFirstName  
     | Age: 25  
     | City: Skövde |
| 2   | Name: ASecondName  
     | Age: 34  
     | City: Stockholm |
| 3   | Name: AFirstName  
     | Age: 45  
     | City: Skövde |
| 4   | Name: AFirstName  
     | Age: 28  
     | City: Lidköping |

Table 2. Example of a key-value-store
Key-value databases are good for schema-less data (Lee & Zheng, 2015). One example is when there is a need to get values for applications like user-profiles (Moniruzzaman & Hossain, 2013). An example would be if we want to get the name and age of user number 1 in table 2, then we would search for values where key = 1 and we would then get the values needed. The simplicity of this makes it very fast (Chen et al., 2014), but it is not good for more complex queries or aggregations (Redmond & Wilson, 2012).

Examples of key-value databases on the market are: Dynamo (used by Amazon), Voldemort (used by LinkedIn), Redis, BerkeleyDB and Riak (Moniruzzaman & Hossain, 2013).

2.3.2 Document databases

Document-databases are, as the name suggests, designed to managed and store documents. These databases are more complex than the key-value databases because the value column contains semi-structured data. The value column consists of documents that are encoded in XML, JSON or BSON. A single column can include hundreds of attributes and the number and type of attributes can differ from row to row. Both keys and values are searchable in document databases (unlike key-value databases) (Moniruzzaman & Hossain, 2013; Chen et al., 2014; Lee & Zheng, 2015).

Document-databases are good for storing and managing Big Data-size collections of literal documents (text-documents, XML etc.) In general they are also good for storing sparse data (that would require a lot of null values in a RDBMS) (Moniruzzaman & Hossain, 2013; Chen et al., 2014). Examples of document databases on the market are CouchDB (stores values encoded in JSON) and MongoDB (stores values encoded in BSON). Both of these are open source (Moniruzzaman & Hossain, 2013).

2.3.3 Column databases

Column databases are distributed, column oriented data structures with multiple attributes per key. Some column databases are inspired by key-value databases (like the key-value store Dynamo which inspired the column-database Cassandra) but most of them are inspired by Google’s Bigtable, an internal distributed data storage system developed by Google (Moniruzzaman & Hossain, 2013; Chen et al., 2014). The structures are composed of tables and is therefore the most similar to relational databases of its NoSQL siblings. However, instead of having a fixed schema the number of columns may change from row to row. This means that the concept of a table in these types of databases are a lot more flexible and vague than a table in a relational database (Lee & Zheng, 2015). Adding columns to a column-database is quite inexpensive, since it is done on a row-by-row basis. This type of database
can almost be seen as a combination of relational and key-value databases (Redmond & Wilson, 2012).

Column databases are great for exploratory and predictive analysis, distributed data storage, large-scale and batch oriented data processing like sorting and conversion (Moniruzzaman & Hossain, 2013; Chen et al., 2014). Examples of column databases on the market are BigTable and HBase (Moniruzzaman & Hossain, 2013).

2.3.4 Graph databases

Graph databases are structured relational graphs of interconnected key-value pairings. They are similar to object oriented databases because the graphs are represented by an object-oriented network of nodes and relationships between nodes and properties (Redmond & Wilson, 2012; Moniruzzaman & Hossain, 2013). It is the only type of NoSQL-database that works with relations. The focus is visual representation of information which make them more human-friendly than other NoSQL-databases (Moniruzzaman & Hossain, 2013).

Graph databases are useful when the focus is the relationships between the data more than the data itself, because they are optimised for showing relationships and not querying. Examples of this are when showing social networks or pattern detection (Moniruzzaman & Hossain, 2013; Redmond & Wilson, 2012). Some graph databases on the market are: Neo4j, InfoGrid, Sones GraphDB, AllegroGraph and InfiniteGraph (Moniruzzaman & Hossain, 2013).
2.4 Data warehouse evolution

Data warehouse evolution is an old problem-area. The core of the problem is that a data warehouse must always contain the latest information in order to be able to reflect the operational system that it gets its data from. Data warehouse evolution has 3 areas: versioning, evolution and view management (Subotic, Poscic & Jovanovic, 2014).

In schema evolution there is only one schema at the time which means that when changes has to be made the schema is transformed but there is no history. There has been a lot of research in this area but it does not take into account the preservation of history (Subotic et al., 2014).

Schema versioning transfers data from the old schema to the new schema but keeps the old schema. The main problems of schema versioning are:

- slow and expensive transformation and migration of data between different schema versions
- loss of information during the transformation and migration processes
- inefficient adaptation of existing queries and user applications to the new schema version
- inefficient and error prone work with multiple schema versions at the same time
- lack of effective integration, organisation and management of metadata.

View maintenance studies how to effectively maintain views and how these should be built in order to increase performance. This area can be further categorised into two approaches: view adaptation and view synchronisation. In view adaptation the latest structural information is added to the materialised view (to adapt to changes in metadata). In view synchronisation the view is rewritten after changes in data sources (Subotic et al., 2014).
2.6 Apache HBase

Apache HBase is a distributed column-oriented database built on top of the Hadoop file system (HDFS). It is horizontally scalable and is an open source project. The data model of HBase was inspired by Google’s bigtable and is designed to give quick random access to huge amounts of structured data (Tutorialspoint, 2018).

One important feature that HBase provides is column-families. Column families are physically stored together in a distributed cluster making reading and writing faster when only columns from one family is affected. The number of column-families are always the same but it is easy to add another column to a column family. HBase also keeps different versions of each column’s values (marked by time-stamps), so it is possible to go back to previous values if needed (Capriolo, Wampler & Rutherglen, 2012). HBase is also fault tolerant thanks to the use of Hadoop Distributed File System (HDFS) (Tutorialspoint, 2018).

The basic structures of HBase are:
- A table is a collection of rows
- A row is a collection of column families
- A column family is a collection of columns
- A column is a collection of key value pairs

(Tutorialspoint, 2018)

<table>
<thead>
<tr>
<th>Row_id</th>
<th>Column_family</th>
<th>Column_family</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Column 1</td>
<td>Column 1</td>
</tr>
<tr>
<td></td>
<td>Column 2</td>
<td>Column 2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Example of an HBase table

<table>
<thead>
<tr>
<th>HBase</th>
<th>RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schemaless</td>
<td>Schemabound</td>
</tr>
<tr>
<td>Horizontally scalable</td>
<td>Difficult to scale</td>
</tr>
<tr>
<td>Widetables</td>
<td>Small tables, connected by keys</td>
</tr>
<tr>
<td>No transactions</td>
<td>Transactional</td>
</tr>
<tr>
<td>Denormalised data</td>
<td>Normalised data</td>
</tr>
<tr>
<td>Good for semi-structured and structured data</td>
<td>Good for structured data</td>
</tr>
</tbody>
</table>

Table 4. HBase vs RDBMS (Tutorialspoint, 2018)
2.7 Ways to query HBase using SQL-like language

There are two ways to query HBase using SQL-like languages, both part of the Hadoop ecosystem, Apache Hive and Apache Phoenix.

Apache Hive is a software in the Hadoop ecosystem that enables querying and managing large datasets using a SQL-like language called Hive Query Language (HiveQL). What Hive does is translating the HiveQL language into MapReduce (java) code. Hive can translate most queries into MapReduce jobs so that the scalability of Hadoop can be used while still using SQL. Hive is suited for data warehouse applications, where a large data set is maintained and mined for insights, reports, etc. since it does not provide record-level updates, inserts or deletes (Capriolo et al., 2012).

Hive lowers the barrier between using SQL-based relational databases and Hadoop since people who know SQL can learn Hive easily. HiveQL does, however, differ a bit from the familiar SQL versions provided by Oracle, MySQL and SQL Server. HiveQL is most like MySQL out of the previous mentioned versions of SQL (Capriolo et al., 2012).

Apache Phoenix is an open source relational database engine that supports OLTP for Hadoop using Apache HBase as its backing store. It uses standard SQL and has ACID transaction capabilities. It can also work with other Apache products such as Hive, Pig, Flume and MapReduce (Hortonworks, 2018).
2.8 Previous research on converting relational data warehouses to NoSQL

There are some studies on converting STAR-schemas to column-oriented-NoSQL-structures.

Dehdouh, Bentayeb, Boussaid & Kabachi (2015) propose three approaches to implement big data warehouses as a column-oriented-NoSQL-database:

- **Normalised logical approach (NLA):** Different tables for dimensions and facts
- **Denormalised logical approach (DLA):** Same table for dimensions and fact
- **Denormalised logical approach using column families (DLA-CF):** Same table for dimensions and fact using column-families

They have compared the different models by implementing the star schema benchmark data warehouse (SSB) using HBase as the column-oriented DBMS.

The results show that NLA, which uses different tables for dimensions and fact, has slower query times than DLA and DLA-CF. DLA and DLA-CF have about the same query time. DLA-CF only performs better when handling columns in the same column family.

Chevalier et al. (2015b) also propose 3 approaches to convert the relational model of a data warehouse to a column-oriented-NoSQL-database:

- **MLC0:** stores data grouped in a unique column family
- **MLC1:** one column family for each dimension and one dedicated for the fact
- **MLC2:** splits data into multiple tables

They also use HBase to test their mapping using experiments. Data is generated using an extended version of SSB based on their model. The data consists of one fact table containing 4 dimensions (customer, supplier, part and date). The results show that the shattered model (MLC2) is slow at answering queries but uses less disk-space. MLC1 and MLC0 do not show any significant performance differences (Chevalier et al., 2015b).

The same authors also published a paper were they benchmarked an implementation of HBase to an implementation of MongoDB. The guidelines for transforming the data warehouse to the column-oriented model in that paper was was to (Chevalier et al., 2015a):

- Transform each schema into one table
- Transform each fact into one column-family, where each measure is a column
- Transform each dimension into a column-family, where each dimensional attribute is a column

Chevalier et al., 2015b
Yangui, Nabli and Gargouri (2016) propose new rules for transforming multidimensional conceptual models into two different NoSQL-models, column-oriented and document-oriented. They use two different types of transformations: simple and hierarchical. They implement four different data warehouses using Cassandra as a column-oriented-database and MongoDB as a document-oriented-database. The use of Cassandra as a column-oriented-NoSQL-database means that they can use so called super-families and super-columns together with column-families and columns to show hierarchies.

In the simple transformation for the column-oriented-database each fact is transformed into a column-family and each dimension is transformed to a column-family. The name of the dimension or the fact will be the name of the column-family.

In the hierarchical transformation for the column-oriented-database each fact is transformed into a column-family and each dimension is transformed to a super-column-family. Then each hierarchy in the dimension is transformed into a super-column and each attribute of the super-column will be transformed to a column in the super-column.

Both the hierarchical and the simple transformations include putting all of the data in one table.

The results of the experiments show that the document-oriented structure was more efficient when asking queries while the column-oriented one had better loading times.

These previous studies show that query-performance is better when storing all of the data in one table. Using column-families in HBase doesn't make that much of a difference unless the query handles columns in the same column-family.
4. Method/research design

In order to answer a research question, relevant approaches need to be identified (Yin, 2003):

1. Type of question: How, what, where, why.
2. How much control the research has over the situation.
3. Is the focus recent of historical events.

The research question in this study wants to answer what. The data that will be used is actual data (but scrambled) based on a company’s relational model which means that the environment is not fully controlled and it does focus on contemporary events. According to Yin (2003) a less controlled environment could motivate the use of a survey, archival analysis, history analysis and a case-study. Since the use of real-life data will make the environment less controlled this could motivate the use of a case-study (Yin, 2003).

However, since this study will use mapping-strategies from previous studies and compare the implementation to the original relational data warehouse this could also motivate the use of experiments since it focuses more on testing previous research than studying the organisation the data belongs to (Berndtsson et al., 2008).

The scope of the experiment is to convert one fact-table and its dimensions, containing economical data. An economical model was chosen because it is very common to start with economical data when implementing a data warehouse. Economical data is often quite structured and doesn’t require that much cleaning before it can be used. This project therefore focuses on transforming a STAR-schema for an economical relational model in a data warehouse.

To evaluate the results of the mapping and implementation of HBase a few important queries in the relational data warehouse will be asked in both the HBase-implementation and the relational data warehouse to compare the response-time. Also the process of transforming the STAR-schema and implementing the logical model in HBase is taken into account in the evaluation.
The experimental design includes the following steps:

1. **Set up the relational data warehouse and import data:** Examine the data from the relational data warehouse to find out its structure and set up the relational data warehouse. The STAR-schema for the data warehouse will be modelled. After that the data will be exported from the company and imported into a new database. The new environment will be set up to be ready to run the queries.

2. **Define the queries to be asked in both databases:** Frequent queries from the relational data warehouse will be defined.

3. **Choose a mapping based on previous research:** Based on previous research a mapping from a STAR-schema to a logical-column-oriented-NoSQL-structure will be defined.

4. **Transform the conceptual model of the data warehouse to the logical structure for HBase and export it to a csv:** The STAR-schema defined in step 1 will be mapped to a HBase structure using the mapping from step 3. Data will then be transformed into the logical structure.

5. **Set up the column-oriented database and import data:** Install HBase and import data into to HBase. The table/tables from step 4 will be exported to csv-files which can be imported into HBase.

6. **Connect HBase to a program to be able to ask SQL-like queries:** To be able to ask the queries from the relational data warehouse HBase has to be connected to another program that enables the use of SQL to interact with the data.

7. **Ask queries in both the relational data warehouse and HBase to compare performance:** When the data has been imported and the program that enables SQL has been installed queries will be run in both the relational data warehouse and in HBase.

Step 4-6 will first be done with a subset of the data (1000 rows), just to see that the structure works.
5. Implementation
This chapter explains how the relational data warehouse was converted into HBase. It starts out by describing the data and the relational data warehouse. It then explains the mapping that was used and how the data model was mapped to the column-oriented-NoSQL-structure. After that the evaluation is described.

5.1 The relational data warehouse
The data model comes from a business in Sweden. The data is scrambled and used as demo data for a business intelligence product. The data chosen for the experiments was economical data and the fact table consisted of 6 253 058 rows.

5.1.1 The STAR-schema
The data consisted of 12 dimensions. Eight of these were simple dimensions and two of them were hierarchical ones, each consisting of two dimensions (year and project). The dimensions have, however, been treated as simple dimensions to make the model easier to work with. Since the dimensions were in Swedish their names have be translated for better understanding (see table 5).

Figure 3. STAR-schema of the economical model
5.1.2 Implementation of the relational data warehouse

The data was exported into csv-files from the business’s data warehouse. Microsoft SQL server management studio was used as the RDBMS on a windows computer with 16 GB of RAM. The csv-files were exported and imported using the import/export wizard. The result was 13 tables, one fact table and 12 dimensional tables. To optimise the queries the dimensional tables were indexed on the id column (clustered) and the fact-table were indexed on year (clustered).

<table>
<thead>
<tr>
<th>Name</th>
<th>Translation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>År</td>
<td>Year</td>
<td>The year the transaction took place</td>
</tr>
<tr>
<td>Period</td>
<td>Month</td>
<td>The month the transaction took place</td>
</tr>
<tr>
<td>Bolag</td>
<td>Company</td>
<td>If the business consists of several subsidiaries</td>
</tr>
<tr>
<td>Konto</td>
<td>Account</td>
<td>The account the transaction was booked on</td>
</tr>
<tr>
<td>Projekt</td>
<td>Project</td>
<td>The project the transaction was booked on</td>
</tr>
<tr>
<td>Projekttyp</td>
<td>Project-type</td>
<td>The type of project the transaction was booked on</td>
</tr>
<tr>
<td>Ansvar</td>
<td>Responsibility</td>
<td>The responsibility the transaction was booked on, can be a particular section of the company</td>
</tr>
<tr>
<td>Verksamhet</td>
<td>Function</td>
<td>The function the transaction was booked on, can be a particular section of the company</td>
</tr>
<tr>
<td>Aktivitet</td>
<td>Activity</td>
<td>The activity the transaction was booked on</td>
</tr>
<tr>
<td>Objekt</td>
<td>Object</td>
<td>The object the transaction was booked on</td>
</tr>
<tr>
<td>Motpart</td>
<td>Counterpart</td>
<td>The counterpart the transaction was booked on. This keeps track on transaction between different parts of the company.</td>
</tr>
<tr>
<td>Fri</td>
<td>Free</td>
<td>An extra dimension that can be used if needed.</td>
</tr>
</tbody>
</table>

Table 5. Translation of the dimensions from Swedish to English
5.2 Mapping styles

The mapping chosen combines the model called DLA-CF from the paper by Dehdouh et al. (2015) and the model MLC1 from the paper by Chevalier et al. (2015b). Both of these mappings are basically the same.

In both the experiments by Dehdouh et al. (2015) and Chevalier et al. (2015b) using one big table that includes all of the dimensions needed performs better. Using column-families don’t make any difference unless a query only handles columns in the same column-family. Since using column-families doesn’t slow down the queries and could possible lead to some slightly better query-times they will be used in the mapping.

The mapping that will be performed in this study consists of the following steps:
- Transform each schema into one table
- Transform each fact into one column-family, where each measure is a column
- Transform each dimension into a column-family, where each dimensional attribute is a column
5.3 Transformation of STAR-schema to column-oriented-NoSQL-structure

After mapping the STAR-schema, the structure to be implemented in HBase consisted of the column-families in table 6.

<table>
<thead>
<tr>
<th>Column_family</th>
<th>Column</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>year</td>
<td>year_title</td>
</tr>
<tr>
<td>month</td>
<td>month</td>
<td>month_title</td>
</tr>
<tr>
<td>company</td>
<td>company</td>
<td>company_title</td>
</tr>
<tr>
<td>account</td>
<td>account</td>
<td>account_title</td>
</tr>
<tr>
<td>project</td>
<td>project</td>
<td>project_title</td>
</tr>
<tr>
<td>project_type</td>
<td>project_type</td>
<td>project_type_title</td>
</tr>
<tr>
<td>responsibility</td>
<td>responsibility</td>
<td>responsibility_title</td>
</tr>
<tr>
<td>function</td>
<td>function</td>
<td>function_title</td>
</tr>
<tr>
<td>activity</td>
<td>activity</td>
<td>activity_title</td>
</tr>
<tr>
<td>object</td>
<td>object</td>
<td>object_title</td>
</tr>
<tr>
<td>counterpart</td>
<td>counterpart</td>
<td>counterpart_title</td>
</tr>
<tr>
<td>free</td>
<td>free</td>
<td>free_title</td>
</tr>
<tr>
<td>amount</td>
<td>amount</td>
<td></td>
</tr>
</tbody>
</table>

Table 6. Structure to be implemented in HBase

The different tables in the relational data warehouse, included in the STAR-schema, were combined into one big table. Previous research have shown that this is more effective for queries (Dehdouh et al., 2015; Chevalier et al., 2015b). The measure “amount” was put in its own column-family. Each dimension also got its own column-family which in this implementation consisted of the dimension’s identification-number and the name of the dimension. If the dimension had more attributes these would have been included too.

The table needed to export the data from the relational data warehouse to produce the structure in table 6 had to include all the columns. Column-families weren’t defined until the data was imported into HBase. To get the data needed to implement it into HBase the dimensional-tables and the fact-table were combined using inner joins. Since HBase also needed a row-key the function ROW_NUMBER() OVER (SELECT 100) was added to the
query to show the row-number. This data was then exported to a csv-file using the import/export wizard.

5.4 Implementation of HBase
The Hortonworks’ sandbox for Hortonworks data platform (HDP) and Hortonworks data flow (HDF) was used. This is a single node virtual machine (VM) that comes with Apache Hadoop, Apache Hive, Apache Phoenix, Apache HBase and other Hortonworks products. The products used in the implementation were Apache HDFS, Apache HBase and Apache Hive. The virtual machine was given 10 Gb of RAM. HBase was configured with 1GB memory for the master and 1 GB memory for the region server.

First the csv-file that had been exported from the relational data warehouse was uploaded to the HDFS. After that the file was imported from the HDFS into HBase using a function called importtsv. In this function the column-families and the columns were defined.

It took about one hour to import the data. The results showed about 1 million bad rows which meant that only about 5 million rows were imported. It was difficult to troubleshoot which rows were missing since the import wouldn’t state which rows were classified as bad rows. Since it wasn’t possible to deduce which rows were missing in the timeframe, the study continued with a smaller amount of rows.

To be able to query HBase in an SQL-like way, to compare it to the relational data warehouse, Apache Hive was used. In Hive one can create an external table that works with the HBase table. This way it is possible to use HiveQL to query the HBase table. Apache Hive was used since it is described as the data warehouse of the Hadoop ecosystem (Capriolo et al., 2012).
5.5 Queries
Two types of questions were chosen, all typical questions when getting economical reports. They are all questions that can be used when getting an economical report.

The first type of query summarises the amount column on a number of dimensions. The first query sums the amount on one dimension, the second query on two dimensions and so on (see figure 4 and figure 5). All of the queries can be seen in appendix 1 and 2. This is a pretty common type of query since it can be used to show an economical report that shows the results for each account (see example in table 7).

---
SELECT account_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661 --2016
GROUP BY account_title

---
SELECT title as account_title, sum(amount) as amount
FROM eco_all t
INNER JOIN dbo.account_dim_new k on k.id=t.account
WHERE year = 113661 --2016
GROUP BY title

Figure 4. The first type of query in HiveQL

Figure 5. The first type of query in SQL

<table>
<thead>
<tr>
<th>Account</th>
<th>Results current year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account 1</td>
<td>123,5</td>
</tr>
<tr>
<td>Account 2</td>
<td>234</td>
</tr>
<tr>
<td>Account 3</td>
<td>345</td>
</tr>
<tr>
<td>Account 4</td>
<td>2345</td>
</tr>
</tbody>
</table>

Table 7. Example of output from the questions

Year: 2018
The second type of query sums the amount column on one dimension but each new query adds a condition. This type of query is also very common when getting economical reports since it is common to filter the data of the report on different conditions, like month, year and project. All of the queries can be seen in appendix 1 and 2.

```
--Question 6
SELECT account, account_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661 and month = 21179--Juli
GROUP BY account, account_title
```

**Figure 6. The second type of query in HiveQL**

```
--Question 6 |
SELECT k.title as account_title, sum(amount)
FROM eco_all t
inner join dbo.account_dim_new k on k.id=t.account
WHERE year = 113661 and month = 21179--Juli
GROUP BY k.title
```

**Figure 7. The second type of query in SQL**

More queries could have been tested especially with different operations like ‘<’, but the aim was to keep it simple and focus on queries that are common when getting economical reports with this data source. Usually one looks at data from one year and if one were to look at data from January to a specific month one would most likely create an accumulative table. This means that for the data source, most likely, queries using ‘=' operators would be used. The study has not taken into account different aggregated tables and how one would optimise these the best in the new column-oriented store.
6. Analysis and results

This chapter presents the results of the implementation and analyses them.

6.1 Evaluating the queries

Apache HBase/Apache Hive did not perform very well compared to the relational data warehouse. The queries in the relational data warehouse took no time at all while the queries in HBase/Hive took several minutes. In figure 8 and 9 and in table 8 and 9 the query-times can be seen. There is a jump between using three dimensions and using 4 dimensions for HBase for the first type of queries.

![Figure 8. HBase compared to RDBMS for the first type of queries (summarising the amount on one, two etc. dimensions)](image)

<table>
<thead>
<tr>
<th></th>
<th>1 dim (sek)</th>
<th>2 dim (sek)</th>
<th>3 dim (sek)</th>
<th>4 dim (sek)</th>
<th>5 dim (sek)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBase</td>
<td>163</td>
<td>168</td>
<td>174</td>
<td>242</td>
<td>246</td>
</tr>
<tr>
<td>RDBMS</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 8. The results from the first type of questions (summarising the amount on one, two etc. dimensions)
Even though the queries did not perform very well in HBase compared to the relational data warehouse, its performance seems to be in the same range as previous studies using the same type of mapping. The study by Dehdouh et al. (2015) used some queries that were similar to the queries used in this study and their results were about 250 seconds for the model DLA-CF when summarising on one dimension and about 600 seconds for summarising on four dimensions. They did, however, use $10^9$ rows while this study only had 6 million rows so the amount of data used in their study was a lot bigger. Their data did, however, only have 6 dimensions while the data in this study had 12 dimensions. Another parameter is also that they used Apache Phoenix instead of Apache Hive to ask SQL-queries.

The study by Chevalier et al. (2015b) used different amounts of data $10^7$, $10^8$ and $10^9$. It is a bit unclear which of these amount of data that they used to get the results of their queries and it is also a bit unclear which queries they asked. It does, however, seem like they also summarise the data on different dimensions and that they have $10^7$ rows in the fact-table, which is almost in the same range as the data used in this study. They also used Apache Hive.
to ask SQL queries. The different combinations of three dimensions shows results from 540 seconds to 642 seconds for the model MC1.

Compared to these previous studies the results obtained from HBase here doesn’t seem completely off. It does however perform really bad compared to the relational data warehouse which is why it is interesting to compare the relational data warehouse to the column-oriented one. Previous studies have only compared the results of the mapping to other types of NoSQL databases and not a classic relational data warehouse which is very relevant.

6.2 Evaluating the implementation

A sandbox provided by Hortonworks was used for the implementation. This sandbox had everything needed to try out the Hadoop ecosystem which made the implementation a lot easier. There is, however, a lot to learn when starting to work with such a large ecosystem of programs.

The virtual machine was given 10 Gb of RAM and HBase was configured with 1GB memory for the master and 1 GB memory for the region server. Giving more memory to these entities could possibly have improved the query-times but by monitoring the memory usage during the querying one could see that it wasn’t overloaded.

Using the sandbox could, of course, have some drawbacks like limitations of optimise the installation. If this were to be used in a actual company one might have to consider implementing the programs from scratch.

Another optimisation possibility that was considered was the use of index in HBase/Hive. Since this was used in the relational data warehouse it seemed a good idea to use it in column-oriented implementation too. Indexing in Hive is said to be limited (Capriolo, Wampler & Rutherglen, 2012) and since an external table was used indexing it didn’t work. The combination of indexes and external tables doesn’t seem to appear in literature or documentation from Hortonworks.

The mapping between the STAR-schema and the logical column-oriented structure wasn't too difficult to do since it only consisted of three steps. Combining the actual data in the data warehouse into one big table wasn’t too complicated either but it took some time to run the query that combined all of the dimensions, since there was quite a lot of data. Since it was known that the dimensional references in the fact-tables didn’t include any null-values, inner joins could be used. If this hadn’t been the case left-joins would have had to be used which could have made the query take longer.
Using the bulk import from HDFS to HBase took a bit more than one hour to do. Not all rows were imported though and it was difficult to trouble-shoot why. The bulk import just said that there was a certain number of bad rows which means that they weren't imported. They did, however, not state why the rows were bad. Some trouble-shooting were done by exporting the data from HBase back into a csv-file that then were read into the relational database again to be able to compare the two tables to see which rows were missing. This way of trouble-shooting was very tedious and no results were found. It would have been good if the bad lines could have been identified in the Hadoop ecosystem but that doesn’t seem to be the case.

The missing rows in HBase meant that HBase would be working with less data than the relational data warehouse. The relational data warehouse did, however, still perform better than HBase so it does not appear to have favoured HBase.

Since the column-oriented database cannot be queried using SQL, a layer that transforms a SQL-like language to java code is used, called Apache Hive. Since this means that there is some sort of internal mapping between HiveQL and java code a guess is that this could make the queries slower. It is, however, great of be able to use a SQL-shell over HBase. This makes it a lot easier for a person already used to SQL (and that is a lot of people) to use the column-oriented structure. HiveQL is not as advanced as other dialects of SQL but for the queries in this study it was enough. If more advanced queries has to be done, one example being that HiveQL does not support sub-queries, one might have to rethink using it.

There are, however, usually ways to avoid subqueries and the queries used in the relational data warehouse does not seem to include them. There are also other options to Hive, like Phoenix, that also enables the user to use SQL with HBase, but they do not state exactly how advanced this dialect of SQL is.

HBase is supposed to have better flexibility than a relational database. It is however, not tested in this study. There is the option in HBase to add a new column in a column-family on a row by row bases so if a new column was needed for newer rows it could just be added for the new rows and the history wouldn’t have to be changed. In a relational database one would have to update the schema for all the data to do this. How one would handle the new column in HBase when using Hive is however, unclear since this program creates an external table using HBase as a store so it seems to have a more fixed schema.

The choice to use Apache Hive instead of Apache Phoenix might also have affected the query-times of the queries. The previous study by Dehdouh et al. (2015) did use Phoenix instead of Hive but still had queries that took several minutes. They did, however, work with more rows so it is difficult to compare. It would have been good to also use Apache Phoenix
to ask the same queries to see if this way of querying HBase using SQL would have lead to better performance. Table 10 contains a summary of results from previous studies.

<table>
<thead>
<tr>
<th></th>
<th>Chevalier et al. (2015b)</th>
<th>Dehdouh et al. (2015)</th>
<th>This study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of rows (in fact table)</td>
<td>$10^7$</td>
<td>$10^9$</td>
<td>$\approx 5000000$</td>
</tr>
<tr>
<td>Number of dimensions</td>
<td>4</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Program for querying HBase</td>
<td>Apache Hive</td>
<td>Apache Phoenix</td>
<td>Apache Hive</td>
</tr>
<tr>
<td>Summarising on one dimension</td>
<td>N/A</td>
<td>$\approx 250$ sek</td>
<td>163 sek</td>
</tr>
<tr>
<td>Summarising on two dimensions</td>
<td>N/A</td>
<td>$\approx 500$ sek</td>
<td>168 sek</td>
</tr>
<tr>
<td>Summarising on three dimensions</td>
<td>540 - 642 sek (depending on dimensions combination)</td>
<td>$\approx 600$ sek</td>
<td>174 sek</td>
</tr>
<tr>
<td>Summarising on four dimensions</td>
<td>N/A</td>
<td>$\approx 700$ sek</td>
<td>242 sek</td>
</tr>
<tr>
<td>Summarising on five dimensions</td>
<td>N/A</td>
<td>N/A</td>
<td>246 sek</td>
</tr>
</tbody>
</table>

Table 10. A comparison of the results to previous studies
7. Conclusion

7.1 Contributions and conclusion

This study has mapped an economical model for a relational data warehouse to a logical column-oriented structure to answer the question:

What are the consequences of converting a data warehouse based on a STAR-schema to a column-oriented-NoSQL-database?

The logical structure has been implemented using Apache HBase. Both the relational and the column-oriented data warehouse have been queried to evaluate the result of the mapping. This shows that even though the relational structure can be mapped to a column-oriented one and implemented, the time it takes to query the data is a lot longer for the new column-oriented data warehouse ergo a consequence of converting a data warehouse based on a STAR-schema to a column-oriented-NoSQL-database is longer query-times.

One of the contributions of this study is that it has implemented mapping-styles, that has previously only been tested in controlled environments, using real-life data. The results, however, show that using this data doesn't make that much of a difference because the query times are in the same range as previous studies.

The main contribution is that this study has compared the query-times in the column-oriented-NoSQL-database with the query-times in the relational data warehouse. Previous studies has only compared this to other NoSQL implementations.
7.2 Future work

This study has mostly evaluated the mapping and the implementation of the logical structure that resulted from the mapping by comparing query-times. In the future it would have been interesting to check the flexibility of using HBase instead of a relational data warehouse since the ability to add another column to the data without much difficulty is one of the advantages of using this type of storage. It would therefore be relevant to compare the time it takes to add another column in a relational data warehouse and the time it took to add it in HBase.

Another aspect of this is that since the data that was used performs well in the relational data warehouse and isn’t unusually large it might not be the best fit for the Apache Hadoop ecosystem. Since Hadoop is supposed to be great when working with huge amounts of data we might have had other results if the data had been so large that it performed badly in the relational data warehouse. It would have been interesting to see if there is a line where the data starts performing worse in the relational data warehouse while Hadoop performs the same.

It would, of course, also be interesting to see if there are possibilities to optimise the HBase implementation and if it would perform better when using another SQL-layer like Apache Phoenix.

Since this study has only focused on converting one STAR-schema it would be interesting to see what the consequences of implementing more STAR-schemas and also combinations of STAR-schemas in HBase are.

Future work could also include testing more queries, possibly using another dataset where more queries are natural. It would also be interesting to see how one would handle aggregated tables.
References


Zhao, G., Li, L., Li, Z., & Lin, Q. (2014, November). Multiple nested schema of HBase for migration from SQL. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on* (pp. 338-343). IEEE.
Appendix 1. Queries and results relational data warehouse

---------PART 1----------

--Question 1 --Execution-time: 0 sek
SELECT title as account_title, sum(amount) as amount
FROM eco_all t
INNER JOIN dbo.account_dim_new k on k.id=t.account
WHERE year = 113661 --2016
GROUP BY title

--Question 2 --Execution-time: 1sek
SELECT k.title as account_title, proj.title as project_title, sum(amount) as amount
FROM eco_all t
inner join dbo.account_dim_new k on k.id=t.account
inner join dbo.project_dim_new proj on proj.id=t.project
WHERE year = 113661
GROUP BY k.title, proj.title

--Question 3 --Execution-time: 2
SELECT k.title as account_title, proj.title as project_title, f.title as function_title, sum(amount) as amount
FROM eco_all t
inner join dbo.account_dim_new k on k.id=t.account
inner join dbo.project_dim_new proj on proj.id=t.project
inner join dbo.function_dim_new f on f.id=t.[function]
WHERE year = 113661
GROUP BY k.title, proj.title, f.title

--Question 4 --Execution-time: 4 sek
SELECT k.title as account_title,
    proj.title as project_title,
    f.title as function_title,
    act.title as activity_title,
    sum(amount) as amount
FROM eco_all t
inner join dbo.account_dim_new k on k.id=t.account
inner join dbo.project_dim_new proj on proj.id=t.project
inner join dbo.function_dim_new f on f.id=t.[function]
inner join dbo.activity_dim_new act on act.id=t.activity
WHERE year = 113661
GROUP BY k.title, proj.title,f.title,act.title

--Question 5-- Execution-time: 7 sek
SELECT
k.title as account_title,
proj.title as project_title,
f.title as function_title,
act.title as activity_title,
obj.title as object_title,
sum(amount) as amount
FROM eco_all t
  inner join dbo.account_dim_new k on k.id=t.account
  inner join dbo.project_dim_new proj on proj.id=t.project
  inner join dbo.function_dim_new f on f.id=t.[function]
  inner join dbo.activity_dim_new act on act.id=t.activity
  inner join dbo.object_dim_new obj on obj.id=t.[object]
WHERE year = 113661
GROUP BY k.title, proj.title, f.title, act.title, obj.title

--------------------------------------
---------PART 2-----------------------
--------------------------------------

--Question 6 --Execution-time: 0
SELECT k.title as account_title, sum(amount)
FROM eco_all t
inner join dbo.account_dim_new k on k.id=t.account
WHERE year = 113661 and month = 21179--Juli
GROUP BY k.title

--Question 7 --Execution-time: 0
SELECT k.title as account_title, sum(amount) as amount
FROM eco_all t
  inner join dbo.account_dim_new k on k.id=t.account
  inner join dbo.project_dim_new proj on proj.id=t.project
WHERE year = 113661 and
month = 21179--Juli
and project = 452963-- Projekt saknas
GROUP BY k.title

--Question 8 --Execution-time: 0
SELECT k.title as account_title, sum(amount) as amount
FROM eco_all t
  inner join dbo.account_dim_new k on k.id=t.account
  inner join dbo.project_dim_new proj on proj.id=t.project
  inner join dbo.function_dim_new f on f.id=t.[function]
WHERE year = 113661 and
month = 21179--Juli
and project = 452963-- Projekt saknas
and [function]=456625-- 92070 IT PERSONAL LOKAL
GROUP BY k.title

--Question 9 --Execution-time: 0 sek
SELECT k.title as account_title, sum(amount) as amount
FROM eco_all t
    inner join dbo.account_dim_new k on k.id=t.account
    inner join dbo.project_dim_new proj on proj.id=t.project
    inner join dbo.function_dim_new f on f.id=t.[function]
    inner join dbo.activity_dim_new act on act.id=t.activity
WHERE year = 113661 and
month = 21179--Juli
and project = 452963-- Projekt saknas
and [function]=456625-- 92070 IT PERSONAL LOKAL
and activity=433556 --15830000 INFRASTRUKTUR
GROUP BY k.title

--Question 10 --Execution-time: 0 sek
SELECT k.title as account_title, sum(amount)
FROM eco_all t
    inner join dbo.account_dim_new k on k.id=t.account
    inner join dbo.project_dim_new proj on proj.id=t.project
    inner join dbo.function_dim_new f on f.id=t.[function]
    inner join dbo.activity_dim_new act on act.id=t.activity
    inner join dbo.object_dim_new obj on obj.id=t.[object]
WHERE t.year = 113661 and
t.month = 21179--Juli
and t.project = 452963 -- Projekt saknas
and t.[function]=456625 -- 92070 IT PERSONAL LOKAL
and t.activity=433556 --15830000 INFRASTRUKTUR
and t.[object]=448666 --Objekt saknas
GROUP BY k.title
Appendix 2. Queries and results Apache Hive/Apache HBase

--------------------------------------
---PART 1---
--------------------------------------

--Question 1 --Execution-time: 2m 43s 7ms
SELECT account_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661 --2016
GROUP BY account_title

--Question 2 --Execution-time: 2m 48s 347ms
SELECT account_title, project_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661
GROUP BY account_title, project_title

--Question 3 --Execution-time: 2m 54s 515ms
SELECT account_title, project_title, function_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661
GROUP BY account_title, project_title, function_title

--Question 4 --Execution-time: 4m 2s 902ms
SELECT account_title, project_title, function_title, activity_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661
GROUP BY account_title, project_title, function_title, activity_title

--Question 5-- Execution-time: 4m 6s 967ms
SELECT account_title, project_title, function_title, activity_title, object_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661
GROUP BY account_title, project_title, function_title, activity_title, object_title
--Question 6 --Execution-time: 2m 53s 819ms
SELECT account, account_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661 and month = 21179--Juli
GROUP BY account, account_title

--Question 7 --Execution-time: 2m 57s 813ms
SELECT account, account_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661 and month = 21179--Juli
and project = 452963-- Projekt saknas
GROUP BY account, account_title

--Question 8 --Execution-time: 3m 8s 607ms
SELECT account, account_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661 and month = 21179--Juli
and project = 452963-- Projekt saknas
and functionid=456625-- 92070 IT PERSONAL LOKAL
GROUP BY account, account_title

--Question 9 --Execution-time: 3m 58s 481ms
SELECT account, account_title, sum(amount) as amount
FROM hbase_table_eco_all
WHERE year = 113661 and month = 21179--Juli
and project = 452963-- Projekt saknas
and functionid=456625-- 92070 IT PERSONAL LOKAL
and activity=433556 --15830000 INFRASTRUKTUR
GROUP BY account, account_title

--Question 10 --Execution-time: 4m 11s 564ms
SELECT account, account_title, sum(amount)
FROM hbase_table_eco_all
WHERE year = 113661 and month = 21179--Juli
and project = 452963-- Projekt saknas
and functionid=456625-- 92070 IT PERSONAL LOKAL
and activity=433556 --15830000 INFRASTRUKTUR
and object=448666 --Objekt saknas
GROUP BY account, account_title