Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

# Consistency and Fault-Tolerance in Data Warehouses

Bachelor Thesis

2022
Radka Ďurčová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

# Consistency and Fault-Tolerance in Data Warehouses

Bachelor Thesis

| | |
|---|---|
| Study Programme: | Computer Science |
| Field of Study: | Computer Science |
| Department: | Department of Computer Science |
| Supervisor: | Mgr. András Varga, PhD. |

Bratislava, 2022
Radka Ďurčová


Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Radka Ďurčová
**Študijný program:** informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
**Študijný odbor:** informatika
**Typ záverečnej práce:** bakalárska
**Jazyk záverečnej práce:** anglický
**Sekundárny jazyk:** slovenský

**Názov:** Consistency and Fault-Tolerance in Data Warehouses
*Konzistentné dátové sklady a ich odolnosť voči chybám*

**Anotácia:** V komerčnom svete sa dátové sklady využívajú na podporu rozhodovania, keďže ponúkajú užitočné - a často agregované – dáta z minulosti. V dnešnej dobe s rýchlo rastúcim objemom informácii dátové sklady sú často založené na aplikáciach pre spracovanie veľkých dát, napr. Spark alebo Hive. Táto zmena umožňuje budovanie dátových skladov na tzv. cloude systéme, pričom konzistencia dát a odolnosť voči rôznym haváriám, pádom ostávajú naďalej najdôležitejšími požiadavkami.

**Cieľ:** Cieľom tejto práce je vytvoriť prehľad možných dizajnov dátových skladov, ktoré udržiavajú dáta v konzistentnom stave. Inkonzistencia dát môže vzniknúť rôznymi spôsobmi, napr. pri chybných transformáciách alebo pri páde ETL procesov. Dizajn dátových skladov by mal umožniť obnovenie procesov alebo zariadiť, aby pády nemali vplyv na konzistenciu dát. Cieľom tejto práce je prezentovať aspoň jeden dizajn dátových skladov a implementovať ho pomocou aplikácií pre veľké dáta na vhodnom príklade.

**Kľúčové slová:** Analytické databázy, Reportovanie, Dátový sklad, Veľké dáta, Spark, Hive, Odolnosť voči chybám, Konzistencia dát

**Vedúci:** Mgr. András Varga, PhD.
**Katedra:** FMFI.KI - Katedra informatiky
**Vedúci katedry:** prof. RNDr. Martin Škoviera, PhD.

**Dátum zadania:** 17.10.2021

**Dátum schválenia:** 04.11.2021 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce


Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

# THESIS ASSIGNMENT

| | |
|---|---|
| **Name and Surname:** | Radka Ďurčová |
| **Study programme:** | Computer Science (Single degree study, bachelor I. deg., full time form) |
| **Field of Study:** | Computer Science |
| **Type of Thesis:** | Bachelor´s thesis |
| **Language of Thesis:** | English |
| **Secondary language:** | Slovak |

**Title:** Consistency and Fault-Tolerance in Data Warehouses

**Annotation:** In the domain of business data warehouses are widely used to support business decision-making by providing useful, often aggregated, information about the past. With the amount of processed information rising sharply data warehouses often migrate to a big data toolkit, e.g. Spark, Hive. This shift enables data warehouses to be hosted in cloud systems, which, on the other hand, rises the concern of fault-tolerance and data consistency.

**Aim:** The aim of this thesis is to provide an overview of possible design solutions to ensure data consistency in the data warehouse utilizing the big data toolkit. Data inconsistency may occur for several reasons, e.g. incorrect business transformations or runtime failures of ETL jobs. The design of the proposed data warehouses should be able to recover gracefully or fail without corrupting data. The goal of this thesis is to propose at least one data warehouse design and implement it on a simple data warehouse schema.

**Keywords:** Analytic Database, Reporting, Data Warehouse, Big Data, Spark, Hive, Fault-Tolerance, Data Consistency, Job Orchestration

| | |
|---|---|
| **Supervisor:** | Mgr. András Varga, PhD. |
| **Department:** | FMFI.KI - Department of Computer Science |
| **Head of department:** | prof. RNDr. Martin Škoviera, PhD. |
| **Assigned:** | 17.10.2021 |

**Approved:** 04.11.2021 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

**Acknowledgments:** I would like to thank my supervisor Mgr. András Varga, PhD. for his constant support and guidance.

I would also like to thank IBM Slovakia for providing the servers used for the data warehouse implementation.

# Abstrakt

Distribuované dátové sklady hrajú dôležitú rolu v spracovaní a analýze veľkých dát. Ich zvýšená citlivosť voči pádom však môže mať značný vplyv na konzistenciu a dostupnosť dát, čo sú jedny z hlavných požiadaviek ich klientov. V tejto bakalárskej práci sa zaoberáme možnými dizajnami dátových skladov, ktoré sú odolné voči pádov a technikami obnovenia, ktoré sa v rámci nich využívajú. Pre účely porovnania rôznych prístupov v prostredí veľkých dát sme implementovali distribuovaný dátový sklad s použitím nástrojov pre veľké dáta ako sú Hive a Spark. Experimenty boli zamerané na vyhodnotenie efektivity optimalizovanej Dependency Analysis metódy v porovnaní s naivným prístupom. Výsledky nášho porovanania ukazujú, že Dependency Analysis značne skracuje proces obnovy a predstavuje efektívny spôsob na zachovanie konzistencie dát aj v prostredí distribuovaných dátových skladov.

**Kľúčové slová:** analytické databázy, dátové sklady, veľké dáta, odolnosť voči chybám, konzistencia dát

# Abstract

Distributed data warehouses play an essential role in big data processing and analytics. However, their increased exposure to failures has a detrimental effect on the business requirement for consistent and available data. In this thesis, we consider various designs of fault-tolerant data warehouse systems and their resumption strategies. We implement a distributed data warehouse using big data tools like Hive and Spark to test the efficiency of the error-handling methods in a big data environment. In particular, we evaluate the performance of the optimized Dependency Analysis method compared to the naive resumption approach. The results of our testing suggest that the Dependency Analysis can improve the performance considerably and proves to be efficient even in a distributed setting.

**Keywords:** analytic database, data warehouse, big data, fault-tolerance, data consistency

# Contents

# List of Figures

# Introduction

Data management and analytics have become standard practices in any modern enterprise. They are essential for making confident, data-driven business decisions. As a result, an immense amount of data is generated, collected, and subsequently analyzed every day. Data warehousing and business intelligence (DW/BI) systems provide an infrastructure to store and process the extensive consequent data sets. Over the years, the amount of data generated and stored in certain systems has become too vast to be maintained using conventional methods, which has led to the creation of a new paradigm in data processing, the so-called *big data*.

In the area of big data, it is anticipated that the input data cannot be computed or transformed in a timely manner using conventional resources. This led to the development of several big data-specific tools, which operate on top of a cluster of computers, often referred to as *nodes*, to achieve acceptable speed. The data transformation is often executed in a distributed manner on several nodes, with the input data being split between these transformations. This approach enables data-driven computations on vast input data sets, but due to the nature of clusters, it introduces a new problem, namely *node failures*.

With the importance of data analytics in clients' decision-making process, the issue of data quality and consistency arises. If users cannot rely on the veracity of the data, they cannot make confident decisions based on them. Consequently, numerous strategies were presented throughout the years to ensure data consistency. However, the specifics of big data processing introduce new challenges in this area.

In this thesis, we compare the efficiency of the Dependency Analysis resumption method introduced in paper [17] to a non-optimized approach. We aim to review its findings in a big data environment with a distributed data warehouse. Our goal is to show that Dependency Analysis remains an efficient error-handling method in a distributed environment.

The basis of the data warehouse design is a fabricated use case in which the client is a fictitious music streaming platform. The goal is to provide the client with information about the music preferences of its users. This use case provides us with enough flexibility to design a minimalistic data warehouse schema that is complex enough to demonstrate the required behavior. In addition, we are able to generate a sufficient

amount of quality data for the use case.

In Chapter 1 we provide the basics of data warehousing and expound on the concept of consistency and fault-tolerance within the context of a data warehouse and big data. In Sections 1.4 - 1.6 we introduce the strategies for a fault-tolerant system and the related recovery methods.

In Chapter 2, we give an overview of the system architecture and the technologies employed in this thesis. In Section 2.1 we present the details of the cluster we use for our implementation. In Section 2.2 we provide a shortlist of tools we consider for the implementation of the distributed data warehouse, followed by the reasoning behind our particular choices.

In Chapter 3, we present the logical design of the data warehouse implemented. We first introduce the model of the source system in Section 3.1, followed by the logical design of the data warehouse in Section 3.2. Last, in Section 3.3 we define the extent of data transformation between the two designs.

In Chapter 4, we describe the details of the implementation process. First, the creation of test data sets is described in detail, followed by the provisioning of the data warehouse system on the existing cluster. In Sections 4.3 - 4.4 we present the implementation design of the data transformation with and without the Dependency Analysis method. Finally, in Section 4.5 we present the performance comparison results between the different methods.

# Chapter 1

# Background and Basic Concepts

In this chapter, we provide a short overview and main concepts related to data warehousing and big data. Next, we describe the state of the art in the area of consistent data warehousing based on two methods for error-recovery published in papers [6], [15] and [17].

## 1.1 Data Warehouse

A data warehouse is a type of analytic system designed to support business intelligence analytics. In article [3] it is defined as "a centralized repository of integrated data from one or more heterogeneous sources".

Book [1] describes several differences between operational and analytic database systems. The operational system handles individual transactions. Its primary function is to manage data; it is optimized to perform updates to the database. On the other hand, the analytic system focuses on aggregated transactions – to analyze a business process, we need to consider all related transactions. Historical data are relevant for future analysis and stored in a data warehouse, in contrast to operational systems, which focus mainly on current data.

Several DW/BI architectures were proposed throughout the years, although they all share essential characteristics. In this thesis, we implement the dimensional data warehouse architecture proposed by Ralph Kimball in book [7] (see Figure 1.1).

While building a data warehouse, data are usually collected from operational systems. These source systems are autonomous, and their structure rests solely on their own design, unrelated to the needs of the data warehouse. Due to this, a data warehouse often receives data differing in format or structure from its distinct source systems. An ETL process is responsible for effective and reliable data migration and integration from data source systems to a data warehouse. Input data need to be extracted from the source systems and validated. Subsequent transformations (e.g., joining, filtering,

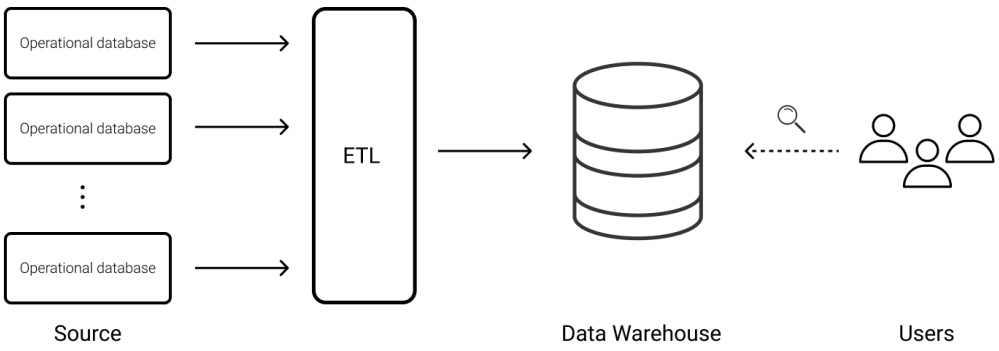


**Figure 1.1:** The dimensional data warehouse architecture proposed by Ralph Kimball

splitting or merging columns) ensure that the data conforms to the rules of a data warehouse database and are loaded in a unified format, utilizable for future analysis.

Source systems tend to be updated frequently, and these changes need to be migrated into a data warehouse. While it is possible to run transaction-based loading using streaming ETL, traditionally, an ETL process runs periodically using bulk loading. Based on the decision of the business, a data warehouse might become unavailable for clients while an ETL process is running to avoid using partially loaded data sets. For this reason, the execution of an ETL process is often limited by a specified time window. Both frequency and time window length are subject to the business logic.

Communication between a client and a data warehouse occurs through Data Query Language (DQL), meaning that the client can only fetch data, not change or manage them. Since the volume of data may be significant, data warehouses are optimized to perform data retrieval efficiently, even at the expense of write efficiency.

## 1.2 Dimensional modeling

Book [7] proposes *dimensional modeling*, a technique explicitly designed for analytic systems, addressing the need for both high performance and comprehensible data format. Each dimensional model is based on a particular business process the user is interested in. Consequently, the analytic system can evaluate related queries faster and provide all information the user requires.

In this thesis, we apply a dimensional model known as a star schema. This model consists of a central fact table and several dimension tables stored in relational database management systems.

A fact table stores measurements (numerical values) relevant for the business process evaluation. While it generally does not have many attributes, the number of rows grows rapidly. Clients might be interested in different aggregates or statistics of in-

dividual facts, so it is important that the facts are additive or at least semi-additive. Non-additive facts, such as percentages, are impractical for analysis considering large data volumes.

Dimension tables provide context for the fact table. Standalone facts have no informative value. We need to know the circumstances surrounding a measurement event in order to be able to analyze various aspects of a business process. We use dimension attributes to filter facts and define the aggregation level. It follows that facts are dependent on their related dimensions. They are linked together using primary/foreign key relationships. Each row in a dimension table is assigned a surrogate key, usually generated during an ETL process. The foreign keys are then stored in a fact table to relate measurements to their dimension attributes.

Unlike fact tables, dimension tables tend to have many attributes (dimensions), but the accumulation of rows is considerably slower and not necessarily periodic. Additionally, dimension tables may change if information in the source system changes. We use Slowly Changing Dimensions (SCD) to determine how to handle dimensions that are subject to change. The two most commonly used SCD types are type 1 and type 2. In a type 1 change, we overwrite the dimension losing any information about historical values. This type is usually used to correct errors in records. A type 2 change is connected to the requirement for maintaining historical data in a data warehouse. With a type 2 change, we insert a new record with current attributes into the dimension table. We retain the previous value by having several versions of the same entity, each having a distinct surrogate key but having the same natural key. It enables us to associate each fact with the correct version of the dimension table record.

## 1.3 Consistency and fault tolerance

Maintaining the data in a data warehouse consistent and high quality is one of the top business requirements. It is virtually impossible to avoid all inconsistencies with multiple independent source systems and large data sets. Nevertheless, this does not mean that the whole data warehouse is inconsistent and unusable. Assuming that a large volume of data is available, a small-scale data loss does not affect the overall data analysis. In general, we demand a reasonable level of consistency depending on business logic — the client (business) decides what the acceptable level is.

In Paper [10] the authors study inconsistencies in data warehouses. One of the reasons for inconsistency is data itself, e.g., different ways of recording the same value across company departments or multiple copies of the same data. The transform stage of an ETL process should be able to eliminate such inconsistencies efficiently.

This thesis focuses on another frequent reason for data inconsistency — failed ETL

jobs. A failure of an ETL job may lead to severe data corruption with an extensive impact on data analysis results. Repairing the affected data can be potentially very complicated, if not impossible. For example, there is no way to restore SCD attributes of type 1 without backup since the source systems generally do not store historical data. For this reason, while designing an ETL process, we should always expect failures and have a strategy for how to respond to them. Even though an ETL process implemented should be idempotent, simply rerunning it in cases of failure is not necessarily sufficient. If the failure occurs during a load stage, a data warehouse would contain a partially loaded data set. Rerunning an ETL job without prior damage control may result in duplicated data.

A critical aspect to consider when designing a fault-tolerant ETL process is the performance. Given that we expect ETL jobs to run within a certain time window, the objective of a recovery strategy is not only to maintain data consistency but also to be efficient and finish in a timely manner. If a failure occurs and there is not enough time to repeat the whole ETL process, it may be skipped altogether, resulting in outdated data in a data warehouse as discussed in paper [8]. Based on the client expectation, this behavior might be the preferred strategy.

## 1.4 Naive Approach

The straightforward response to a failure of an ETL job is fixing the error and executing the whole process again. In order to avoid corruption and duplication of data caused by partially loaded data sets, it is necessary to recover affected tables prior to that. The recovery process needs to identify the data that has already been loaded and purge them from the data warehouse. Afterward, we can rerun the ETL job.

As mentioned in paper [17], this strategy is highly inefficient since both the volume of data that has to be potentially revisited and the number of repeated operations may be significant. If the data scope is too big or the requested time window for the ETL process is too short, this approach is not convenient.

## 1.5 Checkpointing

Checkpoints are fixed points in a sequence of ETL steps in which we store the current data state. A checkpoint ensures that all the operations executed before this point in a workflow were executed correctly. For this reason, a checkpoint may serve as a recovery point. When a failure occurs, we use the last checkpoint saved before the failure to restore the consistent state of data. The follow-up recovery process can then use this state as a starting point, avoiding the need for a complete ETL job restart.

In order to fully utilize checkpoints, an ETL process should be divided into separate logical sections with well-defined functionality. Aside from the apparent benefits such as flexibility or facilitation of failure location, it also allows checkpointing to reach its full potential by enabling to skip or rerun specific parts of the code during the recovery process.

It is vital to assess the reasonable frequency. Creating checkpoints, as proposed in Paper [6], inevitably increases latency and reduces efficiency. Consequently, it is unattainable to place them after every step. It is common to leave the placement to the programmer's individual preference, for example, after a sizeable operation. However, ETL procedures are often complex, and they may have various additional objectives to fulfill. Paper [15] while studying the ETL workflow and its fault-tolerance claims that without a comprehensive strategy, finding an optimal design may be challenging.

Simitsis et al. (2010) in Paper [15] propose several different strategies for the efficient calculation of checkpoint placement considering several metrics. They propose to formulate the issue as an optimization problem that aims to maintain performance and minimize the recovery time.

Gorawski et al. (2007) describe the Checkpoint-based resumption algorithm that combines checkpointing with the Design-Resume (DR) algorithm. They aim to reduce the overhead caused by checkpointing in a regular ETL process. Both algorithms take the graph representation of an ETL workflow as its input, with the nodes being the ETL steps.

The Design-Resume (DR) algorithm was proposed by Labio et al. (2000) in paper [8]. The Design stage of the DR algorithm considers high-level properties of transform operations to filter out the rows that have been successfully loaded before a failure occurred. Consequently, only a subset of the original data set has to be processed again in the Resume stage. This approach significantly increases the efficiency of a recovery process without any additional operation needed during a regular ETL workflow.

As a result of the DR algorithm addition, only checkpoints in the transform nodes are required to be created regularly. In case of an error, we restore the state from the last saved checkpoint, making it the starting point o the recovery process. We then apply the DR algorithm to create new filter nodes, which are inserted into the workflow. Finally, we can resume the ETL job on the modified graph.

## 1.6 Dependency Analysis

Paper [17] from Shitao et al. proposes a fault-tolerance algorithm based on the analysis of dependency relationships between fact and dimension tables. It considers the sequence of ETL process steps to optimize the recovery process. As we mention in Section

1.2, facts are dependent on their dimensions. This relationship must be acknowledged in any ETL workflow by loading dimension tables prior to fact tables. Consequently, a failed dimension table processing tends to have a more prominent effect than a fact table one.

Before launching a regular ETL job, we need to determine the relationships between the tables in a data warehouse. From that, we then derive the previous steps each ETL action is contingent on and store the dependencies in a temporary table. Another temporary table is used for logging each step's execution result during an ETL run. We use these temporary tables to determine whether we execute or skip a step in an ETL workflow. We trace which preceding steps the current operation relies on and execute it only if all of the prerequisites were carried out successfully. By applying this procedure, we avoid potential data corruption.

The recovery workflow of an ETL process is launched in the event of an error. We consider the information collected in a regular run and repeat only those steps that failed or were skipped. In general, using this recovery technique reduces both the number of repeated operations and the volume of data that needs to be restored and thus increases the efficiency and performance of an ETL process.

## 1.7 The World of Big Data

The processing of large volumes of data places heavy demands on computation resources. Some of the challenges stemming from the vast data volumes are processing capacity or storage limitations. For this reason, cluster computing is utilized to perform the so-called *big data* computations. Big data tools are specifically designed to operate across clusters of computers and carry out distributed data processing.

The scope of big data operations amplifies the need for fault-tolerant approaches in big data systems. While employing multiple computers allows for sharing resources, it also adds new potential failure points. Moreover, tools for big data processing often run on a cluster with a master-slave architecture, including those used in this thesis. In a non-fault-tolerant setting, the master node represents a single point of failure — if the master node fails, the whole cluster and all of the ongoing computations are affected, which potentially leads to a significant amount of corrupted data.

# Chapter 2

# High-level Architecture and Design Considerations

In this chapter, we present the architecture of the system and the underlying toolkit facilitating the data warehouse itself. We build the data warehouse on a cluster of servers, described in Section 2.1. In Section 2.2 we present an overview of possible tools facilitating different aspects of the data warehouse, providing short reasoning why we choose certain tools over others.

## 2.1 Overview of the Cluster

In this section, we provide the details of the cluster dedicated to the data warehouse. The servers provided by IBM impose certain restrictions on the system, which we need to incorporate into the system architecture.

The cluster consists of five virtual servers, each having 4 CPUs and 16GB of memory. Additionally, all servers contain a pre-installed blank image of CentOS 8.5 without GUI, and all servers are able to connect to the internet. For security reasons, the incoming communication to the servers is protected by VPN.

From the setup of the cluster, two restrictions follow. First, all virtual servers are located in the same location and on the same rack. This means that we are not able to utilize the rack-awareness functionality of certain tools to further improve the fault tolerance of the data warehouse.

Second, many of the tools presented in the next section are based on the master-slave concept. In this concept, certain servers act as masters, handling metadata and coordinating computations, while other servers act as workers, handling the computation and storing data. Using this setup, a failure of a master node can lead to the unavailability of the whole cluster. To achieve high availability, many tools provide an option to define more than one master for a given cluster. As the cluster consists of

five servers, we are using only one master to maximize the resources as worker nodes. Thus, we use the same server as the master for all tools with a master-slave configuration. We are aware that this choice introduces a single point of failure into the data warehouse architecture.

## 2.2 Technologies Facilitating the Data Warehouse

This section provides an overview of certain aspects of data warehousing and presents a shortlist of possible tools facilitating each aspect. We also reason why we pick certain tools instead of others to build the data warehouse. The lists provided in this section are not exhaustive.

### 2.2.1 Data Transformation

One of the main requirements of data warehouses is to enrich the source data and provide it in an easily presentable format to business users either directly or via reports. This transformation is done by a dedicated tool, often called an ETL tool.

Apache Hadoop is a Java-based open-source framework that provides distributed storage and data processing of large data sets across multiple computers. It is a scalable, fault-tolerant, and designed to run on low-cost hardware [2]. The MapReduce framework provided by Hadoop allows users to do distributed computations following the *divide and conquer* approach, which enables tasks to be run in parallel and on multiple machines. A Hadoop cluster follows the master-slave architecture. It consists of one NameNode acting as a master that maintains the whole system and several DataNodes that store data and execute computations.

Apache Spark is a distributed computing engine capable of processing large amounts of data efficiently. It is a widely used tool with various applications across the big data field, mainly because of its performance and flexibility. Spark achieves this performance by keeping the data in the memory and not strictly following the MapReduce framework. Therefore intermediate results are more readily available for re-use. Spark can run on top of a Hadoop cluster, using YARN to orchestrate the computations, or it can be installed as a stand-alone cluster. In both cases, Spark uses the master-slave architecture.

For ETL, some of the conventional tools are also utilized in big data projects. The most common ETL tools are Informatica PowerCenter, Oracle Data Integrator, and IBM DataStage. These tools can also be installed on clusters under certain conditions. In contrast to Hadoop and Spark, these tools are licensed.

We choose Spark to facilitate the data transformation for our data warehouse as it is currently viewed as a business standard. We use Pyspark, a Python API for Spark,

to implement the ETL job for our data warehouse.

### 2.2.2 Data Storage

Apache Hive is an open-source data warehouse solution built on top of Hadoop. It uses Hadoop to store the data in the cluster and uses HiveQL, a language very similar to SQL, to query and manage data sets in a distributed file system using Hadoop's MapReduce framework. Data stored in Hive is presented in the well-known table format similar to relational database management systems.

As alternatives, we mention two commercial solutions to store data in cloud: Hortonworks Data Platform and Snowflake, which are not feasible for our needs.

Cassandra and HBase are also well-known distributed database management systems. Both specialize in handling transactional data instead of analytical workload.

Therefore, we use Hive as our distributed database management system adhering to the industry standards.

In big data systems, the data is often stored in human-readable CSV files or columnar file formats, providing more efficient access to the data during querying. The most common columnar file formats are ORC and Parquet. We use the latter to store data on the Hadoop Distributed File System utilizing Hive.

### 2.2.3 Installation and Configuration

We use Ansible, open-source configuration management and automation tool, to set up the cluster. Ansible eliminates the need to set up each server manually. It uses the *infrastructure as code* method, where the required state of each server is described in a YAML-like language, which can be executed against the cluster and versioned in git for easy traceability.

### 2.2.4 An Alternative Architecture

Currently, there is a trend to utilize Kubernetes to orchestrate job execution over a cluster of servers. This subsection presents an alternative architecture, where the servers are joined into a single Kubernetes cluster. Both Spark and Hive support execution on top of a Kubernetes cluster. The Kubernetes cluster's main advantage is the easy addition of further compute resources, while maintaining a Kubernetes cluster comes with an overhead. As our cluster is of a fixed size, this architecture is not feasible for us.

### 2.2.5 Cloud Native Architectures

Another new trend is an industry-wide push towards cloud solutions and cloud-based architecture. In this subsection, we present another commercial architecture based on the capabilities of Amazon AWS. As this solution is costly, it is not feasible for us.

AWS Glue or Spark on Amazon EMR can be used for data transformation. AWS Glue is a serverless data transformation offering based on Apache Spark. Amazon also provides several database management systems for data warehouses, e.g., RedShift, storing data in S3 and using RedShift Spectrum to query data, or Apache Hive on Amazon EMR.

# Chapter 3

# The Logical Model and Data Transformation

In this chapter we present the dimensional design of the data warehouse implemented in this thesis. We also provide the model of the source system for our data warehouse and the description of ETL transformations between the source system and the data warehouse.

## 3.1 Source System

The source system for our data warehouse is a dump in a .csv format from an operational database recording streaming-related user interactions.

The interactions are stored in the table *Event* which records the date and time of interaction, the email address of the user, the title of the streamed track, the name of the artist and album, and the specific action the user performed. We are interested in two types of user actions — playing and pausing a song.

The table *User* contains the information from a user's profile. Each user is uniquely identified by their email address, making it the table's primary key. Additionally, a user's name, birthday, gender, subscription plan type, and complete physical address are stored.

The music-related information is stored in tables *Track* and *Album*. The *Track* table holds the information about individual tracks — title, artist, album, genre, and duration of a track. The composite primary key of this table consists of the columns title, artist, and album since we need all three to identify a particular track.

Tracks are naturally organized into albums which are stored in the table *Album*. Each album has a title, artist, release date, and publisher. The primary key is the combination of columns title and artist.

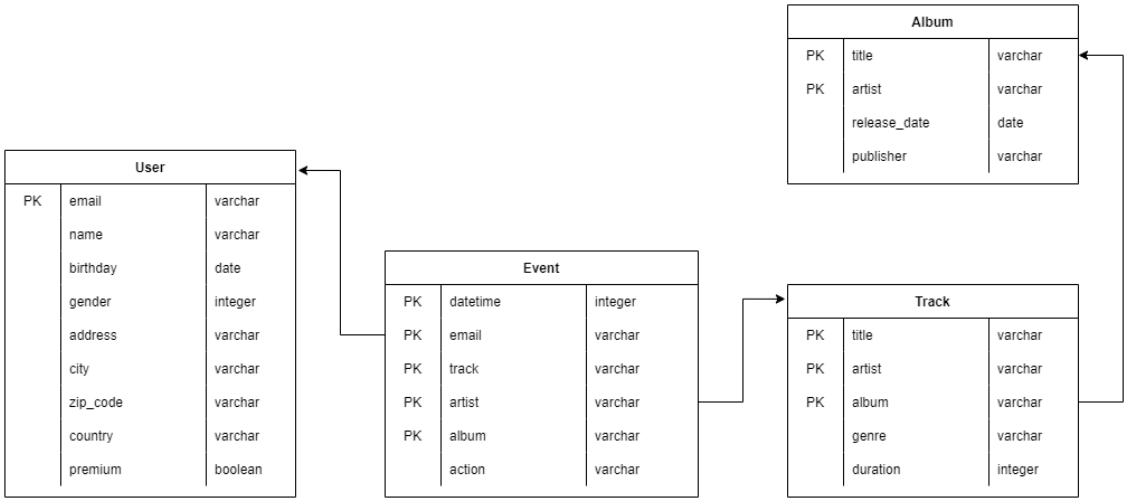


**Figure 3.1:** The logical model of the source system

## 3.2 Data Warehouse

In this section, we present the logical model of the data warehouse. The design process follows the steps of Kimball's four-step dimensional design, which consists of selecting a business process, defining the grain and identifying the dimensions and the facts.

### 3.2.1 Business Process

Based on Chapter 1 the whole design of a data warehouse depends on the modeled business process. For this reason, identifying a business objective is the first step in a data warehouse design process.

The client of our data warehouse is a fictional music streaming service interested in what music their users listen to. Moreover, they need to know what music is popular among specific user groups. Therefore, the business process we model is music streaming.

### 3.2.2 Grain

The fictional client wants to understand who their users are and the correlation between tracks and users. Both of these questions require a low level of grain.

To determine the grain, we specify what a single row in a fact table describes. In our case, it is one row per each instance of a streaming of a song. With this grain, the business users are able to answer questions such as which artists are popular among teenage male users or the difference in user music preferences on weekends versus weekdays. Our fictional client deems that the ability to answer such questions is crucial for successful targeted marketing.

### 3.2.3 Dimension Tables

Dimension tables provide context for facts – they describe various aspects of events recorded in the fact table. We define grain as "one row per an instance of a song stream", which is expanded into "which user played which song and for how long" using the corresponding dimensions. Using this wording, we identify the dimensions and dimension tables needed for the design.

The dimension table *User_Dim* contains the following attributes: the user's email, name, birthday, gender, country, and region. These dimensions provide details about individual users necessary for categorizing them into target groups. The email attribute serves as a natural key used for identifying specific users. Although the source system records the full address of a user, we consider country and region sufficient to fulfill the business objective.

Next, we define the dimension table *Track_Dim* providing information about songs. It contains the song's title, artist, album, genre, duration in milliseconds, release date, and publisher. Based on the design of the source system, to uniquely identify a row, we need the combination of the attributes title, artist, and album. Therefore, we choose this triplet to be the natural key of the table.

The *Calendar_Dim* dimension table is a common feature in the dimensional design. It stores individual dates, and it contains numerous attributes describing year, month, week, and day in various ways, e.g., we record the full name of a month together with its abbreviated and numeric representation. The abundance facilitates filtering the records in a fact table for the business user. This table is not extracted from the source system, but it is populated with dates in advance before the first ETL process is executed. The whole schema of the *Calendar_Dim* dimension, as well as the other two dimensions, can be seen in Figure 3.2.

Additionally, all dimension table contains a unique synthetic (or surrogate) key, acting as a primary key internally, in the scope of the data warehouse.

The last dimension we include in our dimensional model is the type of subscription plan of the user streaming a song. However, subscription management on its own is a separate business process, which is out of the scope of our use case. Therefore, we consider the subscription plan of the user as a context of each instance of song streaming, depicting the active subscription plan of the user when a particular song was started. Hence, we add the subscription as an additional attribute to the fact table to avoid redundant dimension tables in our model. Dimension attributes stored in a fact table are called degenerate dimensions.

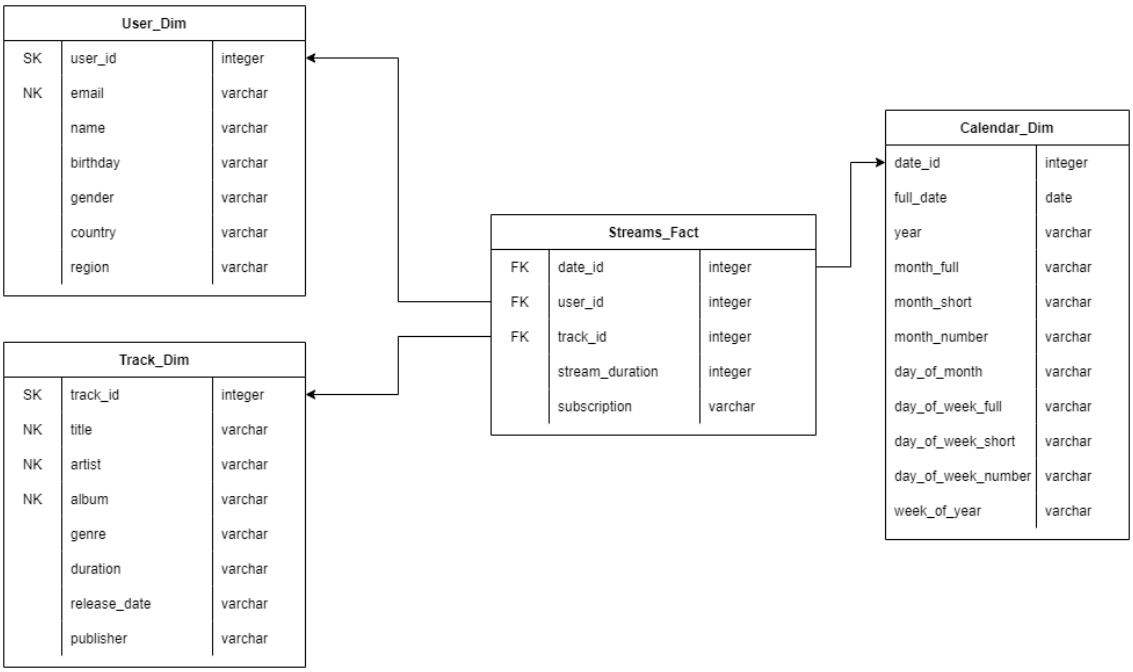


**Figure 3.2:** The logical model of the data warehouse

### 3.2.4 Fact Table

The last table in our dimensional design is a fact table, named *Stream_Fact*. It records the measurements, also called facts, which serve as metrics for analysis. The only fact in our table is the attribute stream_duration, the number of milliseconds a particular user spent listening to a given song. The table also contains foreign keys *user_id*, *date_id*, and *track_id* to the *User_Dim*, *Calendar_Dim* and *Track_Dim* respectively.

## 3.3 Data Transformation

Based on the logical models we propose in Sections 3.1 and 3.2, the schemata of the source system and the data warehouse differ considerably. Therefore the source data is transformed during the load of the data warehouse tables. In order to determine how the source data need to be processed and transformed during an ETL process, we define a set of rules describing the relationship between the columns of the source system and the columns of the data warehouse.

To load data into the dimensional tables, the natural key, the primary key of the source system, is used to identify the entities. The surrogate keys are auto-generated. All other columns are defined using one of the methods collectively called *Slowly Changing Dimensions*, described in Section 1.2.

The mapping for the *Track_Dim* table is straightforward. The natural key of the table consists of attributes title, album, artist. Attributes genre and duration are

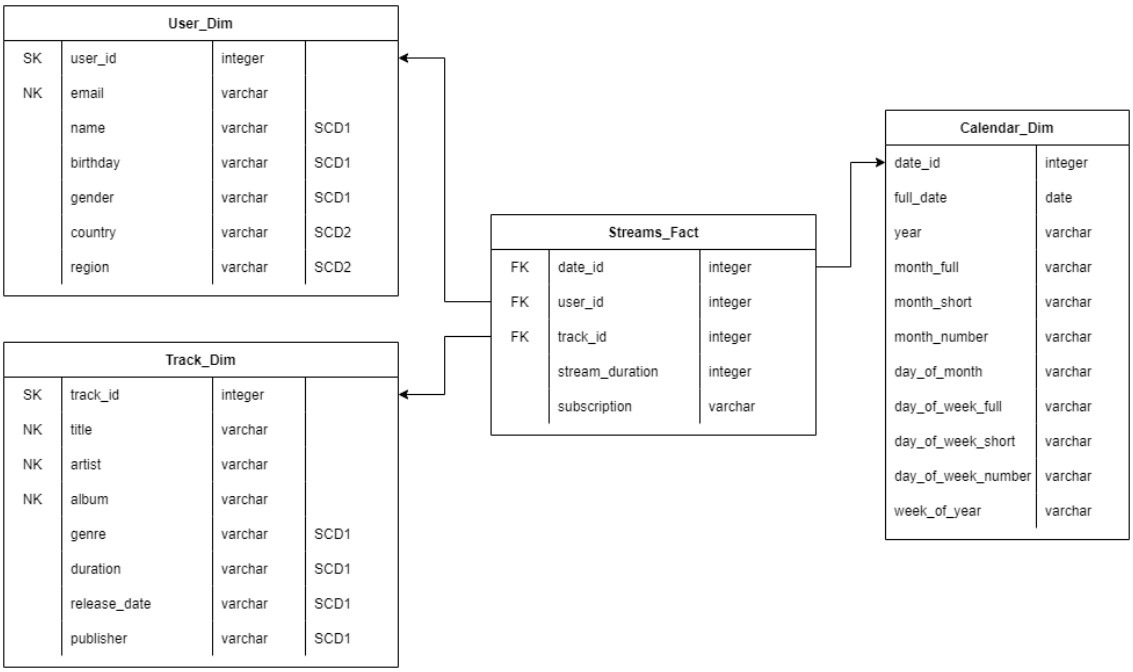


**Figure 3.3:** Slowly Changing Dimensions identified for dimensions

directly mapped from the *Track* source table using SCD type 1 logic. The release_date and publisher attributes are directly mapped from the *Album* table using SCD type 1 logic. There is no need to additionally conform any attributes.

For the *User_Dim* table, the natural key consists of the email attribute. Attributes name and birthday are loaded from the *User* table using SCD type 1 logic. The gender attribute in the source table is represented as an integer with possible values zero, one, and two representing *unknown*, *male*, and *female* respectively. We convert the integer representation to a string representation for ease of understanding. The gender attribute uses SCD type 1 logic. The country attribute is pulled from the *User* source table using an SCD type 2 logic to ensure the correctness of historical facts when a user moves. To populate the region column, we employ the method called data enrichment — we use external sources to obtain the information not present in the source system. In this case, we use an external table to derive the value of the region attribute based on the country column.

The *Streams_Fact* table contains a degenerate dimension subscription, which we map from the column premium of the *User* table from the source system. The value of the subscription attribute never changes, and its value is either *premium* if the premium attribute is true or *basic* otherwise.

In order to calculate the only measurement in our table — stream_duration — we need to identify the actions corresponding to the start and end of streaming of a particular song in the *Events* table in the source system. The stream_duration attribute

is computed as the time difference between the end and start of given streaming.

We assume that the source system registers any streaming interruption as a pause action, not only the manual pausing by a user (e.g., the song ends, the streaming service closes). If a user jumps to a different part of the song, the system successively registers a pause action and a play action. As a result, there is a corresponding pause action for each play action in the source system.

# Chapter 4

# Implementation

In this chapter, we present the technical details of the implementation of the error-correction method within the scope of this thesis, namely Dependency Analysis and naive error correction. We compare the efficiency of these methods on test data sets of different sizes. Additionally, we compare their implementation complexity compared to a non-error-corrective implementation.

## 4.1 Obtaining Test Data

To test the implemented ETL process, we generate data sets representing the data coming from the operational database. This section describes the open-source data sets we use and how we transform them to create the source data.

A user's name comes from the data set [16] containing thousands of first names and surnames. The data set of email domains [4] is used for the email address generation. For user's address details, we use the data set of European streets and their zip codes [13] and the data set of world cities [11]. The data set [11] is also used to create the external table needed for data enrichment in the data warehouse ETL process. Aside from some minimal filtering, all of the previously mentioned data sets are usable as they are without the need for additional editing.

To generate a user for the final *User* table, we choose a random first name, surname, and email domain which are then used to create the unique email address. We then pick a random record from the street and city data sets, which comprise the user's address details. Lastly, we generate a random birthday, gender, and subscription type.

The *Track* and *Album* source tables are created from the data sets [9] and [14]. The records in both data sets contain the title and author of a track, but they differ in the rest of the track properties they capture. Out of the columns relevant to the final source tables, the data set [9] contains the duration (in milliseconds) and release date attributes, while the data set [14] has the album attribute. Neither of the data sets

contains information about publishers and genres.

Track records that do not contain the information about the album are considered singles, meaning that we use the same value for both the track title and album title. We produce a separate *Album* table with the album, artist, and release date properties keeping only one record per album entity. The columns track title, album, artist, and duration of the original table constitute the *Track* table. We then fill in the missing or incomplete values in the release date and duration columns. Finally, we use the list of publishers from the website [12] to populate the publisher column in the *Album* table and the list of genres from the website [5] to populate the genre column in the *Track* table.

The data sets *User* and *Track* are used to generate event records for the *Event* data set. First, we pick a random user and track. Each selected pair adds two new records to the *Event* data set — a play event and a pause event. We then generate a play timestamp and a percentage representing how much of the track had been streamed before it was paused. The percentage is then used to calculate a pause timestamp based on the track duration attribute.

## 4.2 Provisioning

In this section, we describe the setup of tools needed to implement the data warehouse. We use Ansible playbooks to facilitate the process.

Hadoop and Spark use a master-slave architecture for their clusters. There are a few configuration steps that differ depending on a role of a server, so we decide which server acts as a master for both tools. Unless stated otherwise, we act on all nodes.

First, we install JDK since it is required for running both Hadoop and Spark. Additionally, we install Python as a prerequisite for Spark Pyspark API. We distribute the master's SSH key to the slave machines to allow communication between the nodes. Next, we download Hadoop and Spark. To configure all required properties, we create the configuration files for both tools and place them within their respective configuration directories. Finally, we install Hive on our master node in the same fashion. The final state of our cluster can be seen in Figure 4.1.

## 4.3 Basic Implementation

We process each target table as an independent Spark job. The order in which the jobs are launched adheres to the relationship between facts and dimensions established in Chapter 1 — the fact table is loaded only after the successful loading of the dimension tables.

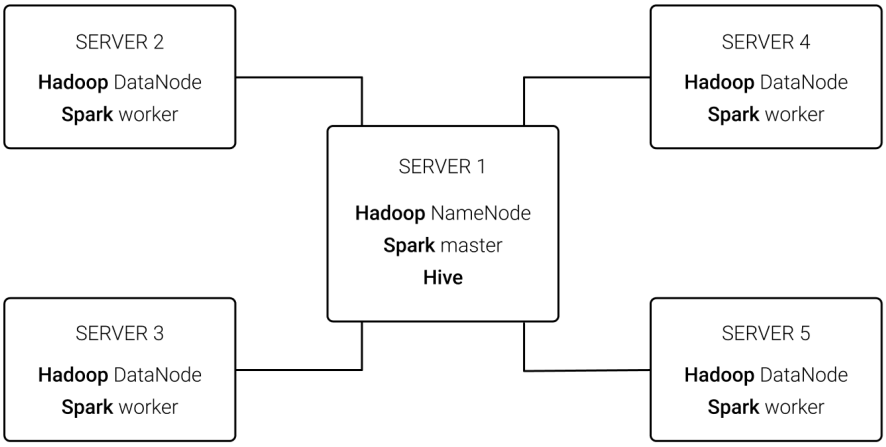


**Figure 4.1:** Server cluster

As the dimension table *Calendar_Dim* does not rely on the source data, it can be created independently of the rest of the ETL process. When the ETL process is launched, we check if the table exists in the data warehouse and create it if it does not.

The first common step of the ETL jobs loading *User_Dim* and *Track_Dim* dimension tables is reading the corresponding source tables from HDFS. We then apply transformations specific to each dimension table as described in Chapter 3 as well as some additional transformations related to data quality, such as elimination of null values and data validation.

For the *User* table, we start the transform stage by removing the premium column since it does not appear in the *User_Dim* table. In order to populate the region column, which is not present in the *User* table, we use the external table stored in HDFS as a CSV file. We then convert the values in the gender column from the integer representation to the string representation and validate the dates in the birthday column — we check whether a date is in the correct format and is an actual date.

To obtain all the columns for the *Track_Dim* dimension table, we join the *Track* table with the *Album* table. Subsequently, we validate the release_date column values.

It is expected that null values appear in the source tables or that a transformation results in null values in the target table. The standard practice in a data warehouse design is to avoid nulls because they can cause inconsistent analysis results. As a general rule, we replace any null value in a source table with the string *unknown*. Moreover, we insert one specific record to each dimension table, including the *Calendar_Dim*. The ID attribute of this record is -1, and the rest of the columns are filled with the value *unknown*. We utilize this record to avoid nulls in the foreign key columns of the fact table, which may emerge if the corresponding dimension record is not found.

Before loading the dimension tables *User_Dim* and *Track_Dim*, we compare the

records with the data already in the data warehouse to identify new and changed rows based on the natural keys.

We generate a unique ID for all records representing a new entity. Additionally, we add a new maintenance attribute is_current with the value *True* to new records in the *User_Dim* table. Its function is to differentiate between record versions since the table contains an SCD type 2 attribute.

When identifying the changes in the SCD type 2 attribute country, we compare the records obtained in the ongoing ETL process only to those records from the stored *User_Dim* which is_current attribute is *True*. If the change has occurred, the new version of a record is inserted as a new row — a new unique ID is generated and its is_current attribute is set to *True*. Moreover, we update the value of the is_current attribute in the previous version to *False*.

The transformation of the *Event* source table to the *Streams_Fact* table starts with validation of the timestamps in the datetime column and elimination of null values. Before computing the stream_duration fact, we create a temporary column action_number, which serves as an event serial number within a group of events with the common user and track. The reason for this column is the situation in which the same user streams the same track multiple times. In such a case, we would not be able to identify the correct pairings of play and pause actions solely using the data present in the *Event* table. We assume that the source system logs user actions as they occur, and thus they are ordered by the timestamp. The action_number column allows us to rely on the fact that the action_number attributes of related events always differ by one. The example of the data after the computation of the action_number column can be seen in Figure 4.2. Consequently, we populate the stream_duration column by self-joining the table, filtering the event pairs based on their action_number attributes, and computing a stream_duration value from the timestamps columns, as can be seen in Figure 4.3. Finally, we substitute the natural keys for the foreign keys to the dimension tables.

All tables are first loaded into a new temporary table within the data warehouse. We then "swap" the table with the previous version — we change the name of the previous version, then change the new table's name to the target table, and finally, drop the previous version. We use this technique to avoid the corruption of the target tables caused by failed loading. As a result, we can directly employ the naive resumption method without additional steps.

| | datetime | email | track | artist | album | action | action_number |
|---|---|---|---|---|---|---|---|
| 261 | 2021-01-01 07:03:04 | nawel7@mailcan.com | Échenme la tierra encima | Pedro Infante | Échenme la tierra encima | play | 1 |
| 262 | 2021-01-01 07:04:41 | nawel7@mailcan.com | Échenme la tierra encima | Pedro Infante | Échenme la tierra encima | pause | 2 |
| 263 | 2021-01-01 10:24:16 | nawel7@mailcan.com | Šagrénová koža | Miro Žbirka | Šagrénová koža | play | 1 |
| 264 | 2021-01-01 10:26:34 | nawel7@mailcan.com | Šagrénová koža | Miro Žbirka | Šagrénová koža | pause | 2 |
| 265 | 2021-01-01 05:43:55 | neculina.c@inoutbox.com | Det börjar verka kärlek banne mig | Claes-Göran Hederström | Det börjar verka kärlek banne mig | play | 1 |
| 266 | 2021-01-01 05:44:53 | neculina.c@inoutbox.com | Det börjar verka kärlek banne mig | Claes-Göran Hederström | Det börjar verka kärlek banne mig | pause | 2 |
| 267 | 2021-01-01 22:49:15 | neculina.c@inoutbox.com | Det börjar verka kärlek banne mig | Claes-Göran Hederström | Det börjar verka kärlek banne mig | play | 3 |
| 268 | 2021-01-01 22:50:52 | neculina.c@inoutbox.com | Det börjar verka kärlek banne mig | Claes-Göran Hederström | Det börjar verka kärlek banne mig | pause | 4 |

**Figure 4.2:** Data after the computation of the action_number attribute

```
13      def compute_stream_duration(self):
14          self.stream = self.stream.withColumn("action_number",
15                                               row_number().over(Window
16                                                                 .partitionBy("email", "track", "artist", "album")
17                                                                 .orderBy("datetime")))
18
19          self.stream = self.stream.alias('play') \
20              .join(self.stream.alias('pause'),
21                    [col('play.track') == col('pause.track'), col('play.artist') == col('pause.artist'),
22                     col('play.album') == col('pause.album'), col('play.email') == col('pause.email')],
23                    'left') \
24              .where((col('play.action') == 'play') & (col('pause.action') == 'pause') &
25                     (col('pause.action_number') == col('play.action_number') + 1)) \
26              .select(date_format(col('play.datetime'), 'yyyy-MM-dd').alias('date'),
27                      'play.email', 'play.track', 'play.artist', 'play.album',
28                      to_timestamp(col('play.datetime'), 'yyyy-MM-dd HH:mm:ss').alias('play_time'),
29                      to_timestamp(col('pause.datetime'), 'yyyy-MM-dd HH:mm:ss').alias('pause_time')) \
30              .withColumn("stream_duration",
31                          (col("pause_time").cast("long") - col("play_time").cast("long"))) \
32              .drop('pause_time', 'play_time')
```

**Figure 4.3:** The computation of the stream_duration attribute

## 4.4 Dependency Analysis

For the Dependency Analysis method, the execution of a job is divided into three stages — extract, transform-load, and swap stage.

Prior to launching the ETL process, we create two temporary tables to store the information required for the Dependency Analysis method. The dependencies between the job stages are stored in the *config_log* table. We describe how these dependencies are derived in Section 1.6. The *cmd_log* stores the execution results. For each job stage of our ETL process, we insert one record with the initial value of the result column being *False*.

Storing the *config_log* and *cmd_log* within the data warehouse would be highly inefficient. The more suitable solution is to use a relational operational database.

| | etl_cmd | prev_step |
|---|---|---|
| 1 | user_dim_tl | user_dim_ext |
| 2 | user_dim_swap | user_dim_tl |
| 3 | track_dim_tl | track_dim_ext |
| 4 | track_dim_swap | track_dim_tl |
| 5 | streams_fact_tl | streams_fact_ext |
| 6 | streams_fact_tl | user_dim_tl |
| 7 | streams_fact_tl | track_dim_tl |
| 8 | streams_fact_swap | streams_fact_tl |

**Figure 4.4:** The *config_log* table used in our Dependency Analysis implementation

```
SELECT config.*
FROM config_log config
LEFT JOIN (SELECT etl_cmd
           FROM cmd_log
           WHERE result) results
ON config.prev_step = results.etl_cmd
WHERE config.etl_cmd = 'user_dim_tl' AND results.etl_cmd IS NULL;
```

**Figure 4.5:** Query to determine the execution result of predecessors

We choose MySQL since it is an open-source, easy-to-use option with a simple setup process. The MySQL server runs on the master node of our cluster. The auxiliary tables created are small and operational in nature; hence a standard relational database management system is favored over distributed systems, like Cassandra or HBase.

In the basic implementation, a job is either executed fully or not at all. It is not the case when we use the Dependency Analysis method, in which the outcome of the ETL process may be a partially executed job. At the end of each stage, we store an intermediate result as a temporary table in the data warehouse and update the stage execution result in the *cmd_log* to *True*. The following stage then reads and processes the temporary table instead of processing the extracted table directly. This modification allows us to resume a job from the first failed or skipped stage during a resumption procedure. We condition the execution of a stage by the successful execution of its predecessor according to the *config_log*. Moreover, we consult the *cmd_log* to check whether the stage has not been already successfully executed in the previous run to avoid superfluous operations. In case of a failure, we do not need to perform any additional steps; the ETL process is directly relaunched.

## 4.5 Comparison and Findings

We shall use the average overall execution time of a failed ETL process as our primary metric. We test the methods in two failure scenarios — a failure during dimension table processing and during fact table processing. As can be seen in Figure 4.7, the Dependency Analysis method performed better in both cases. In case of a fact table failure, the execution time is reduced considerably.

To be able to measure the benchmarks for the error-handling methods, we simulate erroneous/faulty execution as follows. We choose a random table from our date warehouse schema, which fails in the current ETL run. The reason for this is to differentiate between the dimension table and fact table failures. We then generate a random number of seconds, after which we interrupt the processing of a given table.

Using the simulation of erroneous computations, we measured several execution times for each method with different input sizes. We present our findings in Figure 4.6

Next, we present the aggregated results in the bar chart in Figure 4.7, comparing

| | dataset_size | failed_table | naive_avg_time | da_avg_time |
|---|---|---|---|---|
| 1 | 677.9 M | dim | 0:05:37 | 0:04:48 |
| 2 | 677.9 M | fact | 0:06:56 | 0:04:34 |
| 3 | 2.1 G | dim | 0:07:24 | 0:06:21 |
| 4 | 2.1 G | fact | 0:08:04 | 0:05:29 |
| 5 | 1.3 G | dim | 0:06:09 | 0:05:12 |
| 6 | 1.3 G | fact | 0:07:40 | 0:05:01 |

**Figure 4.6:** Test results

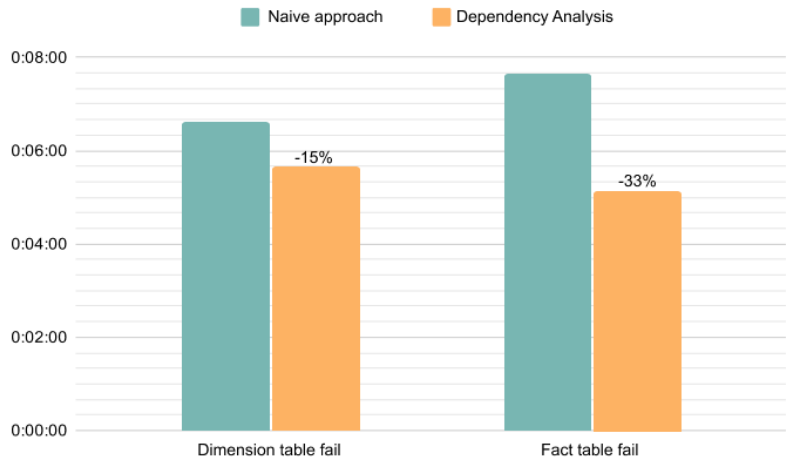


**Figure 4.7:** Performance of methods by the failed table

the average execution time of different methods under different circumstances.

Figure 4.8 shows the average performance of an ETL with and without a failure. We conclude that the additional operations in the Dependency Analysis have minimal effect on the execution time of the regular, non-faulty ETL process.

Compared to the basic implementation, the Dependency Analysis method requires an additional operational database to ensure efficient execution. Moreover, we need to perform a few pre-ETL steps to create logs. Incorporating execution conditions and checks based on the logs is relatively straightforward and does not call for any major logic restructuring of an ETL pipeline.

We deem the Dependency Analysis as an effective method for ETL resumption in a big data environment. Generally speaking, it provides a satisfactory performance improvement without substantial deceleration of a regular ETL run. It needs to be said that based on the substance of the Dependency Analysis method, the difference in performance grows with the number of tables in a data warehouse schema. For this reason, we expect the optimization to be even more significant in a real-life setting with more tables.

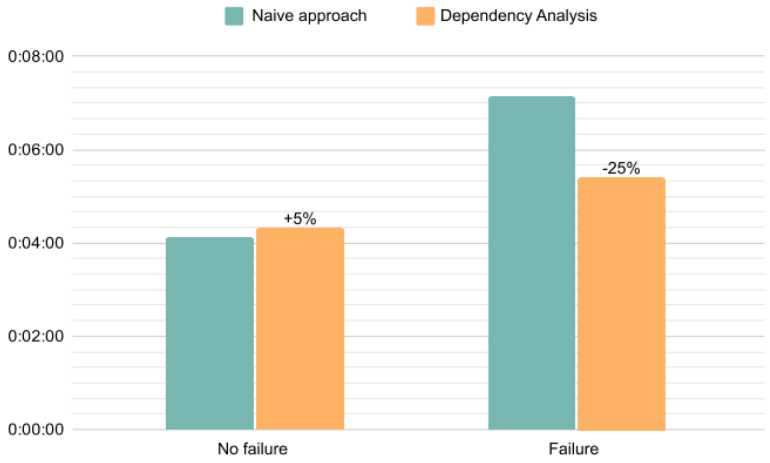


**Figure 4.8:** Performance of methods by the presence of failure

# Conclusion

The main goal of this thesis is to analyze the efficiency of the Dependency Analysis error-handling method introduced in paper [17] in a distributed data warehouse environment. This is achieved by creating a fabricated use case with minimal applicable complexity, which we use to benchmark the efficiency of Dependency Analysis compared to a naive error-handling method.

We build our data warehouse on a cluster of five computers using industry-standard big data tools. In the core of the data warehouse lies Hive, facilitating data storage on top of a Hadoop cluster. The data transformation process is handled by Spark on top of the same cluster. Using this cluster, the execution time of Dependency Analysis is measured and compared to naive error handling on several different input data sets of different sizes.

As a result of this testing, we conclude that Dependency Analysis is an efficient error-handling method in a distributed data warehouse, becoming especially efficient in the case of fact tables, where it performs around 33% faster than the naive error-handling method. The "cost" of this efficiency is a slightly increased code complexity and a need for certain auxiliary structures, namely two auxiliary tables, which are operational by nature.

In our thesis, we store the auxiliary tables in a MySQL server instance. In a complex real-life system, a traditional database management system might become a bottleneck during the parallel execution of several jobs. The auxiliary tables are small and frequently accessed; therefore, it might be interesting in the future to examine the possibility of storing them in distributed in-memory data stores, such as Redis or Memcached.

# Bibliography


[1] Christopher Adamson. *Star schema: The complete reference*. McGraw-Hill, 2010.

[2] Apache Software Foundation. HDFS Architecture. `https://hadoop.apache.org/docs/r3.0.3/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`.

[3] Azure Architecture Center. Data warehousing in Microsoft Azure. `https://docs.microsoft.com/en-us/azure/architecture/data-guide/relational-data/data-warehousing`.

[4] fnando. Email Domains Dataset. `https://github.com/fnando/email_data`.

[5] Gemtracks Stuff. List of genres. `https://www.gemtracks.com/guides/view.php?title=complete-list-of-music-genres&id=298`.

[6] Marcin Gorawski and Pawe Marks. Checkpoint-based resumption in data warehouses. In *Software Engineering Techniques: Design for Quality*. Springer US, 2007.

[7] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Wiley, 2013.

[8] Wilburt Juan Labio, Janet L Wiener, Hector Garcia-Molina, and Vlad Gorelik. Efficient resumption of interrupted warehouse loads. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 46–57, 2000.

[9] Lehak Narnauli. Spotify Datasets. `https://www.kaggle.com/datasets/lehaknarnauli/spotify-datasets`.

[10] Sergio Luján-Mora and Enrique Medina. Reducing inconsistency in data warehouses. 2001.

[11] Matthew Proctor. World City Data. `https://www.matthewproctor.com/worldwide_cities`.

[12] Music Publishers Association of the United States. List of publishers. `https://www.mpa.org/all-publishers/`.

[13] OpenAddresses. OpenAddresses - Europe. `https://www.kaggle.com/datasets/openaddresses/openaddresses-europe`.

[14] Rodolfo Figueroa. Spotify 1.2m+ Songs. `https://www.kaggle.com/datasets/rodolfofigueroa/spotify-12m-songs`.

[15] Alkis Simitsis, Kevin Wilkinson, Umeshwar Dayal, and Malu Castellanos. Optimizing ETL workflows for fault-tolerance. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 385–396. IEEE, 2010.

[16] smashew. Name Dataset. `https://github.com/smashew/NameDatabases`.

[17] Shitao Tu and Lanjuan Zhu. An optimized ETL fault-tolerant algorithm in data warehouses. In *2013 IEEE Third International Conference on Information Science and Technology (ICIST)*, pages 484–487. IEEE, 2013.