The Modernization Process of a Data Pipeline

Sebastian Pulkka 42004

Master’s thesis in Computer Engineering

Supervisor: Mats Aspnäs

Åbo Akademi University

Faculty of Science and Engineering

2023

Abstract

Data plays an integral part in a company's decision-making. Therefore, decision-makers must have the right data available at the right time. Data volumes grow constantly, and new data is continuously needed for analytical purposes. Many companies use data warehouses to store data in an easy-to-use format for reporting and analytics. The challenge with data warehousing is displaying data using one unified structure. The source data is often gathered from many systems that are structured in various ways.

A process called extract, transform, and load (ETL) or extract, load, and transform (ELT) is used to load data into the data warehouse. This thesis describes the modernization process of one such pipeline. The previous solution, which used an on-premises Teradata platform for computation and SQL stored procedures for the transformation logic, is replaced by a new solution. The goal of the new solution is a process that uses modern tools, is scalable, and follows programming best practises. The cloud-based Databricks platform is used for computation, and dbt is used as the transformation tool. Lastly, a comparison is made between the new and old solutions, and their benefits and drawbacks are discussed.

Keywords: Data warehouse, ETL, ELT, dbt, Databricks

Preface

I want to thank my manager for providing me with the opportunity to work on this project and write this thesis. Additionally, I would like to thank team members who have helped with the implementation of the project and given feedback on the thesis.

Sebastian Pulkka
Turku, 1 May 2023

Table of Contents

List of Abbreviations

API	Application programming interface
ETL	Extract, transform, and load
ELT	Extract, load, and transform
dbt	Data build tool
SQL	Structured query language
CI/CD	Continuous integration / continuous deployment
BCNF	Boyce-Codd normal
3NF	Third normal form
OLAP	Online analytical processing
DBMS	Database management system
ACID	Atomicity, consistency, isolation, and durability
DDL	Data definition language
DAG	Directed acyclic graph
DML	Data manipulation language
CTE	Common table expression

1. Introduction

For a business to be able to make good decisions, data is needed to support the decisions. The more comprehensive and accurate the data is, the better analysts and managers can make important decisions. Another essential factor is that the data is in an easy-to-use format, making it convenient for the end users. This is where data integration plays a significant role in converting heterogeneous data from multiple sources into one homogeneous data warehouse. This thesis studies one such real-world data pipeline.

1.1 Problem Statement

The company studied for this thesis is a large company in the financial sector with business in many countries. Because of this, there are many source systems that produce data in varying formats. This data is then loaded into a central data warehouse to provide one easy-to-use system to aid decision-making.

This thesis describes the process of modernizing one of these data pipelines, and the benefits and drawbacks of the new solution are compared with the old solution.

Three main steps must be performed so that the data can be used by the data consumers in a meaningful and efficient manner. First, the data needs to be extracted from the source. This is done by reading text files, other databases, or APIs. Once the data has been extracted from the data producers, it has to be transformed to make it easier to use for analytical purposes. Some common steps done at this stage are enriching the data by joining it with other sources, casting data into their correct format, and pivoting the data to achieve the correct granularity level. Lastly, the data is loaded into target tables in the data warehouse, where it can be used for analytical purposes. The order of these steps can vary depending on what tools and architectures are used. One common method is extract, transform, and load (ETL). Another method is extract, load, and transform (ELT).

The process of extracting, transforming, and loading the data into the data warehouse is important to be able to have data from all the source systems in a format that is easy to use for analysts and allows for easy comparisons between different systems. This becomes especially important in a larger company with business in multiple countries.

There is also a multitude of different tools that can be used for this process which all have different benefits and drawbacks [1]. Therefore, choosing the right tool for the occasion is particularly important. This thesis aims to compare the previous approach of using SQL stored procedures for the transformation, with newer tools that allow developers to follow programming best practises. For the transformation part of the pipeline, dbt-Lab's data build tool (dbt) is the tool that will be used, and the benefits and drawbacks will be compared to the previous approach. Another goal of the thesis is to move the data processing away from the current on-premises solution, which is under a very heavy load due to increased usage, into the cloud. Again, the benefits and drawbacks of having a cloud-based platform for data processing are discussed.

1.2 Goal of the Thesis

As a part of this thesis, one data pipeline will be modernized. This is needed, since the previous tools that were used lacked many key features that are essential for a smooth process. Some of the benefits that will, hopefully, be achieved with the new tools are:

- Create a solution that uses modern tools in favour of outdated ones.
- Create a more efficient loading process, meaning that data will be available for analysis at an earlier stage.
- Run the data transformations in a cloud environment, allowing for more scalability compared to the currently overloaded on-premises solution.
- Create a solution that is version controlled in Git, allowing for change tracking and improved maintainability compared to the previous solution.
- Incorporate continuous integration (CI)/continuous deployment (CD) to simplify and speed up the development and deployment process.
- Improve the data quality by introducing automated data testing. This will help find issues in the source data at an earlier stage and mean that the consumers of the data warehouse data will be able to trust the correctness of the data better.

In addition, the current version of the pipeline lacks some core features that will be added in parallel with the modernization steps:

- Create a “frozen” solution, meaning that the process can be run multiple times and still produce the same result. This is important, since the current solution changes when corrections to the data come in. This is not always desired, as the ability to go back and check what data decisions were made upon is lost.
- Add two new measures for counting the number of products and objects.

The scope of the work in this thesis is to implement and document the modernization of a data pipeline, starting by reading data from an existing normalized data warehouse. Then the data will be transformed into an easy-to-use denormalized star schema using modern tools. The processing will be moved to a cloud environment, while still providing the target star schema also on the on-premises environment for compatibility reasons. The benefits and disadvantages of the new solution will then be discussed in *5. Evaluation of Results.*

2. Data Warehousing Concepts

2.1 Data Warehousing

In general, an organization has two types of database systems [2]. The first is an operational system, with the goal of keeping record of operational activities, such as handling customer orders. The other database system used in many organizations aims to provide an overview of the transactions logged in the operational database. This type of system is called a data warehouse. It can be defined as a central repository where information is gathered from multiple source systems, and the data is integrated using one common model [3]. The primary goal of a data warehouse is to provide valuable data that can help a company make data-driven decisions.

Especially in larger enterprises, it is common to have many different operational systems that all create data in different formats and separate locations. For analytical purposes, it is important to have all the data archived in one central place. This presents one of the challenges in data warehousing, where the goal is to combine data from multiple sources and present it using one common architecture and model. This architecture and model should be optimized for analytical queries. The queries that are needed for analyses often involve querying a large number of rows (up to millions or even billions of rows) and then aggregating the data to see trends and, thereby, evaluate the performance of the different operational activities. This contrasts with an operational database, where there are many small requests, often just updating or adding one single row at a time. Another difference between the two types of databases is that an operational database often stores only the current information. A data warehouse additionally stores historical data to help see trends in the data and thereby be able to make more informed decisions.

Since data is loaded to the data warehouse by reading from the operational source systems, there will always be a delay between when the data is produced in the source system and when it is available for analysis in the data warehouse. There are different methods of loading the data to the data warehouse. Daily batch loading is a common method where new information is loaded to the data warehouse daily using batch jobs. However, more current data, i.e., data-freshness, has become increasingly common. This can be accomplished using mini-batches, run multiple times a day, or with streaming solutions if near real-time updates are required [3].

One of the core metrics of a successful data warehouse solution is that the business community must accept and use the new solution. The value provided by a data warehouse comes from the decisions made after analyzing the data in it. Therefore, if there are few or no users, it will not provide any business value. To ensure that the data warehouse is used, it is important to create a solution that is both fast and easy to use, so that business users are eager to use the new data warehouse in favour of alternative solutions. [2]

2.1.1 Dimensional Modelling

Normalization is an important aspect to consider when designing a database. Normalization is the concept of splitting the information into schemas in a way that reduces redundancy, meaning that a completely normalized model should have no repetition of information. Having a normalized database is important for consistency, as updates are needed only in one place, and there is less risk of inconsistencies where some information would be updated in parts of the database but not in others. However, a highly normalized database comes with the drawback of complex queries with many joins, as the information is split into many tables to avoid redundancy. There are multiple levels of normalization defined, which all have different rules for the required normalization [3]. Two examples of these are Boyce-Codd normal form (BCNF) and third normal form (3NF).

Dimensional modelling is one commonly used method in data warehousing that can be used to achieve a data model that is easy for business users to understand and provides good query performance. [2] Dimensional models are denormalized, and the goal is to optimize read performance. This means that dimensional models have some data redundancy, decreasing write performance, but fewer joins are needed, improving the read performance. This is in contrast to BCNF and 3NF modelling, where the goal is to normalize the database to achieve optimal write performance.

Dimensional data models can be represented in two ways depending on what type of database system is used. If a relational database system is used, the model is a star schema. If a multidimensional database management system is used, it is called an online analytical processing (OLAP) cube.

OLAP cubes have improved query performance compared to star schemas because of optimizations such as indexing and pre-calculations. This improved performance comes with the drawback of worse performance when loading the data into the data warehouse. Therefore [2] recommends using star schemas at least as a starting point for loading the data. If needed, OLAP cubes could then be populated based on the star schema.

With relational dimensional modelling (star schemas), there are generally two types of tables, fact tables, and dimension tables.

Fact tables are used in star schemas for storing the measurements from a business process, like sales in a grocery store. In this example, the fact table would contain the price of the sale and potentially a transaction number. Other information like the product, customer, store, or clerk information will be stored in dimension tables so that the fact table only contains a foreign key to these tables.

All rows in a fact table should be on the same level of detail, i.e. have the same grain. This means that one row in the fact table should always correspond to the same type of event in the real world. For example, if we have a fact table containing information about sales, one possibility would be to have one row in the fact table per sold product. It is important that all rows are on this level and that there are no rows representing multiple products or groups of products. This is important so that aggregation can be done in a way that avoids double counting.

According to [2], the measures in fact tables should ideally be numeric and additive. Additive facts are useful since many rows are often retrieved in analytical applications, and grouping on some dimensional attribute and summing on the measurements is often useful. An example of an additive measurement is the sales amount. Account balance or unit price, on the other hand, are not additive, which means that summation is not possible. In these cases, only averages or counts can be used. Textual facts are also theoretically possible, but since most textual information is derived from a list of possible values, it should instead be modelled in a dimension table to reduce redundancy and consume less space. Therefore, textual measurements should only be used if the content of the text field is unique. However, this type of information would be tough to analyse in a meaningful way, which is why textual measurements are rarely used.

Dimension tables should, according to [2], describe the measurement in the fact table by containing attributes that give the context of: “who, what, where, when, how, and why”.

Dimension tables consist of a unique primary key that uniquely identifies one row in the dimension table and is used for joining the dimension with the fact table. In addition to the primary key, a dimension contains attributes that describe the business process. The number of attributes is often high, and large textual fields can be included. Since the number of rows in a dimension table is small compared to a fact table, the space needed for dimension tables is still less than what is needed for the fact table, in most cases. Attributes should be descriptive, and the use of codes should be avoided to make the model intuitive and easy to use for business users.

These attributes in the dimension tables can then be used for filtering answer sets and doing groupings. If we have a fact table consisting of product sales, we might have one dimension for the product and one for the store. These dimensions could then be used to filter and find the sales of a particular product or sales made in a particular store or city.

When deciding if a numeric field should be in a fact table or in a dimension table, one should look at the possible values the field can take. If the field can only take a discreet number of different values, it should usually be modelled in a dimension, and if it is a continuous field that can take on a broad range of values, it should most likely be modelled in a fact table. This can sometimes be ambiguous. For example, the standard price of a product might seem like it is constant and should therefore be in a dimension. On the other hand, it can also change quite often over time and should therefore be modelled in a fact table.

Dimension tables should be denormalized to improve query performance and simplicity. This means that some information will be duplicated, increasing the storage need. However, this is necessary to keep the structure simple and the queries performant. If normalization is still desired, snowflake schemas can be used to achieve this.

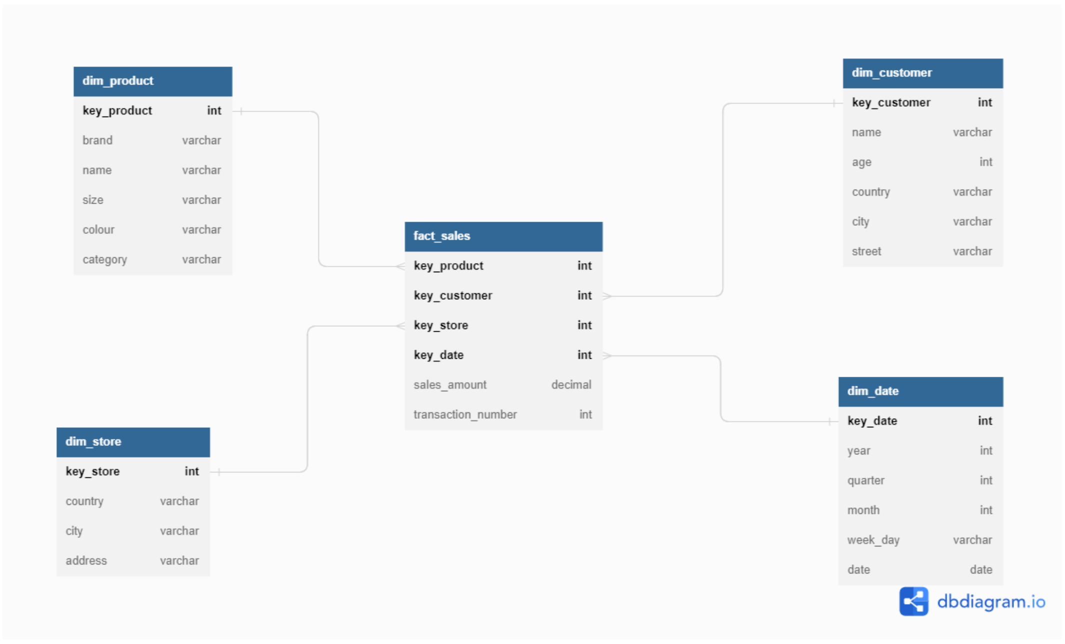

Figure 1 Diagram of dimension and fact tables in a star schema

Figure 1 shows an example of a star schema with four dimensions and one fact table in the middle. The fact table contains the measurements *sales_amount* and transaction number, as well as foreign keys used for joining with the four dimensions.

With snowflake schemas, the dimensions from a star schema are normalized. New sub-dimensions are created and connected to the normalized dimensions [4]. The normalization can be done for all the dimension tables in the star schema or only for a subset of the tables. The dimension tables can also be completely normalized or only partly normalized.

The advantages of normalization and creating the snowflake structure are that storage can become more efficient as duplication of large fields can be avoided. Maintenance may also be easier because of the normalized structure, meaning that updates are only needed in one place.

Disadvantages are creating a more complex structure which makes usability difficult, and the performance will also be degraded due to the additional joins needed with a more normalized structure. Therefore, one should carefully consider if a snowflake schema is worth it over an easier-to-use star schema.

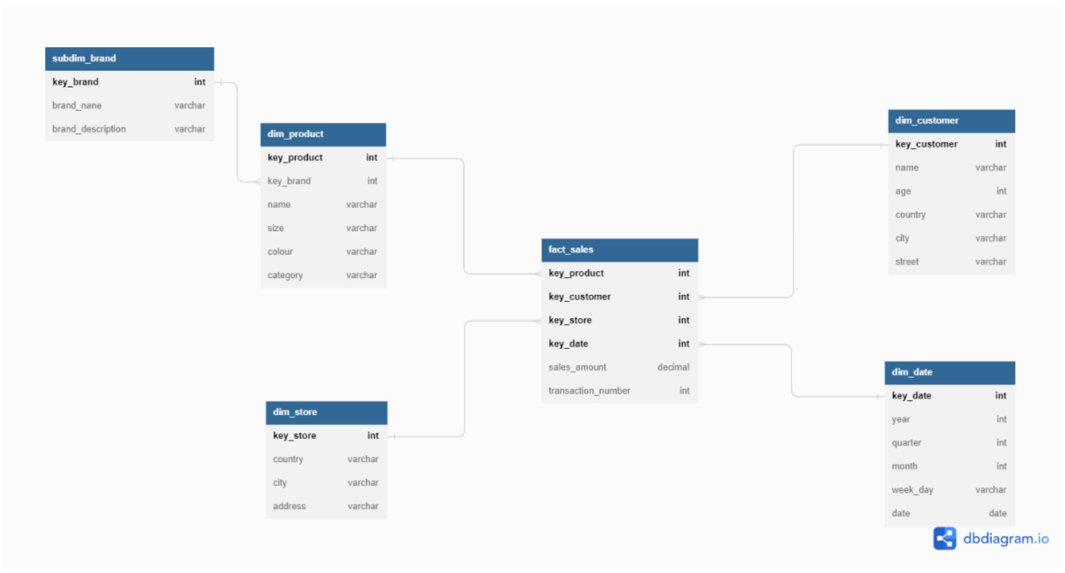

Figure 2 Diagram of a snowflake schema where the product dimension has been partly normalized.

Figure 2 shows a "snowflaked" version of the star schema from *Figure 1*. The product dimension has been partially normalized by separating the brand information into a separate sub-dimension. This approach can save some space as the large brand name and description fields are not duplicated for all the products of the same brand. In addition, making changes to the brand name or description fields is simple, and only one row needs to be updated in the subdimension. However, as the product dimension is likely not very big, the amount of space that can be saved is small, and the added complexity of the model and the potential performance impacts mean that one should carefully consider if using the snowflake structure is strictly necessary.

Referential integrity is one important aspect of the star and snowflake schemas. This means that every key that exists in the fact table should find a match in their respective dimension table. On the other hand, keys not present in the fact table may still exist in dimension tables. Database management systems can often enforce referential integrity by defining the tables' foreign and primary key attributes.

As a result of how fact tables and dimension tables are structured, joining becomes simple. Often only a couple of joins are needed together with where clauses and group by statements to be able to answer questions that are relevant to the business. This means that the core requirement when designing a data warehouse, ease of use, can be met using this structure. *Figure 3* shows an example of a simple SQL query using the schema defined in *Figure 1* to determine the monthly total sales in stores located in Turku during 2022.

```
SELECT
    dim_date.month,
    SUM(fact_sales.sales_amount)
FROM
    fact_sales
    INNER JOIN dim_store ON
        fact_sales.key_store = dim_store.key_store
    INNER JOIN dim_date ON
        fact_sales.key_date = dim_store.key_date
WHERE
    dim_date.year = 2022 and
    dim_store.city = 'Turku'
GROUP BY dim_date.month
```

Figure 3 Sample SQL statement for querying a star schema.

After the initial model has been designed, there will inevitably be changes required to this initial model. Therefore, it is important that changes can be made with as little development effort as possible. With a star schema, it is easy to create completely new dimensions and connect them to fact tables, add new measures given that they are of the same grain as the current fact table, or add new attributes to a dimension. Most importantly, all these changes can be made so that the existing users will not be affected, and previous queries will still produce the same result. Take the example in *Figure 2* as an example. If a new dimension for the clerk who made the sale were added, the query would not need to be changed, and the result would still be the same.

2.1.2 The Enterprise Data Warehouse

Another common architecture for the data warehouse is the enterprise data warehouse first proposed by Inmon [5]. He advocates for creating one central repository where all the data from the whole enterprise is stored. The data in the enterprise data warehouse should be normalized and modelled using traditional relational modelling techniques meaning that updates are more efficient, and less storage space is required. The enterprise data warehouse can then be used as a base for further transformations and aggregations that are done departmentally to create data marts that meet the specific needs of different departments. This method tries to reduce inconsistencies as the same base is used for all the data marts, while also emphasizing creating a technically performant solution.

The main difference between Inmon's approach and Kimball's star schema [2] is that Kimball advocates for process-oriented data modelling [5]. With the star schema, the goal is to have one star schema representing one business process. Business processes often overlap departments meaning that the departmental distinction is not made when using Kimball's approach. Inmon also argues for one central repository covering the whole organization, but unlike Kimball, he believes that the modelling should be done in a data-centric way. This means that the characteristics of the data should be used as a base when creating the schema. As a result, Kimball's model has a lower initial cost as the modelling is lighter and easier for people without a strong IT background to understand. On the other hand, designing an Inmon-style enterprise data warehouse takes more initial effort, as creating a relational model covering the whole enterprise is time-consuming. However, after the initial development has been done, subsequent changes and expansion require less effort due to the normalized design.

As an alternative to the two methods Kimball [2] presents the option to use the two approaches together so that the normalized enterprise data warehouse is used as a base when creating the denormalized fact and dimension tables of the Kimball star schema. This approach may be good when a normalized enterprise data warehouse solution already exists but cannot meet the needs of the business users due to the complexity of the model, making it difficult to understand for non-IT personnel. However, Kimball argues that because of the overhead this dual approach creates in both storage and processing, it will be more expensive and time-consuming to develop. Therefore, the dimensional star schema approach should be used instead when starting from scratch.

Recently, due to the popularity of cloud-based platforms, one big table modelling has become an alternative strategy. Tables are completely denormalised with one big table modelling, meaning all needed data is available in the same table. This approach can have some performance benefits when working tables in the cloud [6], as joining can become more expensive due to the need for moving data.

However, denormalization increases the storage requirements, but as cloud-based storage is cheap, it is often acceptable.

2.2 Data Integration

Many businesses have developed multiple separate systems that are used in parallel to each other. This can be an issue as it makes it difficult to find holistic information about the whole company. Therefore, data integration is needed to provide a system that gives the users access to all the data through one single platform.

One of the central issues that need to be tackled when integrating multiple source systems is that the target system needs to have one common data model. This is especially difficult since source systems are rarely designed to be integrated, which means that additional adaptations and transformations are needed so that all the data can be represented in one common model.

Integration can be done at different levels [7]:

- **Manually**, meaning that the persons interested in the data will manually access the different source systems and perform needed integrations by themselves.
- **Using a common user interface**, meaning that the data can be accessed using a common user interface, but the integration and unification are still to be done by the users.
- **Applications** can be used for accessing the different source systems and then creating a unified representation for the user.
- **Middleware** can be used to provide reusable functionality and reduce integration work needed on the application level.
- **Through uniform data access**, the unification of the data is done when the data is accessed by applications. A unified view is provided covering the physically distributed data. This has the benefit of many applications being able to use the same unified view, but as the integration is done at runtime, it can be resource heavy.
- **Using common data storage**, meaning that data from the different source systems is loaded into a new common data storage. To keep the data fresh new data from the source systems needs to be loaded periodically.

For data warehousing, a common data storage solution is used [7]. Therefore, we will take a closer look at this process.

The process of loading data into a data warehouse has three distinct steps: extracting the data from the source system, transforming the data, and loading the data to the target data warehouse. The order of these steps can differ depending on a few factors discussed in later chapters. Next, we will go through each of the steps individually.

2.2.1 Extract

The first step of any ETL task is to extract the data from various source systems so that it can then be processed further and loaded into the data warehouse. According to Kimball [2], data profiling is the first step to extract data efficiently. Data profiling entails investigating the source systems to find what information is needed for the data warehouse. It also plays an important role in getting an understanding of how much work will be needed to load the data into the data warehouse. Potential shortcomings and data quality issues in the source should be discovered at this stage. Finding them at an early stage will help make better estimates of the total development efforts for all parts of the pipeline. To help with profiling of the source system, documentation could be used, and therefore good quality source system documentation can be very valuable.

Two types of extraction techniques are needed when extracting data to the data warehouse, incremental and initial loads. Initial loads are used the first time data is extracted. Incremental loads are performed after the initial load has been performed at an interval depending on the business needs, usually daily or weekly. The benefit of incremental loads is that only the changed data is extracted, and therefore the extraction is less computationally heavy compared to an initial load. Some change capture method needs to be used to find the changed data [4], defines two types of change capture methods for these incremental batch-based extractions, timestamp-based extractions, and comparison-based extractions.

Timestamp-based extraction can be used, provided that the source contains timestamps that show when a record has been updated. If that is the case, these timestamps can be used to find records that have been updated after the previous extraction.

Comparison-based extraction can be used in cases where timestamps are not available. This method entails first creating and storing snapshots of the source. A current snapshot is then compared with the previous snapshot to find the changed records. This method can be very inefficient for large data volumes. Therefore, it should only be used when other methods are not feasible.

These two methods have the drawback of potentially missing information about states of records that have changed multiple times between the extractions. This is because source systems often keep only the most recent versions of records. If multiple changes have been made to one record, only the latest state will be extracted, and the other information is thereby lost. Real-time data extraction can be used to prevent these issues. Ponniah [4] describes three different methods for performing real-time extractions, using transaction log files, using database triggers, and modifying the source application.

The transaction log files of a database system can be examined to find updates, inserts, and deletes. This method works only for extracting data from database systems and not from flat files. One added benefit is that no extra strain is put on the source system as the log file would be maintained by the database management system regardless of if it is used for extracting data to a data warehouse or not.

Database triggers are stored procedures that can be set to run when a certain action is performed on the database. These stored procedures can then be used to write the desired data to a file that can later be used to extract the data. This method is quite robust but can require more development effort and put more strain on the source system because running the triggers is additional work that would not be needed if no data were extracted.

The source application can be modified in a way that can help with the data extraction by capturing the transactions in a separate file that can then be extracted into the data warehouse. This is work that the application must do in addition to updating the operational database, which may mean performance issues in the source application.

When the needed data has been identified, and a change detection method has been decided on, it is finally time to extract the data. When extracting, either a commercial ETL tool or an in-house tool can be used. Both [4] and [2] recommend using a commercial tool as they are better at adapting to changes in source systems and have good metadata capabilities.

2.2.2 Transform

Data transformation has the main goal of improving the quality of the source data and making it more user-friendly. Since many source systems are loaded into the same data warehouse, there are many unification measures needed. Data to the warehouse can be loaded from legacy mainframe systems and other newer systems, which can lead to issues with multiple naming standards, different data types, missing values, or inconsistencies between the systems. Therefore, there is usually a significant amount of transformation needed to load the data into a common model that has good data quality. [4] Lists some common tasks that are often performed during the transformation step of an ETL pipeline.

- **Data type conversions** may be needed if the column definitions for the same field are different between systems. In the data warehouse, one common type should be used, meaning that data might need to be cast to a different type.
- **Decoding of fields** is done when the source systems have cryptic codes that represent some useful information. One example is having 1 and 2 represent male and female in one system, whereas M and F may be used in another. These types of fields should be decoded, and more descriptive names like male and female should be used so the fields are easy to use and understand for everyone.
- **Performing calculations**, sometimes analysts want data that is not directly available from the source but can be calculated based on available data. An example of this would be the tax-free versus the tax-included price of a product. The source may only include the tax-free price and the tax rate. This information could then be used to calculate the price with the tax included.
- **Splitting information**, which is represented as one field in the source system, into multiple smaller ones in the data warehouse. An example would be the city and area codes that may be represented as one field in the source system. In the data warehouse, this should be split into two separate fields, as analysts may be interested in using only one of these fields.

- **Enriching the data** by joining with other sources. Sometimes all the relevant data cannot be found from the same source. Then it becomes necessary to join with other sources from either other internal source systems or even data from external vendors. By doing this, all the information that is needed for the final tables in the data warehouse can be gathered.
- **Summarization**, sometimes, the level of detail of the data in the source is unnecessarily large, and if it is certain this level of detail will never be needed, data can be summarized. The drawback with doing this is that if requirements change and more detailed data is needed at a later stage, more work is needed to make the changes. Therefore [2] recommends having the lowest level of detail available, which can then be summarized, as this method is better at answering future needs without making assumptions about what the use cases will be.
- **Deduplication**, sources may contain duplicates of a record, mostly because of mistakes. In the data warehouse, these duplicates should not be included. Therefore, deduplication should be done at the transformation stage.

According to Ponniah [4], many companies make the mistake of thinking that the data transformation step is simple. In reality, the transformation is often more complex than it seems and requires much time and effort. This is because integrating many source systems, all with different structures and different data quality challenges, requires a great deal of problem-solving and time.

When deciding what tool to use for the transformation, the options are either commercial ETL tools or coding manually with SQL stored procedures or other programming languages. [4] recommends using ETL tools as they can be more efficient, and less error-prone compared to manual coding. ETL tools also have the advantage of taking care of the metadata creation, whereas when coding manually, this must be done by the programmer. However, Ponniah says that even if a transformation tool is used, one should still be prepared to have to do some parts manually in cases where the tool is not able to meet the needs of the transformation.

2.2.3 Load

Loading the data is the process of writing the data from the source system to the data warehouse system. Similarly to extraction, either a full initial load, containing all the data, or an incremental load, containing only the latest changes, can be performed. Additionally, a full refresh can be done. A full refresh is similar to an initial load, the difference being that with a full refresh, data already exists in the target database, which needs to first be erased. When loading the data into the database, four different methods can be used [4]:

Load, this method always overwrites everything that is currently in the target table with the data that is being loaded. The load method is mostly used with initial loads and full refreshes when all data is loaded a once.

Append works similarly to load, with the difference that the table is not wiped clean before the data is loaded. This means that the incoming data is added to the already existing data. This may lead to duplicates if an identical record to the one being loaded already exists in the database. In some cases, this is acceptable. Otherwise, the rows in the current load that would lead to duplicates can simply be ignored. This method is commonly used if the initial or full refresh needs to be split up into smaller loads due to performance considerations. In these cases, the data can usually be split in a way that duplicates are not an issue.

Constructive merge compares the incoming data to the target data using the primary key. If the primary keys do not match, the data is loaded similarly to the append mode. If the keys match, the row in the target is marked as expired, and the new row from the incoming data is loaded into the target. This method is commonly used when doing incremental loads, as it preserves the history of changes.

Destructive merge is similar to constructive merge, with the difference that if the primary keys match, no new rows are inserted into the target. Instead, the matching row is updated with the latest information from the incoming data. This means that the previous state of the record is lost. Therefore, this method is mostly used when correcting erroneous data.

Once tables have been populated with data after the initial load, there are two options for how to deal with future data loads. One is to perform a full refresh of the table, completely wiping the table clean and inserting the new data. The other option is to perform an incremental load and update only the rows that have changed since the last time the table was loaded. A full refresh is easier to implement from a technical perspective, as no comparison is required. However, with large data volumes, significant performance improvements can be achieved by doing incremental loads instead. The more the rows change, the less efficient updating becomes, whereas the cost of a full refresh stays relatively constant. According to [4], the cut-off percentage when a full refresh becomes more efficient than an update is when between 15 to 25% of rows have changed. This is illustrated in *Figure 4*.

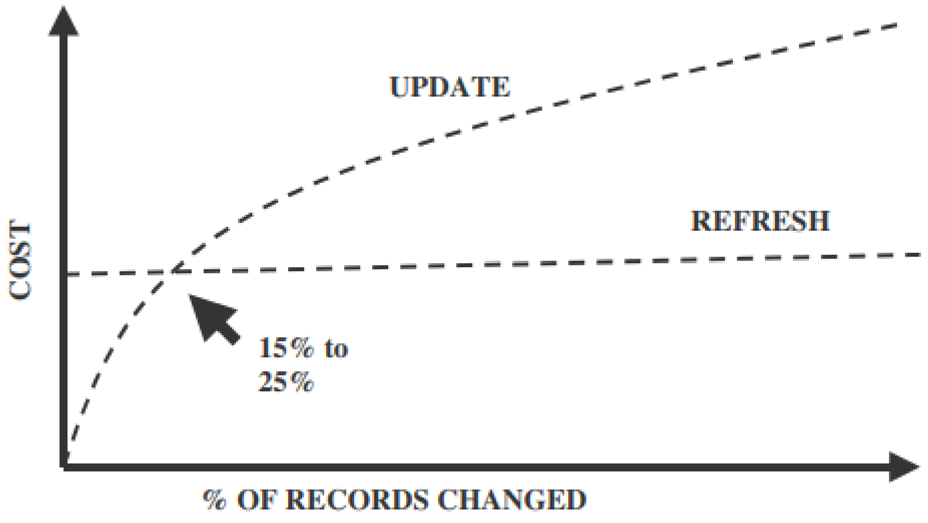

Figure 4 Cost of refreshing and updating [4]

2.2.4 ETL and ELT

The order in which the previously described extract, transform, and load steps are performed can vary depending on architectural and design decisions. There are two different possible orders extract, transform and load (ETL) and extract, load and transform (ELT).

ETL is what has traditionally been used when building data warehouse solutions. With ETL, separate systems are used for transforming the data and storing the data in the data warehouse. The data from various source systems is first extracted and transformed by one system. Finally, the transformed data is loaded into the data warehouse system, where business users can access it.

With ELT [8], on the other hand, the data is first extracted and loaded into the data warehouse before the transformation is done on the data warehouse system. Some advantages of this sequence are that the raw data from the source systems can be loaded to the target system faster as no transformation is needed in between. The raw data can also be made available for more advanced analysis to users across the organization. With the ETL approach, there are often limits on getting access to the source systems meaning that only the data warehouse developers can access these systems. Having the raw data staged in the data warehouse also decouples the transformation process from the extraction and load process allowing for changes to the transformation logic without necessarily needing to perform the time-consuming extraction process again. This is also an advantage in case of errors in the transformation step that causes the execution to halt. When using ETL, this would mean that no data is loaded to the data warehouse before the error is fixed. On the other hand, when using ELT, the source data is loaded to the staging tables, and the transformation step can then be more easily rerun after the error has been fixed.

Recently ELT has become a more popular method for doing data integration. The reasons for this are that ELT is better able to support unstructured and real-time data, which has become important in many organizations with the rise of cloud-based data warehouses [8].

2.3 Bitemporal Data

The objects that we store in our data warehouse often change over time. These changes can also be of value to keep track of, as they allow for seeing trends in how the data has changed over time. However, the data provided from source systems is often nontemporal, meaning that it can only provide us with the current state of an object. Therefore, we need a method for tracking the changes. The solution to this is to add timestamps to the tables. Unitemporal tables with valid times could be used for keeping track of past states. However, using these tables, it is still impossible to make corrections to previous states of the object as we receive more information or fix errors. Bitemporal tables can achieve this by using both valid and transaction times. This method also allows us to keep track of how the data looked previously in the data warehouse and means that no information needs to be overwritten [9].

Nontemporal tables only store the current information, meaning that they only have one row per object. If the data associated with the object changes, the record is updated, and the information about the previous states will be lost. This also means that the primary key consists of only an id column.

Below is an example of how the data in a nontemporal table changes after the data has been updated.

id	**data**
1	x

Table 1 initial data in a nontemporal table

Suppose that initially, the data was 'x'. Later on 2023-01-01, it changed to 'y'. However, no information about when this change happened and how the data looked before the change is stored in a nontemporal table.

id	**data**
1	y

Table 2 data in a nontemporal table after the update

Unitemporal tables can have multiple rows as the state of an object changes. To allow this, a validity period is used. The validity period describes when something is believed to be true in the real world about the modelled object. Using unitemporal tables partly solves the issues with nontemporal tables. Changes to an object's state are tracked using the validity information, and if there are updates, the valid timestamps can be used to see how an object has evolved in time. There is still one major limitation with unitemporal tables. They only work properly if the changes to the state are updated in the database immediately as they happen. In other words, the unitemporal tables lack support for making corrections or filling in information about a previous state of an object.

Suppose we have a situation where on the first of February, we find out that the state of an object in our database has changed on the first of January. Then there are two options: either have the new state be valid from the first of January or the first of February, but both are incorrect. If the first of January is used, we overwrite information about the previous state that was incorrectly in the database during the month of January. This is not ideal since decisions may have been made based on this data. Therefore, it is important not to overwrite it. In some cases, it may even be legally required not to make these types of changes [9]. On the other hand, if the first of February were used, that would not correctly reflect the situation as it was in the real world. Bitemporal tables can be used to solve this.

The example below illustrates how a unitemporal table can be used if all updates to the state can be made as they happen in the real world and no error corrections are needed:

id	valid from time	valid to time	data
1	2022-01-01	9999-12-31	x

Table 3 Initial data in a unitemporal table

Initially, the object had the data ‘x.’ On the first of January, we find out that starting from the first of January 2023, the data should be ‘y’

id	valid from time	valid to time	data
1	2022-01-01	2023-01-01	x
1	2023-01-01	9999-12-31	y

Table 4 Data in a unitemporal table after the update

The data is updated, and the table can be used to track changes to the object's state.

Bitemporal tables are similar to unitemporal ones, but instead of having one period describing when the information about an object is believed to be true in the real world, an additional transaction period is used. This period describes when the information was in the database. This means that updates can be made retroactively without losing any information about how the state was previously recorded in the database. The primary key for these tables consists of the id, valid from time, valid to time, transaction from time, and transaction to time.

Suppose we again have a situation where we, on the first of February, find out that the state has changed on the first of January. Now we will add a new row to the database with a valid from timestamp starting from the first of January. The transaction from times for this row would be the first of February. Thus, we can still see the correct data but also see how the data looked at an earlier point in time when needed.

id	valid from time	valid to time	transaction from time	transaction to time	data
1	2022-01-01	9999-12-31	2022-01-01	9999-12-31	x

Table 5 Initial data in a bitemporal table

Suppose we initially have the above information in the database. Then on 2023-02-01, we found out that starting from 2023-01-01, the data should have been 'y' instead of 'x'. Having two separate periods allows us to update the database with the correct information without overwriting anything. It is also possible to track both how we now believe the data to have looked during the full history of the object and how the data looked in the database at a given point in time.

id	valid from time	valid to time	transaction from time	transaction to time	data
1	2022-01-01	9999-12-31	2022-01-01	2023-01-01	x
1	2022-01-01	2023-01-01	2023-02-01	9999-12-31	x
1	2023-01-01	9999-12-31	2023-02-01	9999-12-31	y

Table 6 data in a bitemporal table after the update

To query this type of bitemporal table, constraints regarding the two periods should be added to the query. *Figure 5* shows how the current state of objects can be found. If one instead wanted to see how the objects looked at an earlier point in time, this can be done by simply changing the timestamp.

```
SELECT *
FROM example_table
WHERE
    current_timestamp() between valid_from_time AND valid_to_time AND
    current_timestamp() between transaction_from_time AND transaction_to_time
```

Figure 5 Example of a query against a bitemporal table

These types of conditions can also be used to create views on top of the bitemporal table that can be used to create either a nontemporal version (*Figure 5*) or a unitemporal version (*Figure 6)* of the bitemporal table. This is useful as the nontemporal and unitemporal versions are sufficient for many applications and more user-friendly to query.

```
SELECT *
FROM example_table
WHERE
    current_timestamp() between transaction_from_time AND transaction_to_time
```

Figure 6 Example query for the unitemporal version of the bitemporal table

2.4 Data Warehousing in the Cloud

Traditional data warehouses worked well for use cases like reporting and business intelligence. These use cases take structured data and create reports that follow up on key performance indicators (KPIs), such as profitability or the number of new customers. As data volumes grew and new use cases, like machine learning and data science, became more popular, the traditional data warehouse faced difficulties. Data scientists are often interested in unstructured and semi-structured data and use methods such as machine learning for their analyses. Data warehouses have limited support for unstructured and semi-structured data, and SQL is inefficient for machine learning applications as direct file access is more efficient for the algorithms used.

Data warehouses use proprietary file systems for the storage of the data. Proprietary storage locks organizations into one system, and changing the database management system (DBMS) would be challenging. Additionally, data warehouses are typically hosted on-premises meaning that scaling is difficult. Organizations need to pay for the usage at peak hours, while utilizing the system at a low percentage during other times. At the same time, cloud-based solutions can provide easy scaling both vertically and horizontally. Additionally, cloud-based solutions can often provide better guarantees for uptime and good disaster recovery services. As a result, data lakes and later data lakehouses emerged to solve the issues with data warehouses.

2.4.1 Data Lake

The core concept of data lakes is to store data in an open-source file format like the column-oriented Apache Parquet [10] and Apache ORC [11] or the row-oriented Apache Avro [12]. This allows for decoupling the storage and computation and using cheap and scalable cloud-based storage options. Additionally, the open-source file format supports fast ingestion and cheap storage of semi-structured and unstructured data such as text, audio, and video. Direct access to the data in an open file format can also better support machine learning, compared to accessing data through SQL, which is inefficient for machine learning use cases.

Apache Parquet has become one of the most used file formats in the cloud [13]. Parquet is an open-source and columnar file format, making it a good fit for storage in the cloud. Columnar storage allows for superior compression, and open-sourcing means that many tools support the format. In addition, storing the data in a columnar manner opens the possibility for clever encoding strategies that would not be possible with a row-based approach [14]:

- **Dictionary encoding** maps values to integers using a dictionary. This allows for storing only the integer values and then using the dictionary to find the true value of the column. The approach is similar to the concept of normalization from database design. Instead of storing the references in multiple tables using keys, the file system has a built-in dictionary to handle the mapping. This approach works best with larger fields, such as text fields that do not have many unique values.
- **Run length encoding** RLE is an option when the data contains multiple repeating values. RLE is a common data compression algorithm used in multiple file formats. Instead of storing the same value multiple times in a row, it is possible to store the value together with the number of consecutive occurrences. If the length of concurrent occurrences of the same value is large, this will reduce the file size. However, if the length of consecutive values is short, using RLE may increase the size.
- **Delta encoding** compares the current value with the next value and calculates a delta. This delta is often small compared to the actual value, especially when working with date and timestamp data, allowing for more efficient storage. For example, after calculating the deltas, the sequence 100, 101, 102, 103 would become 1, 1, 1, which can be stored using fewer bits than the original sequence.

Reducing the storage requirements by having efficient compression and encoding algorithms is not the only advantage of columnar storage. The columnar format can reduce I/O operations of OLAP queries, which often access a subset of the columns of a table. All the columns would need to be fetched with row-based storage, whereas columnar storage allows fetching only the specified columns [15].

However, the SQL query performance on data lakes was often insufficient. Data quality was also an issue since data lakes operate with a schema on read architecture, meaning there are low data quality guarantees. Additionally, data lakes are not able to support ACID transactions. ACID transactions have been a core concept in relational databases since the 1980s and played an integral part in their popularity [16]. The acronym stands for atomicity, consistency, isolation, and durability.

Atomicity guarantees that all changes made to a database are made as an indivisible operation. In other words, atomicity ensures that when a transaction containing one or more database operations is made on the database, the whole transaction will either succeed or fail. If all operations, part of the transaction, succeed, the changes will be applied to the database. However, if one or more of the operations in the transaction fail, the whole transaction will fail, and no changes will be applied to the database.

Consistency ensures that the state of the database can only transition from one consistent state to another consistent state, meaning that the state of the database will not change during a transaction. Instead, the state only changes after a transaction has been completed successfully.

Isolation allows multiple transactions to be executed simultaneously against the database in a way that their operations do not interfere with each other.

Durability guarantees that after a transaction has succeeded, the resulting state is permanent and capable of surviving potential system failures.

The lack of support for ACID transactions means that appending tables is the only possibility in the data lake. This, in turn, means that expensive and unnecessary rewrites are needed when data is updated [16]. The poor SQL performance, and lack of schema enforcement, meant that an additional data warehouse solution was often needed on top of the data lake.

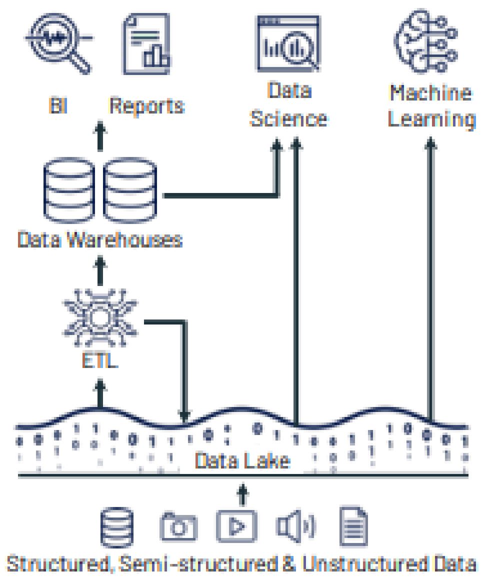

Figure 7 Two-system data lake and data warehouse architecture

A subset of the structured data from the data lake would be further transformed and loaded into a traditional data warehouse. This two-system architecture described in *Figure 7* is still prevalent today. It allows good query performance for business intelligence and reporting needs while supporting fast ingestion of semi-structured and unstructured data into the data lake. However, this architecture has many drawbacks [10]. Firstly, the cost of operating two different systems can be high, and there will inevitably be overhead in storage, as the same data will be in both the data warehouse and the data lake. Secondly, having consistent data between the data lake and data warehouse is challenging, requiring constant development effort.

2.4.2 Data Lakehouse

A data lakehouse combines the data lake and the data warehouse to benefit from both. A data lakehouse can be defined as a data management system that takes advantage of cheap cloud-based storage using an open-source file format and adds reliability, performance, and management features from the data warehouse by building a metadata layer on top of the file format [10]. By adding the metadata layer, the data lakehouse can offer features lacking from the data lake, such as ACID transactions, data versioning, auditing, and good query performance. The data lakehouse also adds features lacking from data warehouses, such as support for machine learning through direct file access and support for unstructured and semi-structured data. Because of this, data lakehouses remove the need for having a separate data lake and data warehouse. Instead, one system can be used for all the data needs, see *Figure 8.* In addition, having only one platform leads to a simplified architecture meaning a lower operating cost.

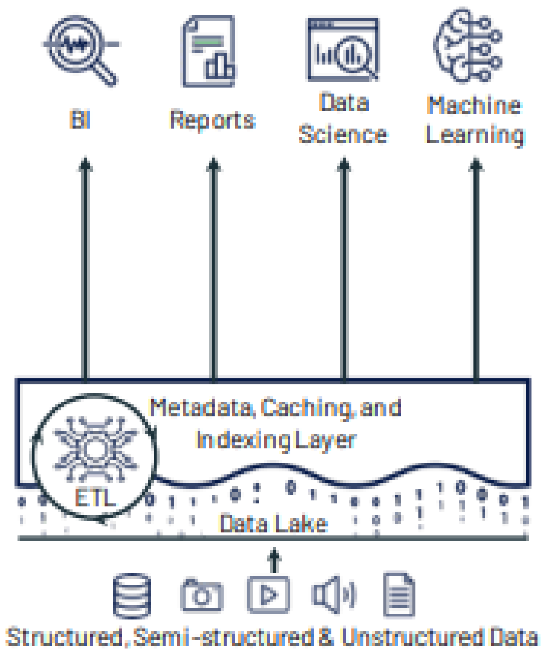
BI
Reports
Data Science
Machine Learning
Metadata, Caching, and Indexing Layer
ETL
Data Lake
Structured, Semi-structured & Unstructured Data

Figure 8 Data lakehouse architecture

At the core, the data lakehouse uses the same open-source file systems as the data lake. This allows the data lakehouse to utilize cheap and scalable cloud storage providers such as AWS S3 or Azure ADLS. The improvement over a data lake comes from adding an additional metadata layer on top of the open-source file system.

One example of this metadata implementation is Delta Lake. Every time an insert, delete or update is made to a Delta Lake table, records are written to one or more files. The Delta Lake transaction log records every change and the files that were written to during that change. The transaction log can then be used as a single source of truth specifying which files are part of a table. This solves the issue of failing jobs that result in partial files. Without a transaction log, it would be impossible to know if a file should be included in the table or if it is corrupted as the result of a failed job. In other words, the transaction log can provide a definitive answer to which files were part of a given table at what time in history [17].

The metadata layer provided by software like Delta Lake can then be used to provide features that were missing from the data lake, such as:

- Support ACID transactions with a metadata file that tells which files are part of which table version. The metadata file is updated only after the data file has been successfully written, meaning that if, for some reason, an operation fails, it will not be included in the transaction log and, therefore, not be a part of the table.
- Allow access control specification on a table, row, or columnar level [13] and manage these permissions with user groups. Delta Lake also provides audit logs to help monitor user actions.
- Logging all changes to the state of the table in the transaction log and not deleting any data makes it possible to revert to an earlier version of a table. This is a useful feature as it allows for looking back at how a table looked at an earlier point in time and rolling back to an earlier table version if accidental updates or deletions are made.

The open-source nature of the data lakehouse provides multiple benefits that do not exist when using a proprietary DBMS system, such as [13]:

Sharing files between organizations becomes easier as the open file format means that almost all the available tools can read them easily. Traditionally, CSV files or other complicated proprietary formats were used together with file transfer protocol (FTP) when sharing files. This led to the data consumer having to translate the file to the needed format. When using an open-source file format like Apache Parquet in a cloud environment, expensive and unnecessary file copying can be avoided. Due to the cloud's good security, privacy, and audit capabilities, many organizations can access the same data, reducing the need for unnecessary file transfers. The need for the data consumers to understand and translate the proprietary file formats is also removed, as there is comprehensive support across most data tools for formats like Parquet.

Using the open file format also gives users access to data catalogues developed and open-sourced by companies like Facebook and Netflix. The goal of data catalogues is to allow easy discovery of already existing data in the organization and give an overview of how the data is used and transformed. These open-source data catalogues also have good search and visualization tools reducing the times when different teams create separate solutions for the same problem.

The data lakehouse can also support direct API access for programming languages like Python and R, which are commonly used for data science and machine learning applications [13]. This is essential since the way data needs to be accessed in machine learning applications is not well suited for SQL. Additionally, the ability to travel back in time via the transaction log gives the ability to rerun models using the exact same dataset at a later point in time. This is something that otherwise would require saving many copies of the data.

The use of open software can provide flexibility and avoid vendor lock-in. Multiple query engines are available for file formats like Delta Lake, and switching between different ones is simple since they all support the same file format. If, at a later point, someone builds a new better performing query engine, switching over to using that one can be done with little effort. Switching between different DBMS, on the other hand, requires exporting data to a different file format and is, therefore, much more time-consuming.

For a data lakehouse to be successful, providing good SQL performance is essential. The open file format causes some limitations in the types of optimizations that can be done and makes achieving satisfactory performance more challenging. In a traditional data warehouse, more optimizations could be done using proprietary file formats. However, the three following techniques can still be used and have shown good results [10]:

- Frequently used data can be **cached** to faster storage mediums such as SSDs or RAM. Cashing is safe as the metadata layer guarantees that the cashed files are still valid. Additional optimizations, such as decompression, can be done while data is cached, as the file format limitations do not apply at that time.
- **Auxiliary data files** can be used to keep track of statistical information. The query engine can then use this information to optimize queries. An example would be keeping track of the minimum and maximum values of a column in a file. The statistics allow the optimizer to completely skip reading files

when a query has a where condition falling outside a file's minimum or maximum values.

- **Data layout,** meaning how the records are clustered together, can also optimize the query performance. In general, records that are often accessed together should be stored together, allowing for faster retrieval.

Based on the abovementioned techniques, Databricks created a query engine called Delta Engine. Compared to widely used cloud-based data warehouses, Delta Engine achieved comparable or better SQL performance at a cheaper price [10]. This shows that despite the limitations conforming to an open-source file system has on optimization strategies, satisfactory SQL performance can be reached.

Because the data lakehouse uses the same file format for storing objects as data lakes, it is easy to convert a data lake to a data lakehouse and receive the added benefits from the metadata layer. Therefore, researchers believe many organizations will convert to this architecture in the future, making the data lakehouse architecture prominent [10].

3. Existing Solution

This chapter will describe the target company's currently existing solution. The goal of the solution is to provide a monthly overview of all the active information as it existed on the last day of each month. This information should be "frozen," meaning that once a month data has been loaded, that data for that month should not change.

First, data is read from a normalized enterprise data warehouse, and then data is transformed using SQL stored procedures. The target schema for the process is a star schema with one fact table and four dimension tables connected to it. Because the dimensions are also used together with other fact tables, they are loaded daily and always reflect the current state of the underlying normalized data warehouse. The fact table, on the other hand, is loaded monthly on the first day of the month. See *Figure 9* for an overview of the current solution.

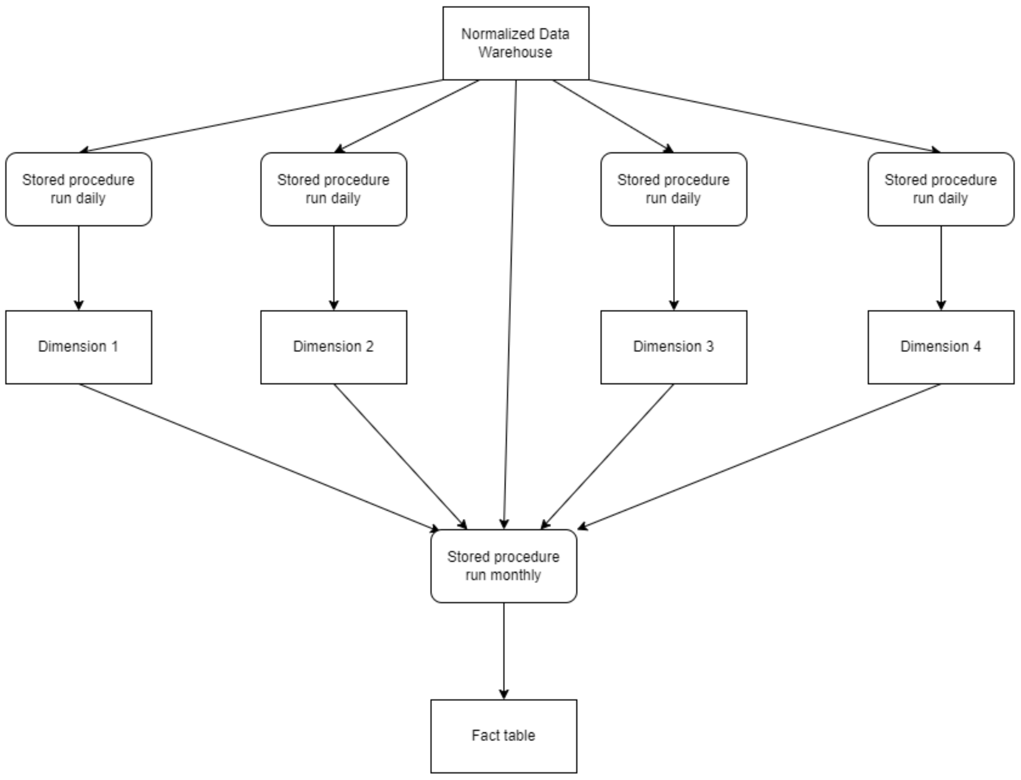
Normalized Data Warehouse
Stored procedure run daily
Stored procedure run daily
Stored procedure run daily
Stored procedure run daily
Dimension 1
Dimension 2
Dimension 3
Dimension 4
Stored procedure run monthly
Fact table

Figure 9 Overview of the current architecture

3.1 Extraction

The sources for the ETL process are an Inmon-style normalized data warehouse. In the normalized warehouse, data has already been integrated from various sources and use one common 3NF schema. Due to this, the extraction process is greatly simplified as the difficult task of reading many different sources using different methods, such as text files or database connectors, has already been done. All the data is on the same platform, and the extraction step is simple. The tables in the normalized data warehouse are all bitemporal, meaning that there is a validity period describing when something is believed to be true and a transaction period showing when information was recorded in the database. When creating the dimension tables, only the most recent representation of the data is used. Therefore, only the data with a transaction timestamp of '9999-12-31' is read. A query similar to *Figure 6* can be used to accomplish this.

The underlying data structure is built from many different source systems that all produce the data at different times. To always have as fresh data as possible available in the target schema, an additional filter on the source system is added to the loading process. This allows loading a source systems data into the target tables immediately when one source has been loaded into the underlying normalized schema.

3.2 Transformation

The transformation for the previous solution is done using SQL stored procedures. Stored procedures are SQL scripts that can be saved on the database and called at a later time to run a sequence of SQL statements. This allows for running the transformation directly on the database server without needing additional ETL tools.

The transformation process for the dimensions reads the data from the normalized data warehouse and applies different transformations to it. These transformations include processes like joining enumeration tables to find human-readable descriptions for codes. Surrogate keys are also created for the dimension tables at this point so that each unique combination of natural keys receive a unique surrogate key. Because the dimension tables are unitemporal, validity periods are used when creating the natural keys in addition to the modelled objects' unique identifiers. These keys are created incrementally and have no relation to the underlying natural keys, as described by Kimball [2]. This is done to make the data warehouse tables more resistant to changes that might happen to the natural keys in the source data.

The fact table is quite simple and currently has four dimension keys and one measure. The data is stored on the most atomic level and grouped by each month. Data that is active in the dimensions at the end of each month is joined with active data from all the other dimensions and measures monthly. An example of what one year's data might look like in the fact table can be found in *Table 7.* This allows for easily finding rows that were active during a given month, and monthly trends become easy to find by grouping by the month. Using this approach means that there is a significant amount of redundant data since, many times, a given row is active for a significantly longer time than one month. An example of this is months 2022-01 – 2022-04, which all have the exact same data and only the month changes. Another alternative would be to provide validity timestamps to show when a combination of rows is valid, but that would make querying for monthly trends harder. This becomes a trade-off, and providing an easy-to-use solution for the end users outweighs the increased computing and storage costs for the data warehouse.

Month	**dim1_key**	**dim2_key**	**dim3_key**	**dim4_key**	**measures**
2022-01	1	1	2	1	x
2022-02	1	1	2	1	x
2022-03	1	1	2	1	x
2022-04	1	1	2	1	x
2022-05	1	1	1	1	y
2022-06	1	1	1	1	y
2022-07	1	2	1	1	y
2022-08	1	2	3	1	y
2022-09	2	2	3	1	z
2022-09	2	2	3	1	z
2022-11	2	2	3	1	z
2022-12	2	2	3	1	z

Table 7 Example of how the data in the fact table might look like

Due to the high amount of data processed, the chosen algorithms, underlying data structure, and compute infrastructure need to work well together so that the computation can be done in an efficient and timely manner. Especially when concerned about the fact table with billions of rows, the dimension tables are slightly smaller and faster to process.

3.3 Load

Like the extraction process, the loading process for this pipeline is straightforward, since the same platform is used for extraction, transformation, and loading. Loading the target dimension tables is done by comparing the new state calculated in the transformation step with the current state in the target dimension. A destructive merge is then performed to reduce the number of rows that need to be inserted and thus speed up the loading time.

The approach for the fact table is slightly different since the data needs to be frozen. Instead of the destructive merge, the table is appended with the data from the latest month. Using this approach, no data is ever overwritten, and the table is, in a sense, frozen.

3.4 Shortcomings of the Current Solution

The current solution has been in production for about one year. During this time, a few issues have been discovered, mostly related to the tool used for the way the frozen data is implemented.

Some drawbacks of using stored procedures are that DDL statements, such as create table statements, cannot be defined in the stored procedure script. Instead, they need to be done manually, making deploying changes a tedious and error-prone process. The developer must remember to also update all the tables used by the stored procedure. Stored procedures also lack modularity leading to code duplication when similar operations are performed in many different places. An example of this, from the solution that is studied in this thesis, is the code for creating the surrogate keys for the four dimensions. This code is identical in all the dimension stored procedures, with the only difference being the columns that are used as natural keys.

The stored procedure approach also makes developing and testing difficult as different tables are used depending on if one is developing, testing, or deploying the change in production. This means that versions of the stored procedure are needed for the different environments, and it becomes a manual process for the developer to find the correct table names for the different environments as well as remember to make the changes when the code is run in a different environment.

One requirement for this solution is for the data to be frozen, meaning that after loading the data into the target tables, it should not change. This was achieved by loading the fact table once a month with the data for the latest month and leaving the other, previously loaded months, untouched. However, this solution has several issues. Firstly, only the fact table is loaded in a way that guarantees no back-dated changes, the dimensions are still loaded daily, and if back-dated changes happen, they are reflected in the dimensions. This means that if a correction like removing or adding a record is made in the data source, it is also reflected in the dimension tables but not in the fact table. As a result, the fact table and dimension become unsynchronized and referential integrity cannot be guaranteed. Secondly, the time when the monthly run of the fact table is executed affects the result, meaning that if there is a delay in the execution of the fact table stored procedure, the data will not be correct as it reflects the data how it looked at the moment of execution and not how it looked at the last second of the month as intended. Lastly, making changes to the fact table, like adding a measure, at a later point in time becomes difficult or impossible. A full refresh of the fact table would be needed to accomplish this, but as discussed, a full refresh would mean that backdated corrections would now be included in the table, and therefore the state of how the table looked historically is lost.

Another issue with the current architecture is vendor lock-in due to the fact that the transformation is run on an on-premises Teradata data warehouse. Since there are some differences between SQL dialects, the process of changing the DBMS system is non-trivial. Changing to a new DBMS would require rewriting the code in a way that is compatible with the new system and, most importantly, storing the data in a new location, since the storage and compute are coupled in the current situation. Lately, this has become a larger problem as the capacity of the current on-premises solution has become insufficient due to growing usage. This, in turn, slows down development because testing takes a long time due to the high load slowing down data processing. Long testing times mean that the time to validate the code and find bugs grows larger, and therefore the overall development speed is impaired.

Therefore, the new solution will run in a cloud environment that uses an open-source file format for data storage.

4. Implementation

An improved solution was needed to solve the issues discussed in the previous chapter. This solution was built using dbt, a modern tool for the transformation part of ELT pipelines promoting software development best practises, such as documentation, testing, code reusability, and code portability. Additionally, the new solution is run in a scalable cloud-based environment using Microsoft Azure and Databricks to alleviate the scalability and performance issues.

The development of the new solution was done in two steps. First, the old stored procedures were converted to dbt code, the new measures were added, and the issues with the frozen data were solved. This solution was still run and tested using the on-premises Teradata platform. Secondly, the data and the dbt code were migrated and executed using the cloud-based Databricks platform to alleviate the scalability issues.

Figure 10 shows an overview of the final architecture. First, the source data is replicated to the cloud. Once the data needed for the transformation process is available in the cloud, the transformation is done using Databricks and dbt. Lastly, the transformed data is replicated back to the on-premises platform, where different data consumers can access and use the data according to their needs. The replication between on-premises and cloud is needed, as providing data only in the cloud would make the new solution difficult to use together with other data that is currently only available on the on-premises platform.

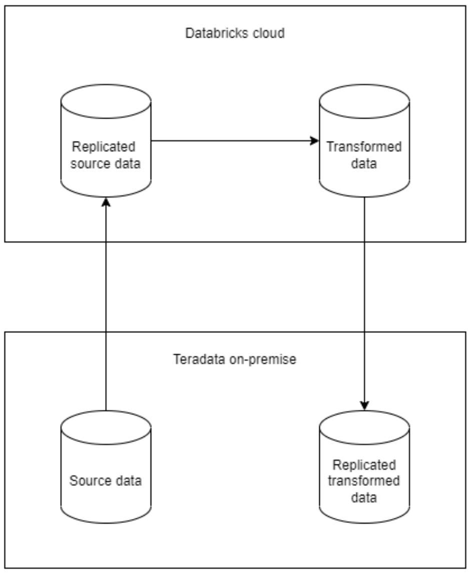

Figure 10 Overview of the relation between on-premises and cloud in the ETL process.

In this chapter, the solution will be described in more detail, starting with describing the tools used and then discussing the ETL process of the new solution.

4.1 Data Build Tool

Data build tool (dbt) is a tool for the transformation part of ELT or ETL, maintained by dbt labs [18]. Dbt simplifies the transformation process by allowing developers to write the business logic using only SQL select statements. Adapters that are available for most of the common DBMS take care of the tedious DDL [19]. Additionally, Jinja templating can be used [20] to enhance the SQL code. Before running the code on the database, dbt will compile the Jinja and use the adapters to create the needed DDL statements. The compiled code is then run on the DBMS system, meaning that dbt does not require any additional platform or moving of data.

Dbt Labs offers two separate products: dbt core and dbt cloud. Dbt core is an open-source version that can be run through the command line. All essential features of dbt are available in this version. Dbt cloud is a paid version that provides additional features such as a web-based UI with an integrated development environment, version control, and job orchestration possibilities. The project discussed in this thesis uses the free dbt core version.

```
{{ config(
    materialized="table",
    schema="marketing"
) }}

with customers as (

    select *
    from {{ ref('stg_customers') }}

),

orders as (

    select *
    from {{ ref('stg_orders') }}

),

final as (

    select *
    from
        customers
        left join orders on
        customers.customer_id = orders.customer_id

)

select *
from final
```

Figure 11 Example of a dbt model before it is compiled

Developers can write dbt code by defining model and configuration files. *Figure 11* shows an example of what a model file in dbt might look like. A model file contains an SQL select statement and an optional config block. The config block defines information about the model, such as materialization strategy, schema, or pre- and post-hooks. Alternatively, the configuration information can be defined in YAML [21] files. Materialization strategy determines if the compiled SQL code should create a view or table in the database [22]. The schema variable defines in which schema the model should be materialized. The use of pre- and post-hooks allows users to define SQL code that should run before or after the SQL for the model. Some examples where pre- and post-hooks may be helpful are collecting statistics or logging metadata about model runs.

The SQL select statement written in the model is where all the business logic is located and is the basis for creating the final tables or views in the database. In addition to writing plain SQL, the use of Jinja templating [23] means that features such as if-statements, for-loops, and macros are available for developers to use. *Figure 12* shows the final SQL statement that dbt would execute against the database after the model from *Figure 11* was compiled and run.

```
create table analytics.customers as (

    with customers as (

        select *
        from analytics.stg_customers

    ),

    orders as (

        select *
        from analytics.stg_orders

    ),

    final as (

        select *
        from
            customers
            left join orders on
            customers.customer_id = orders.customer_id

    )

    select *
    from final

)
```

Figure 12 Example of the SQL code produced after the model has been compiled

The new features enabled by using dbt mean that many of the issues with writing plain SQL and running it through stored procedures can be solved.

Having adapters that automate the DDL means that the developer only needs to write SQL select statements. The tables or views are created automatically using "create table as" or "create view as" syntax. This means that manually updating the table definitions after making changes, such as adding a new column, is no longer needed. Thus, development and deployment become easier and less error-prone.

Using Jinja templating allows for writing code that follows software development best practices, such as using for loops, variables, and, most importantly, macros. Macros allow for reusing common code across models by defining the code as a macro and then calling it in multiple models. This promotes the software development best practice of "don't repeat yourself" (DRY), meaning that the code is reusable and the same piece of code is not repeated multiple times. Achieving DRY code using SQL together with stored procedures has proven difficult.

Another feature available in dbt is the ref-function. The ref-function is an added abstraction that allows users to reference dbt models directly by their name instead of referencing database and table names. This allows for dbt to track dependencies between the models across the dbt project. The added abstraction also makes it possible to easily materialize the tables in different schemas or with different table names depending on whether the table is materialized in a development, test, or production environment. This makes development, testing, and production deployments significantly easier, as the developer does not need to remember to update table names when switching between the different environments manually. Additionally, there is built-in visualization of the resulting directed acyclic graph DAG, making it easy for developers to understand how the models in a project depend on each other. *Figure 13* shows an example of one such graph.

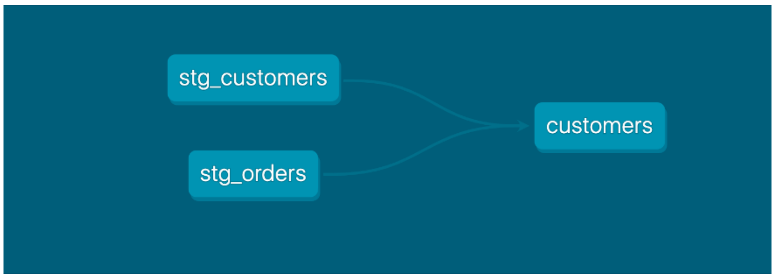

Figure 13 Example of dbt DAG

To ensure the quality of the data in the models, dbt makes it possible to define tests [24]. Tests can be defined on the column or model level in a model-specific YAML [21] file. Dbt comes with basic tests that can be used to validate the results of models. Some examples of tests are not null tests, uniqueness tests to check that a column or combination of columns is unique in a model, and referential integrity tests. In addition to these basic tests that come predefined, users are also able to define their own more specific or complex tests using SQL. Tests for a model can then be run using the command line at any time to check the data quality.

Dbt also offers easy-to-use documentation features, which allow documenting models and columns using markdown syntax. The documentation is defined in the same YAML files as where tests are defined. This documentation can then be visualized in an intuitive web-based UI together with the DAG, and source code for the models. This UI is also available when using the core version of dbt. Having the documentation interwoven with the models can potentially make it easier for new developers to gain an understanding of the code in a faster manner compared to having the documentation in a separate system.

Because all the code needed for creating models is available in the dbt project and the adapters take care of creating the DDL statements, deployments become simple. All that is needed is to run a dbt command that defines which models should be run and what the target environment is. Dbt will then automatically detect the order in which the selected models should be run based on the DAG created by using the ref functions and run the compiled models against the DBMS. When comparing this to stored procedures, dbt is faster, and the chance for errors is significantly lower. Developers no longer need to manually create tables or change the database and table names in the code to match the desired environment. All the code needed to run dbt is available in the dbt project and can easily be integrated with Git. This makes it possible to automate deployments. For example, new and changed models can automatically be run when they are merged to the main branch using CI/CD pipelines. This process is more closely described in chapter *4.8 Continuous Integration and Continuous Delivery*

For dbt to be a worthwhile investment, conversion from legacy SQL scripts to dbt should be easy and efficient. In many cases, the conversion from SQL stored procedures to dbt is simple as the SQL code that already exists can be used as a base. However, while switching to dbt, it is also recommended to make the following changes to the code so that it becomes more modular and reusable [25].

- Switch to using common table expressions CTE syntax when writing the SQL queries in favour of using subqueries that can become difficult to debug. CTEs can be read in a top-down manner where the blocks below reference the blocks above. This often becomes easier for humans to read compared to subqueries that build on each other starting from the middle of the code.
- The table references need to be updated to use the ref function. This is needed for dbt to resolve dependencies when running models and create the

DAG. Additionally, this makes deployments more straightforward and allows dependency tracking.

- Restructuring the code to smaller logical models is beneficial, as it promotes reusability between different solutions that may partly use the same data.
- Data manipulation language DML statements need to be rewritten so that only select statements are used. This is needed, since dbt only allows select statements in the models. Luckily, most update or delete statements can be rewritten using joins and filters.
- Avoiding dialect-specific SQL functions is recommended to make switching between different underlying DBMS systems easier. This allows for an easier time if the platform on which the dbt models are run is changed in the future.

If these steps are followed, the result should be code that is easy to run on any DBMS and is written in a modular way so that parts can be reused when building new models.

4.2 Databricks SQL

Databricks [26] is a data lakehouse platform that provides a unified interface for business intelligence and machine learning. Databricks can be used with many of the most common cloud providers, such as Microsoft Azure, Amazon Web Services, and Google Cloud [27]. In this thesis, we will focus on the features provided with the Microsoft Azure version of Databricks, as Azure is the cloud provider used by the target company. When using Databricks, Microsoft Azure provides the cloud resources, but Databricks automates the deployment of these resources [28]. This approach makes it easier for end users to manage cloud resources.

Most of the services provided by Databricks are based on open-source projects, for which Databricks has created improved versions. One of the provided services is Databricks SQL. Databricks SQL is an enterprise data warehouse built on top of the Databricks Lakehouse platform. The main feature that Databricks SQL provides is an optimized, spark-based computing platform, provided in the cloud, called SQL warehouse.

The Databricks SQL warehouse provides two different methods to interact with it: a web-based interface and an API. The web-based interface allows developers and analysts to create and execute SQL queries against the Databricks warehouse and see or even visualize results. The interface also allows users to visualize query profiles that can help developers identify bottlenecks and improve query performance [29]. The query profile can be used to split the query into subtasks and see metrics such as time spent, memory used, and rows affected for each of these subtasks. This, in turn, makes it easy for developers to identify which part of the query is the problematic one, such as an unnecessary full table scan or a poorly constructed join and make changes accordingly.

Databricks also offers Unity Catalog, a data governance solution aimed at the data lakehouse [30]. The Unity Catalog is a central platform for managing access to all data available in the whole Databricks environment, which may include multiple workspaces for different applications such as data warehousing or machine learning. Standard ANSI SQL syntax can be used to manage access to the Unity Catalog. This is beneficial as many administrators are familiar with the syntax from other DBMS systems. The Unity Catalog has a three-level naming hierarchy instead of the two-level hierarchy commonly found in other DBMS systems to allow for a clearer separation of data assets. Instead of the traditional *database.table*, Unity Catalog uses *catalogue.schema.table* for defining and accessing tables. This allows databases, or schemas as they are referred to in Databricks, to be divided into separate catalogues, for example, giving each team or unit their own catalogue. The benefit of this separation is a clearer structure when many different users and teams use the same Unity Catalog.

Since Databricks is cloud-based, there are many options for scalability when choosing the size of SQL warehouse clusters. This means that the size of the SQL warehouse can be chosen so that it fits the needs of the users. Additionally, Databricks offers autoscaling features that allow the number of warehouse clusters to increase and decrease dynamically depending on the load [31]. These features mean that fewer computing resources will go to waste when compared to an on-premises solution that has a constant amount of computing power allocated to it and is not capable of adapting to short-term changes in the load.

4.3 Extraction

Similarly to the previous solution, the bitemporal and normalized data warehouse is used as a source for the extraction part of the ETL process, making it relatively simple. However, as the final solution will move the processing to the cloud, the source data also needs to be moved to the cloud before the processing can be done there. This adds complexity to the extraction process.

Moving large amounts of data over the internet can be time-consuming and expensive. A change detection system can be used when extracting the data so that only the changed rows are moved to the cloud environment to reduce the time and cost of the extraction process. Finding the changed rows can be done using the transaction timestamps of the bitemporal tables in the normalized data warehouse or by using database triggers. There are many different tools available that can be used to move data between different systems. Azure Data Factory ADF is an example of such a tool [32]. At the target company, the whole process of replicating the data between the on-premises Teradata system and Databricks is done using an in-house tool that is based on ADF. Alternatively, other commercial change data capture (CDC) tools could be used to achieve the same result as the current in-house tool. Having a CDC tool means that developers do not need to worry about data replication to the cloud and can use the tables in the cloud the same way as they would on the on-premises platform.

4.4 Methods for Freezing Data

One of the improvements needed in the new solution was creating a method for freezing the data. To accomplish a frozen solution that can be rerun at any time and still produce the same result every time, it is not enough to just read the currently active rows, as was done in the previous solution. Instead, we need to look at both the valid and transaction times. The intersection of these two periods is the data as it was recorded in the database historically. Studying all the data for one entity, the most recent representation of the data can be found when selecting rows with a transaction to timestamp equal to 9999-12-31. However, those rows also include corrections made to the data after it was valid, and new corrections may be made in the future. This means that the data during one validity period may change, and as a result, tables built using this data as a starting point would receive different values depending on when they were loaded.

To solve this issue, we can take the intersection of the validity and transaction period, which represents the data as it was recorded in the database when it was valid. The transaction period is never retroactively updated, and thus loading tables with this intersect method means that the result will be the same no matter when the load is executed.

Table 8 and *Table 9* show an example of this. In this example, the data was retroactively corrected on 2023-01-01. As we can see, when looking at the result of the intersect operation, this change is not present in the resulting frozen data. Additionally, new changes to the current validity period that may be applied in the future would not intersect with the current validity periods. Thus, these changes would not be included. Lastly, it is worth noting that because the row with the validity period from 2021-01-01 to 2022-01-01 was inserted into the database late on 2021-02-01, this row was not recorded in the database during the period 2021-01-01 - 2021-02-01 and thus the valid from timestamp becomes 2021-02-01.

id	**valid from time**	**valid to time**	**transaction from time**	**transaction to time**	**data**
1	2021-01-01	2022-01-01	2021-02-01	2023-01-01	x
1	2022-01-01	2023-01-01	2022-01-01	2023-01-01	x
1	2021-01-01	2022-01-01	2023-01-01	9999-12-31	y
1	2022-01-01	2023-01-01	2023-01-01	9999-12-31	y

Table 8 Example data for one entity in the bitemporal data warehouse

id	**valid from time**	**valid to time**	**data**
1	2021-02-01	2022-01-01	x
1	2022-01-01	2023-01-01	x

Table 9 Example of the intersection between validity and transaction periods.

In practise, this approach was implemented by creating intermediate dbt models based on the tables from the normalized data warehouse. These models have the same data as their underlying bitemporal tables but are unitemporal with an additional timeline identifier. To be able to compare the frozen and the current versions, three different timeline identifiers are used. The first represents the latest representation of the data and can be found by filtering on transaction to timestamps equal to 9999-12-31. This is the same approach that was used in the original implementation. The second timeline is the frozen timeline, where the intersection of the validity and transaction periods are done. Finally, the third, frozen-adjusted timeline is similar to the frozen one but allows including bugfixes in the frozen history.

In many cases, some crucial bugs in the loading process of the underlying data warehouse are found and fixed. When these bugs are fixed, many rows will receive new transaction times starting from the time of the fix. If the frozen timeline is used, these bugfixes will not be included, as the new transaction periods do not intersect with the validity periods. However, some bugfixes are important and should be reflected even in the frozen history for the data to be usable. This is the reason behind including the third timeline in the intermediate tables. To achieve this, entities that should be fixed are identified and included in a dbt seed file. When creating the intermediate tables, an entity's transaction timestamps are checked, and known bugfixes are identified by joining the dbt seed file. These entities are then further processed by moving the transaction from timestamp backwards. Previous transactions, valid before the bugfix, are discarded. Lastly, the intersection of the validity periods and the newly updated transaction period are calculated just like with the frozen timeline. This results in a representation of the data as it was recorded in the database at the time of the bugfix. The data will also remain consistent over multiple loadings. However, entities that are fixed in this manner lose the history before the bugfix, a drawback that is necessary to ensure good data quality.

Timestamp of bugfix	**Description**
2023-01-01	Fix bug in the loading process
2023-03-01	Fix another bug in the loading process

Table 10 Example of a seed file including bugfix timestamps.

Table 10 shows an example of a seed file used to record bugfixes. The seed file is just a CSV file where the timestamps of bugfixes that should be included are written. Then, looking at *Table 11,* we see the next step of creating the frozen-adjusted timeline. In this example, since the entity has a transaction timestamp equal to the bugfix timestamp, the frozen-adjusted timeline is moved back to low date, and all transactions valid before the bugfix are discarded.

id	valid from time	valid to time	transaction from time	transaction to time	data
1	2021-01-01	2022-01-01	1900-01-01	9999-12-31	x
1	2022-01-01	2023-01-01	1900-01-01	9999-12-31	x

Table 11 Example of the frozen-adjusted timeline after rows active before the bugfix have been removed and the transaction from timestamp updated

As we can see from *Table 12*, the resulting data is what was recorded in the database at the time of the bugfix (2023-01-01). If late changes came into the table after the bugfix, those would not be present in the final data after the intersection, meaning this timeline is idempotent.

id	valid from time	valid to time	data
1	2021-01-01	2022-01-01	x
1	2022-01-01	2023-01-01	x

Table 12 example of the frozen-adjusted timeline after the intersection

Table 13 shows an example of all the different timelines assuming that the starting point was the data from *Table 8* and that bugfixes are recorded in *Table 10.*

id	valid from time	valid to time	data	timeline id
1	2021-01-01	2022-01-01	y	1
1	2022-01-01	2023-01-01	y	1
1	2021-02-01	2022-01-01	x	2
1	2022-01-01	2023-01-01	x	2
1	2021-01-01	2022-01-01	x	3
1	2022-01-01	2023-01-01	x	3

Table 13 Example of the three different timelines for frozen data

Having intermediate models containing the three timelines makes the processing in the transformation step significantly easier. All needed data is in an easy-to-use format, and the intersection and corrections to the timelines are already done.

4.5 Transformation

For the transformation part of the ETL process, dbt is used to create models that are compiled and executed against the database. As the data replication process to the cloud is quite time-consuming, this project was done in two steps. First, the dbt code was executed and the transformation logic was tested on an on-premises platform. Later when the code was working, and the logic had been tested, the processing was moved to the cloud, and a Databricks SQL warehouse was used for the transformation. The cloud migration is more closely described in section

4.9 Cloud Migration.

The existing stored procedures already had most of the business logic for the transformations implemented and provided a good base for the conversion to dbt. The process of creating the dbt code for the dimensions and the fact table closely followed the steps outlined in chapter *4.1 Data Build Tool.* The steps that were done to convert the SQL stored procedures to dbt modes were:

- Converting the logic in the stored procedures to CTEs that can be used in dbt models. This means that insert, update, and delete statements need to be rewritten. Insert statements could be rewritten using select and union statements. Update statements could be rewritten using joins and case statements.
- The long stored procedure scripts for dimensions were also split into smaller parts by creating one dbt model that does most of the transformation and a separate one for creating and storing the surrogate keys.
- Instead of directly referencing database and table names, the dbt ref and source functions were used. Using the ref function to reference other dbt models and the source function to reference source tables from the database allows for generating a DAG that can be visualized for developers and used by dbt to run models in the correct order.
- Implementing the frozen data. After the intermediate frozen tables discussed in the previous chapter were done, this step is rather simple. All that is needed is to add the timeline id column to all joins in the dbt models and update the ref function to point to the newly created intermediate frozen tables instead of the bitemporal sources that were used previously.
- Adding two new measures for counting the number of objects and products to the fact table. This was done by creating common product and object definitions for all the different source systems and using them to be able to provide new columns for object and product count.

- Using hash keys instead of incremental keys for the surrogate keys. This change was done to ensure that the same natural keys always map to the same surrogate key. The creation of the hash keys is done using a common macro used by all dimensions.

For the materialization of the intermediate frozen tables and the target tables, an incremental materialization strategy is used [33]. Incremental materialization is an additional materialization strategy provided by dbt in addition to view and table materializations. This strategy is useful in situations when using a view means that the performance of downstream models would be lacking or using a table materialization becomes inefficient as the number of rows grow large. In these cases, using the incremental materialization strategy makes it possible to define filters so that only a subset of the data is read. The subset of data is usually the new rows that have been loaded into the upstream tables since the latest load. A unique key is also defined. Dbt will then do a destructive merge between the incremental table and the subset of data calculated using the unique key. This approach greatly reduces the number of rows that need to be processed on daily loads, as the number of changes from the source system is only a fraction of the total data volume.

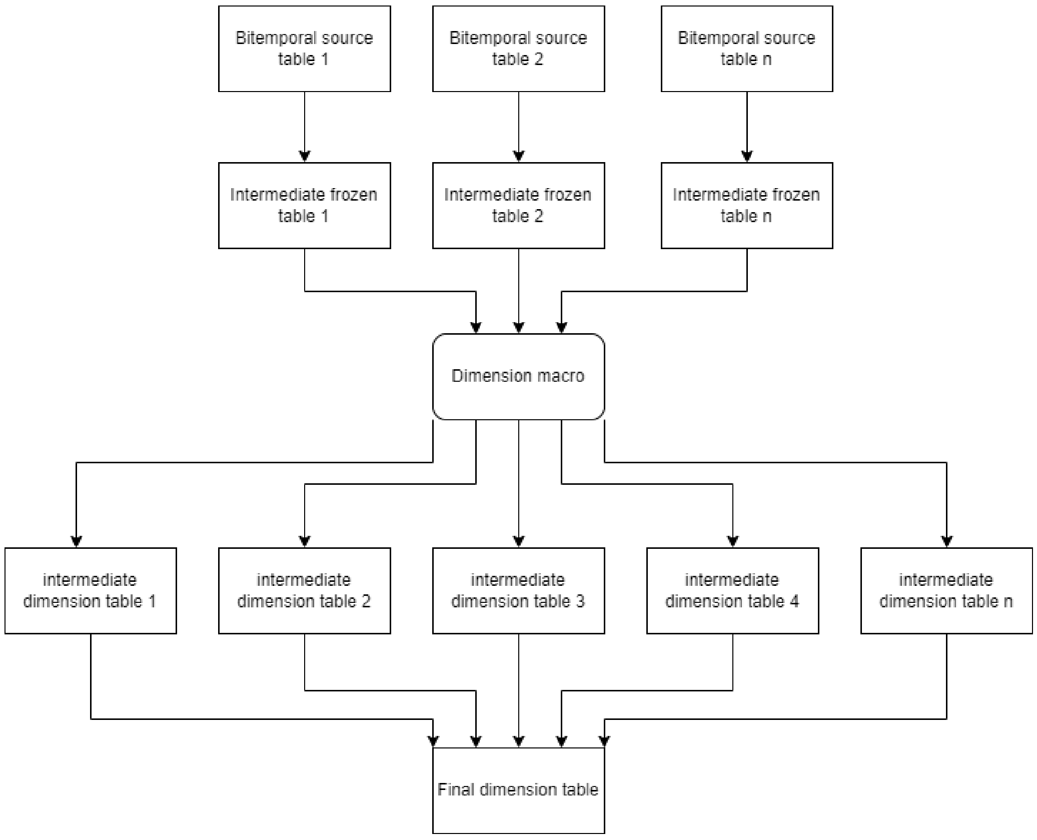
Bitemporal source table 1
Bitemporal source table 2
Bitemporal source table n
Intermediate frozen table 1
Intermediate frozen table 2
Intermediate frozen table n
Dimension macro
intermediate dimension table 1
intermediate dimension table 2
intermediate dimension table 3
intermediate dimension table 4
intermediate dimension table n
Final dimension table

Figure 14 Description of the loading process for one dimension

As with the previous solution, different source systems produce data at different times. Therefore, the target tables need to be loaded at different times. One option for this would be to use variables defined on the command line when running dbt models [34] together with incrementally materialized models. However, dbt needs to do a complete reparsing of the whole project if variables are used, which is a slow process. This, in turn, makes the running of multiple source systems at different times slightly harder. Instead of using a variable, one intermediate dbt model is created per source system. To reduce code duplication, all these models use the same macro that takes as an input argument the source system for the model. This model can then be run at any time without needing to use command line variables. Finally, all the source system-specific intermediate tables are combined into one final table using a dbt model that takes the union of all source system-specific intermediate models.

An overview of the whole process, starting from the source to the target table, can be seen in *Figure 14.* This figure shows the process for one dimension, but the process is very similar for the fact tables as well, the only difference being that the fact table uses the dimension tables as a source in addition to the bitemporal tables.

4.5.1 Data Quality Testing

To guarantee the quality of the data, testing the data is important. Errors may occur at many stages during the ETL process, starting from the source data all the way until the final tables are loaded. The previous solution was severely lacking in tests. Testing was only performed manually by business users and developers. Additionally, table definitions could be used to enforce some requirements like not null fields and data types.

Using the testing capabilities of dbt, several tests were incorporated into the models starting from the source data, intermediate models, and finally, testing the target tables. Most of the tests that are currently used are basic ones that are included in dbt. Some examples are testing for not null values, which is needed since not null columns cannot be defined in the table definitions when using dbt due to the tables being created with the "create table as" syntax. Referential integrity tests are also done to ensure that all keys found in the fact table also find a match in the connected dimensions. Accepted values are used to check that fields, where only a certain number of distinct values are allowed, do not contain any unexpected values. Uniqueness tests are used to check that tables do not contain any duplicates. The overlapping validity test is an example of a self-defined test that is used to check that no two rows for the same entity are valid at the same time.

These tests can be run during development to ensure that new changes do not cause any unintended side effects. Additionally, tests are run automatically when deployments are made, as well as during daily runs, so that potential data quality issues are found and can be remedied as soon as possible.

4.6 Loading

The process of loading the transformed data into the target tables has not changed much from the old solution to the new one. The processed data volumes for the solution are very high. Therefore, it is necessary to have some form of a change detection method.

When loading data for the first time or after making changes to the code, a full refresh is done. On consecutive runs, incremental loads are done instead. The incremental loads of the dimensions are done by reading all the data from the intermediate frozen tables. This data is then transformed and loaded into the source system-specific intermediate tables. However, when loading the final dimension table, a change detection macro is used to compare the rows in the intermediate table with the rows that are present in the final dimension table and only load the rows that are new or have changed. This reduces the number of inserts needed and thus speeds up the processing. For the fact table, the incremental logic is simple. The data is frozen and reported monthly, meaning that there can never be retroactive changes to a month after it has been loaded. This allows just loading the latest month when doing an incremental load.

Once the transformation is done, and the data is available in the cloud environment, the data still needs to be brought back to the on-premises data warehouse. This step is needed for the time being due to many users and solutions needing to use additional data that is only available in the on-premises solution. Having all the data available in the same place makes using it significantly more convenient. Therefore, it is not feasible to only provide the data in the cloud before all other data has been migrated to the cloud. Similarly to the replication from on-premises to the cloud, moving the data from the cloud to on-premises is handled by an ADF-based in-house tool, meaning that it is not something developers need to worry about.

4.7 Scheduling

To automatically run and monitor jobs, a scheduling tool is needed. This becomes especially important once the size of a project grows and the number of dependencies on other jobs increases. For this implementation, BMC’s Control-M [35] tool is used, as it is the tool that all other jobs at the target company are orchestrated with.

Using dbt has helped simplify the scheduling process. This is because dbt can use the DAG to run the models in the correct order and do parallelization. This, in turn, means that fewer jobs need to be configured to run on the orchestration tool. For this implementation, it is enough to run one simple command: *dbt run -s +fact_table*. This plus sign in the command means that all models upstream from the fact table will be run, meaning that the dimensions and other intermediate models are run first and, lastly, the fact table. In the previous solution, all the stored procedures for updating dimensions and fact tables needed to be configured separately. This meant that finding the optimal configuration for the best parallelization was difficult.

The old solution was very dependent on the time at which the transformation was run due to the lacking implementation for freezing the data. This is because the data that was used was the data that was currently active in the database, which would change more and more the further from the reporting time the procedures were run. Meaning that delays would cause inconsistencies between reporting months, making the data unreliable. As the new solution has an improved implementation for the frozen data, the solution can be run at any time after the month has changed, and the result is the same, guaranteeing reliable data. Currently, the process for updating the data for this solution is configured to run after midnight on the first day of each month so that we can strive to have the data available as early as possible.

4.8 Continuous Integration and Continuous Delivery

Continuous integration and continuous delivery CI/CD are used to automate the process of integrating and deploying software [36]. Continuous integration CI is the process of building new changes and testing them using automated tests. This process is important when multiple developers are working on the same software to guarantee that the new changes do not cause issues with other changes. If problems are discovered during the continuous integration stage, it is easy for the developer to fix them as they receive feedback at an early stage. If no errors are discovered, the process can continue to the next stage, which is continuous delivery and continuous deployment. Continuous delivery is the process of automatically merging the changes from the continuous integration into a repository which should always be ready for deployment. If just continuous delivery is used, the deployment process is still done manually. Continuous deployment is used to automate this step as well.

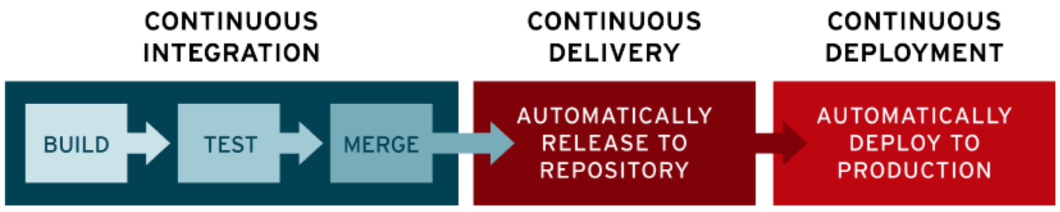

Figure 15 Overview of CI/CD process [36]

CI/CD is one area where the use of dbt has greatly improved the development experience. In the previous solution, this whole process was done completely manually. Developers needed to try their new changes by manually creating test versions of the stored procedures as well as creating the needed tables on the database before finally running the code on the database. Testing was also something that developers needed to do manually, as no automated tests were defined. This increased the risk of new changes causing unwanted errors elsewhere that would be difficult to find. Once the new changes were tested and accepted, the deployment process was also manual work by the developers. The new changes had to be added to the stored procedure, and then potential changes to table definitions were made. Lastly, the procedure was run to verify that it worked. Again, there was no automated testing to help guarantee that the deployment did not cause any unexpected issues. Git was used to track the changes to procedures and tables, but there was no guarantee that the code that was in Git was actually in sync with the code deployed on the database.

As a result of using dbt, it has become possible for us to use Microsoft's CI/CD pipelines [37] to automate the whole deployment process. We now have one CI pipeline that is run on the feature branch and one pipeline for CD. The CI pipeline is run by the developer when the development of a new feature is ready. First, the quality of the code is checked using a linter. Next, the pipeline detects changed models by comparing the new branch with the situation in production. Changed models are then built in a test environment, downstream models from the changed models are also built. Lastly, the tests defined for the models are run. If all these checks succeed, the developer can open a pull request to let other team members review the changes. Once the pull request is approved, the CD pipeline can be started. First, the new code is merged to the main branch, after which changed models are again detected and built, this time in the production environment.

Finally, the tests defined for the models are run again, now in the production environment.

This approach guarantees that the code that is in the version control system is the same as what is executed on the database. Automated testing also means that potential problems with the integration or deployment are detected at an early stage. As the whole deployment process is automated and extensive testing is done during CI, the risk of errors during deployment is also reduced significantly.

4.9 Cloud Migration

Once the dbt code was written and tested on the on-premises platform and the source data had been replicated to Databricks, the cloud migration process could be started. When using Databricks, a new dbt project was created, which uses the Databricks dbt adapter instead of the Teradata adapter to ensure that the correct DDL syntax is used for the new environment.

Data replicated from the on-premises platform to the cloud is not directly available in Databricks but stored in the cloud provider's blob storage [38]. This means that before it can be used in Databricks, the blob storage needs to be connected to Databricks, and the replicated tables need to be defined as external tables. Defining an external table is simple; all that is needed is to give the path to the table and specify the *catalog*, *schema*, and *table* that will be used for that table in Databricks. Once the external tables have been defined, they can be used in dbt as any other table.

```sql
CREATE TABLE <catalog>.<schema>.<table_name>
(
  <column_specification>
)
LOCATION 's3://<bucket_path>/<table_directory>';
```

Figure 16 Example of an external table in Databricks [39]

During the migration process, it became necessary to rewrite Teradata-specific functions that are not available in Databricks using ANSI syntax. Some examples of functions that needed to be rewritten are *NORMALIZE* and *P_INTERSECT*, both of which are used when creating the frozen data. To help with reusability and further development, the SQL code needed to replicate these functions was defined as macros which can then be reused whenever one of the functions are needed in the Databricks dbt project.

There are also slight differences in auxiliary structures, such as indexes and partitions, when using Databricks. Traditional indexes are not used. Instead, Databricks offers Z-ordering, which aims to locate related information in the same file, which can then be used by data-skipping algorithms [40]. Databricks also automatically collects file-level statistics on the 32 first columns for each table, which can be used to reduce the number of files that need to be scanned. The Z-ordering columns and partitions could be optimized to improve performance, but finding the best combinations is not necessarily straightforward. However, already by using the default settings, the performance was competitive when compared to the on-premises solution.

5. Evaluation of Results

When comparing the new solution with the previous one, three areas have changed. The processing rules have been updated to make the data more reliable and easier to analyse. Additionally, the tool for the transformation was changed from stored procedures to dbt. Lastly, the platform for the computations was moved from on-premises Teradata to Databricks in the cloud. These changes have many benefits but also a few challenges, which will be discussed in this chapter.

Two changes were made to the processing rules of the solution. Firstly, two new measures for counting the number of products were added to the fact table. These fields have common definitions across the various source systems from the underlying data. This solution has worked well during the testing phase and helped analysts to compare different source systems and countries better. It is worth noting that the addition of the new measures is not directly related to the cloud migration process or the updated frozen data implementation. These fields were just added simultaneously with the other changes to make the fact table more useful for analysts. The second change that was made to the business logic was the frozen data implementation. By using the bitemporal nature of the source data, the new solution can build a star schema that can be replicated multiple times and produce the same result every time, independently of when the processing of the data was done. This makes the data more reliable as users do not need to consider potential differences that may be caused depending on when data was loaded. Additionally, this means that corrections and additions, like new fields, can be made to the solution without affecting already loaded information.

The bitemporally frozen solution also caused some new challenges relating to performance and added complexity. Creating the bitemporally frozen data means that more processing needs to be done compared to the previous solution. This has caused some performance issues, especially on the on-premises platform. The computing of the frozen data also means that more intermediate tables and additional logic is needed, which makes the overall solution slightly more complex. However, both the added complexity and increased computation requirements are needed to create a solution with reliable data and, therefore, these drawbacks cannot be avoided. Some of these concerns are partially mitigated using dbt for clearer code, making the added steps needed for the frozen data intuitive and easy for developers to understand. Databricks can be used to help alleviate performance issues caused by the more complex transformation.

The new solution uses dbt for the transformation logic in favour of the stored procedures used in the previous implementation. Using dbt has proven to be beneficial in many ways. Firstly, the improved CI/CD process removes much of the manual work that developers would have needed to do previously, both when setting up a development environment and when deploying changes. The main reason for this is that dbt automates the DDL, which means that there is no longer a need to manually update table definitions in production, or manually create tables in the test environment. Additionally, due to the good Git integration and using deployment pipelines, the code in Git is guaranteed to be the same as the code that is scheduled to run on the database. This was not necessarily the case with the stored procedure-based solution, as deployments and synchronization to Git were done manually.

Working with dbt has also made cooperation easier due to the before mentioned good Git integration, making coordinating the work of multiple developers easier. Automated testing is another factor that has made cooperation easier as the tests can be used to control that the changes a developer has made do not affect any unexpected parts of the code. In addition, testing has helped catch and resolve errors in the source data and the transformations at an early stage before production deployments, usually when running the CI pipeline. Several errors have been caught using this approach that, if the stored procedure-based approach was used, may have gone unnoticed for a long time. Therefore, it is apparent that automated testing is important and a big improvement over the previous solution.

As a possible further use case for dbt, it would be worth investigating using dbt for creating unit tests. This would allow designing more specific test cases, and checking edge cases would become more efficient. This can be done by changing the source data of dbt models when unit tests are run. When running unit tests, mock source data containing the test case would be used. Otherwise, the normal source data would be used. The output of the models using the mock data would then be compared with the expected output. Using unit tests would mean that developers can receive quick feedback on their work, based on the unit tests, instead of running the models using all the data, leading to increased development speed.

The last improvement provided by using dbt is Jinja templating. Using Jinja reduces code duplication, as macros can be defined for commonly used code. Jinja also helps make the code more concise and easier to read by using features such as the previously mentioned macros, if statements, and for loops. These features are familiar from other programming languages and, thus, something many developers are already familiar with. The use of Jinja also ensures that a DAG can be built of all the dependencies between the models in the dbt project. This, together with the documentation features of dbt, provides a good overview of the whole process, making it easier for both the end users and new developers to understand.

Converting the code to dbt has also led to a couple of challenges that are mostly related to performance issues, especially when using the on-premises Teradata platform for the transformation. The main problem has been the performance of long common table expression CTE code blocks. The dbt code is commonly written in a CTE format, as it is more readable compared to using subqueries. These CTEs can sometimes become quite long, which has, in turn, led to Teradata having trouble optimizing the queries in a performant way. When using stored procedures, this issue was solved by materializing intermediate results in volatile tables [41], which are only kept for the duration of a session. Indexes and partitions can then be used on these volatile tables to store intermediate results in a way that makes further computations efficient, in other words helping the optimizer find the best plan. Volatile tables cannot be used in dbt, as one model consists of only one select statement. However, splitting the models into multiple intermediate models is possible and achieves the same result. This approach has the drawback that the DAG becomes unnecessarily cluttered, and as the materialization must be done using physical tables, it also leads to a significant amount of redundant storage of intermediate results taking up disc space. In some cases, the performance issues could also be solved by tweaking and optimizing the queries. However, this is quite time-consuming and a clear drawback of using dbt when compared to the previous solution.

Moving the processing from the on-premises platform to Databricks had several benefits. Initial tests show that the processing time on Databricks is faster when compared to the on-premises platform. The execution time of a full refresh on the on-premises platform takes around 5 hours and requires the overall load to be low when the transformation is run. This has made testing and development difficult. All the needed models have not yet been converted to Databricks, meaning that a complete comparison is not possible. However, the execution times of a few dbt models have been compared, and we can see that the Databricks platform was significantly faster. *Table 14* shows the differences in runtimes between the platforms.

Model	**On-premises execution time**	**Cloud execution time**
A small dimension	4m 5s	1m 15s
A larger intermediate model	10m 59s	6m 6s

Table 14 Comparison of execution times between on-premises and cloud platforms.

Another important feature provided by using a cloud-based solution is scalability. During the testing phase, a medium-sized Databricks SQL warehouse [42] was used. However, upgrading to a larger or smaller cluster, depending on the needs of the workload, is relatively simple. This also means that we can avoid situations where one needs to wait for an optimal time when the system load is low to be able to run a heavy job successfully.

Databricks also offers a modern, intuitive, and graphical query profile that can be used to help identify what causes certain queries to perform poorly and improve them. Some examples of optimizations that are easy to find using this method are identifying when filters could be applied at an earlier stage to reduce the number of rows processed at later stages.

The use of Databricks has also provided some challenges. One new challenge is moving data between on-premises and the cloud, which adds complexity and can be time and resource-heavy. The migration process also requires some work with rewriting the dbt code in a way that works on the new platform. This process was made significantly easier by using dbt instead of migrating stored procedures, as dbt adapters take care of many differences between the two platforms.

6. Conclusion

This thesis describes the modernization process of an ETL pipeline. The benefits and drawbacks of a cloud-based ETL process that uses Databricks as the computation platform and dbt as the transformation tool are compared with the previous on-premises platform that used stored procedures for the transformation logic. Simultaneously with updating the used tools and platform, improvements were made to the business logic of the ETL process. Most importantly, the method for freezing data was changed from a snapshot-based method that stored the active data, as it was recorded in the database at the time of loading, to a more dynamic method that uses the bitemporal source tables. This means that the process is idempotent and can be run multiple times and produce the same result every time.

The new solution allows for more flexibility and scalability because of the cloud-based platform. Simultaneously, the load on the on-premises platform is reduced. The new transformation tool has helped in creating reusable code that can be ported between execution environments with relatively low effort. Using dbt has also improved and helped automate the testing and deployment, which translates to faster and easier development as well as higher data quality.

Moving the processing to the cloud also means the first steps towards a data lakehouse architecture, where all data is made available in one central place, and the same platform can be used for reporting, business intelligence, data science, and machine learning. This would remove the need to maintain two separate platforms and thus simplify the overall architecture. However, before a data lakehouse architecture can be used, there is still much work to be done. Data currently available only on the on-premises platform needs to be made available in the cloud as well so that users can switch to the cloud platform.

6.1 Further Work

The development of the new implementation is still not complete. Further development that is currently planned is to move the processing for the fact table to the cloud so that the whole solution can be run on the same platform. Once that is done and the data replication is ready for production use, the processing for the production runs of the whole solution should be moved to the cloud, where we have seen better performance. Moving to the cloud will also reduce the load on the on-premises system, which will mean more computing resources are freed up for other work.

There are also plans to connect more dimensions to the fact table to have more information available for analysis. This means that new dimensions need to be converted from stored procedures to dbt, and the frozen logic needs to be implemented to create dimensions that are resistant to changes in the source data. For this work, the knowledge gained during the initial work outlined in this thesis will be valuable and help make the new development a simpler process.

Svensk sammanfattning

Inledning

Data spelar en stor roll i beslutsprocessen för dagens företag. Tillgång till data av hög kvalitet möjliggör att beslutsfattare i företagen kan fatta datadrivna beslut. För att åstadkomma detta måste data först integreras. Data integrering har därför blivit en alltmer viktig process för företagens IT-avdelningar att sköta. Dataintegration innebär att transformera data som finns utspridda mellan flera olika källsystem till ett och samma homogena datalager. I denna avhandling kommer en sådan process hos ett företag att beskrivas och moderniseras med hjälp av nya verktyg. Företaget, vars dataintegrationsprocess studeras, är ett stort företag inom finansbranschen med verksamhet i hela Norden. Detta betyder att många olika källsystem används och beroende på land kan strukturen på dessa vara väldigt olika. Därför är det viktigt att data i det slutliga datalagret använder sig av samma struktur så att jämförelser mellan länder och system blir så enkelt som möjligt, vilket gör att slutanvändarna enkelt kan analysera data samt fatta väl informerade beslut på basis av data.

Målet med denna avhandling är att modernisera en dataintegrationsprocess på följande sätt:

- Skapa en ny lösning som använder sig av moderna verktyg
- Effektivera processen så att data blir redo för användare i ett tidigare skede.
- För att få en lösning med bättre skalbarhet och minska belastningen på den lokala databasplattformen ska bearbetningen av data flyttas till molnet.
- Underlätta samarbete mellan utvecklare genom att skapa en lösning som är välintegrerad med Git-versionshanteringsprogrammet.
- Använda en automatiserad CI/CD (continious integration / continious deployment) process för test och produktionssättnigsprocesserna.
- Förbättra datakvalitén genom att inkorporera automatiserade test i den nya lösningen.
- Den nya lösningen borde också vara ”frusen” så att samma resultat kan åstadkommas oberoende av när transformationen körs. Detta är viktigt för att få pålitliga data som är jämförbara oberoende av när själva laddningsprocessen utfördes.

Datalager

Ett företag har ofta två typer av databassystem, ett för online transaktionsbearbetning OLTP (Online Transaction Processing) och ett för online analytisk bearbetning OLAP (Online Analytical Processing). OLTP-system är oftast realtidssystem som används för företagets dagliga operativa verksamhet. OLAP-system används för analys av data och innehåller ofta även historiska data för att kunna analysera trender. I denna avhandling studeras en process för dataintegrations i ett datalager som alltså är ett OLAP system.

För att data ska vara lätt att använda är det viktigt att modellera data på rätt sätt. Data i ett datalager kan modelleras på olika sätt beroende på vad målet är. Ralph Kimball föreslår användning av stjärnschema [2]. Målet med en stjärnschema-arkitektur är att presentera data i ett format som är enkelt för slutanvändare att använda. Stjärnschemat består av två typer av tabeller, faktatabeller och dimensionstabeller. Flera dimensionstabeller kan kopplas ihop med en faktatabell vilket gör att modellen påminner om en stjärna. Kimball argumenterar också för en processdriven modellering, vilket betyder att ett stjärnschema representerar en av företagets processer. Faktatabellen innehåller mått, alltså värden som ändras ofta och beskriver ofta en transaktion. Faktatabellen ska vara på den lägsta möjliga detaljnivån; eventuella aggregationer kan göras senare av användaren. Dimensionstabellerna innehåller övrig information som inte är på en lika låg nivå och ändras därför mer sällan. Faktatabellen innehåller även nycklar som möjliggör att enkelt koppla ihop faktatabellen med de dimensionstabeller man är intresserad av.

Bill Inmon har en annan synvinkel på hur modellering av data i ett datalager borde göras [5]. Han förespråkar en så kallad EDW (Enterprise Data Warehouse) baserad lösning. Idén är att ha en och samma datamodell för hela företaget som en grund; sedan kan olika avdelningar enklare bygga sina egna lösningar som alla baserar sig på en och samma EDW. Till skillnad från den processdrivna modelleringen från stjärnschemat är tanken bakom EDW att vara dataorienterad. Målet är alltså att skapa en modell som baserar sig på strukturen av underliggande data. En annan skillnad från stjärnschemat är att data i EDW är normaliserat vilket medför att uppdaterande av data är lättare och hela lösningen kräver mindre lagringsutrymme. Eftersom en normaliserad struktur är mera komplicerad blir hopkoppling av data svårare. Denna typ av typ av modellering är därför inte lika användarvänlig som stjärnschemat; större fokus läggs i stället på prestanda.

För att populera data till ett stjärnschema eller EDW måste data först läsas från en eller flera källor och sedan transformeras för att få rätt struktur. Slutligen laddas det transformerade data in datalagrets tabeller. Denna process kallas i sin helhet extrahering, transformering och inläsning ETL (Extract, Transform, Load). Processen kan ofta vara rätt invecklad eftersom källdata ofta är bristfälligt. Dessutom har särskilt större företag ofta flera olika system som alla producerar källdata i varierande format vilket gör att det krävs en hel del transformationslogik för att få data i ett och samma format i de slutliga tabellerna i datalagret.

Implementation

I denna avhandling moderniseras en ETL-process som laddar ett stjärnschema som innehåller en faktatabell samt fyra dimensionstabeller. Som källdata för processen används data från ett EDW. Användningen av EDW som datakälla gör processen aningen enklare eftersom en del av stegen som behövs för dataintegration redan gjorts då data laddats in i EDW. Data i stjärnschemat uppdateras en gång per månad så att alla rader som är aktiva i slutet av månaden läggs till tabellen. Den tidigare lösningen var gjord med hjälp av lagrade procedurer (stored procedure) och kördes på en lokal Teradata-databas.

Från affärsverksamhetens sida var det största problemet med den gamla lösningen att data laddades en gång per månad, så som det existerade i databasen vid laddningstidpunkten. Om samma data laddas på nytt vid en senare tidpunkt skulle resultatet inte längre bli samma som följd av försenade uppdateringar i källsystemens data. För att lösa detta problem ändrades ETL-processen så att man i stället för att alltid läsa de rader som för tillfället är aktiva använder den bitemporära naturen av underliggande källdata. Eftersom källdata är bitemporära betyder det att inga data raderas från databasen utan när ändringar kommer in i databasen uppdateras de tidigare aktiva raderna till att inte längre vara aktiva. Detta gör det möjligt att återskapa samma data som fanns i databasen vid vilken som helst tidpunkt i historien. Med hjälp av den nya metoden blir det alltså möjligt att reproducera samma resultat oberoende av den verkliga laddningstidpunkten. Detta är viktigt för att kunna garantera jämförbara data mellan rapporteringsmånader och för att kunna korrigera fel i data utan att samtidigt införa oönskade ändringar, som följd av den nya laddningstidpunkten.

Samtidigt med förbättringar i affärslogiken moderniserades hela processen genom att byta verktyg för transformationen samt exekveringsplattform. Det nya verktyget, som användes för transformationen är dbt som är ett öppen källkods verktyg för datatransformation, underhållet av dbt Labs. Dbt gör det möjligt att skriva SQL-kod och samtidigt använda sig av Jinja-syntax för att skapa makron, if-satser och for-loopar. Koden som skrivs i dbt kompileras till vanlig SQL kod som sedan exekveras mot en databas. Användning av dbt medför att utvecklare inte själva behöver skriva datadefinitionsspråk DDL (data definition language) uttryck för att skapa tabeller och vyer i databasen, utan de är något som dbt tar hand om. Detta betyder att utvecklaren endast skriver select-uttryck, dbt använder sig sedan av adapters som skapar behövlig DDL beroende på databasen som används. Målet med användning av dbt är att skapa lättare underhållbar kod som samtidigt blir mera portabel vilket gör byte av exekveringsplattform enklare.

Efter att lagrade procedurerna konverteras till dbt-kod och den uppdaterade logiken för fryst data implementeras, testades koden först på lokala plattformen. När koden fungerade på ett nöjaktigt sätt blev det aktuellt att försöka flytta exekveringen till molnet. För detta användes Databricks SQL, som är en molnbaserad plattform för datalager. För att åstadkomma molnmigrationen kan nästan samma dbt-kod, som skapats för den lokala lösningen, användas. Det enda som krävde ändring var vissa Teradata-specifika funktioner som inte är tillgängliga i Databricks och måste därför omskrivas med hjälp av dbt makros. Det mest komplicerade med att flytta transformationen till molnet visade sig vara att överföra källdata från lokala plattformen till molnet och sedan överföra transformerade data tillbaka från molnet till lokala plattformen. Datamängderna som måste överföras är rätt stora vilket gör att överföringsprocessen måste vara effektiv. Lyckligtvis kunde vi använda oss av ett in-house utvecklat program för att ta hand om den delen vilket medförde att själva molnmigrationen blev rätt enkel.

Evaluering och avslutning

Denna avhandling har beskrivit moderniseringen av en datatransformationsprocess. Denna process använde ursprungligen lagrade procedurer och kördes på en lokal databas. Efter moderniseringen användes transformationsverktyget dbt och molnbaserade Databricks-plattformen. Samtidigt gjordes också andra ändringar i transformationslogiken för att åstadkomma pålitlig och högkvalitets data.

Den nya implementationen har visat att användningen av dbt-verktyget medför stora fördelar när det gäller produktivitet och garanterande av datakvalitet. Dbt visade sig också vara ett bra verktyg när flera utvecklare arbetar med samma projekt, eftersom dbt naturligt integreras bra med Git. Dessutom förenklar dbt skapande av utvecklings och testmiljöer som följd av automatiserad DDL vilket innebär att tabeller enkelt kan skapas i olika miljöer utan manuellt arbete. Processen för produktionssättning har också blivit enklare och med mindre risk för fel. Ändrande av metoden för att frysa data har också visat sig vara fungerande och möjliggjort mera flexibilitet för vidare utveckling av lösningen och fortfarande kunna producera samma resultat oberoende på när laddningsprocessen körs.

Genom att flytta processeringen av lösningen till molnbaserade Databricks minskade belastningen på lokala plattformen och processeringstiden minskade också betydligt. Samtidigt möjliggör en molnbaserad tjänst också bättre skalbarhet för eventuella framtida behov. Den största utmaningen med en molnbaserad lösning var att källdata först måste överföras till molnet och efter att data transformerats i molnet behöver data ännu överföras tillbaka till lokala plattformen, vilket orsakar en del onödig komplexitet. Förhoppningsvis kommer denna process att kunna förenklas i framtiden då källsystem erbjuder allt mera data direkt i molnet och slutanvändare också har möjligheten att använda data direkt från molnet.

References

[1] N. Mali and S. Bojewar, “A Survey of ETL Tools,” *International Journal of Computer Techniques,* vol. 2, no. 5, pp. 20 - 27, 2015.

[2] M. Ross and K. Ralph, The Data Warehouse Toolkit : The Definitive Guide to Dimensional Modeling, John Wiley & Sons, Incorporated, 2013.

[3] A. Silberschatz, H. F. Korth and S. Sudarshan, Database system concepts, McGraw-Hill US Higher Ed ISE, 2019.

[4] P. PONNIAH, “DATA WAREHOUSING FUNDAMENTALS A Comprehensive Guide for IT Professionals,” John Wiley & Sons, 2004.

[5] M. Breslin, “Data warehousing battle of the giants: Comparing the Basics of the Kimball and Inmon Models,” *Business intelligence journal,* vol. 7, pp. 6 - 20, 2004.

[6] Fivetran - Michael Kaminsky, “Data warehouse modeling: Star schema vs. OBT,” [Online]. Available: https://www.fivetran.com/blog/star-schema-vs-obt. [Accessed 30 April 2023].

[7] P. Ziegler and K. R. Dittrich, “Data Integration – Problems, Approaches, and Perspectives,” in *Conceptual modelling in information systems engineering*, Springer, 2007, pp. 39 - 58.

[8] IBM, “ELT (Extract, Load, Transform),” IBM, [Online]. Available: https://www.ibm.com/topics/elt. [Accessed 30 December 2022].

[9] T. Johnston, Bitemporal data: theory and practice, Newnes, 2014.

[10] M. Armbrust, A. Ghodsi, R. Xin and M. Zaharia, “Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics,” in *Proceedings of CIDR*, 2021.

[11] Apache, “ORC Specification v1,” [Online]. Available: https://orc.apache.org/specification/ORCv1/. [Accessed 30 April 2023].

[12] Apache, “Apache Avro™ 1.11.1 Documentation,” [Online]. Available: https://avro.apache.org/docs/1.11.1/. [Accessed 30 April 2023].

[13] B. Inmon, M. Levins and R. Srivastava, Building the Data Lakehouse, Technics Publications, 2021.

[14] Apache, “Encodings,” [Online]. Available: https://parquet.apache.org/docs/file-format/data-pages/encodings/. [Accessed 29 January 2023].

[15] Databricks, “What is Parquet?,” [Online]. Available: https://www.databricks.com/glossary/what-is-parquet. [Accessed 29 January 2023].

[16] B. Haelen, Delta Lake: Up and Running Modern Data Lakehouse Architectures with Delta Lake, O’Reilly Media, Inc, 2022.

[17] D. Lee, T. Das and V. Jaiswal, Delta Lake: The Definitive Guide, O’Reilly Media, 2021.

[18] dbt Labs, “What is dbt?,” [Online]. Available: https://docs.getdbt.com/docs/introduction. [Accessed 4 February 2023].

[19] dbt Labs, “Supported data platforms,” [Online]. Available: https://docs.getdbt.com/docs/supported-data-platforms. [Accessed 4 February 2023].

[20] Jinja documentation, “Introduction,” [Online]. Available: https://jinja.palletsprojects.com/en/3.1.x/intro/. [Accessed 6 February 2023].

[21] YAML Language Development Team, “YAML Ain’t Markup Language (YAML™) version 1.2,” [Online]. Available: https://yaml.org/spec/1.2.2/. [Accessed 17 February 2023].

[22] dbt Labs, “Materializations,” [Online]. Available: https://docs.getdbt.com/docs/build/materializations. [Accessed 4 February 2023].

[23] dbt Labs, “Jinja and macros,” [Online]. Available: https://docs.getdbt.com/docs/build/jinja-macros. [Accessed 4 February 2023].

[24] dbt Labs, “Tests,” [Online]. Available: https://docs.getdbt.com/reference/resource-properties/tests. [Accessed 6 February 2023].

[25] dbt Labs, “Refactoring legacy SQL to dbt,” [Online]. Available: https://docs.getdbt.com/docs/get-started/learning-more/refactoring-legacy-sql. [Accessed 11 February 2023].

[26] Databricks, “What is Databricks?,” [Online]. Available: https://docs.databricks.com/introduction/index.html. [Accessed 18 February 2023].

[27] Databricks Inc., “Databricks Cloud Partners,” [Online]. Available: https://www.databricks.com/company/partners/cloud-partners. [Accessed 12 February 2023].

[28] Microsoft, “What is Azure Databricks?,” [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/introduction/. [Accessed 12 February 2023].

[29] Microsoft, “Query profile,” [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/sql/admin/query-profile. [Accessed 12 February 2023].

[30] Microsoft, “What is Unity Catalog?,” [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/data-governance/unity-catalog/. [Accessed 12 February 2023].

[31] Microsoft, “Configure SQL warehouses,” [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/sql/admin/sql-endpoints. [Accessed 12 February 2023].

[32] Microsoft, “Azure Data Factory,” [Online]. Available: https://azure.microsoft.com/en-us/products/data-factory#resources. [Accessed 27 February 2023].

[33] dbt Labs, “Incremental models,” [Online]. Available: https://docs.getdbt.com/docs/build/incremental-models. [Accessed 23 February 2023].

[34] dbt Labs, “Project variables,” [Online]. Available: https://docs.getdbt.com/docs/build/project-variables. [Accessed 27 February 2023].

[35] BMC, “Control-M,” [Online]. Available: https://www.bmc.com/it-solutions/control-m.html#. [Accessed 9 March 2023].

[36] Red Hat, “What is CI/CD?,” [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd. [Accessed 9 March 2023].

[37] Microsoft, “What is Azure Pipelines?,” [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops. [Accessed 9 March 2023].

[38] Microsoft, “Introduction to Azure Blob Storage,” [Online]. Available: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction. [Accessed 29 April 2023].

[39] Databricks, “Create Tables,” [Online]. Available: https://docs.databricks.com/data-governance/unity-catalog/create-tables.html. [Accessed 03 April 2023].

[40] Databrics Inc., “Data skipping with Z-order indexes for Delta Lake,” [Online]. Available: https://docs.databricks.com/delta/data-skipping.html. [Accessed 13 April 2023].

[41] Teradata, “Volatile Tables,” [Online]. Available: https://docs.teradata.com/r/rgAb27O_xRmMVc_aQq2VGw/mpJF1z_vSlpMbZYxFmRJfA. [Accessed 20 March 2023].

[42] Microsoft, “Configure SQL warehouses - Cluster size,” [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/sql/admin/create-sql-warehouse#cluster-size. [Accessed 16 April 2023].