



Assignment of bachelor's thesis

Title:	Parallelization of ETL processes DW CTU - case study
Student:	Kristina Zolochevskaia
Supervisor:	Ing. Michal Valenta, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of the thesis is to propose an appropriate parallelization of the current ETL processes of the CTU data warehouse (DW CTU) and to verify the proposal on a relevantly selected example (part of the ETL process) in the form of POC (Proof of Concept). Two concurrent bachelor theses are proposed, each of them trying to solve the problem using a different technology. In the conclusion of the theses there will be an evaluation of the advantages and disadvantages of each respective solution, which will allow to choose a more favorable path for further DW CTU development.

1. Along with the concurrent bachelor thesis, specify the requirements for parallelization of DW CTU ETL processes.
2. Along with the concurrent bachelor thesis, select a suitable part of DW CTU ETL processes to be parallelized in the chosen technology.
3. Undertake research on two or more tools suitable for use in this problem.
4. Select one of the tools from step 3, analyze its possibilities in terms of fulfilling the requirements from step 1.
5. Propose and implement the parallelization of selected DW CTU ETL part from step 2 using the tool chosen in step 4.



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

6. Evaluate the implemented solution in terms of accomplishing the requirements from step 1 and also in terms of future use for the management of the entire DW CTU ETL process.



Electronically approved by Ing. Michal Valenta, Ph.D. on 6 December 2022 in Prague.



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Bachelor's thesis

Parallelization of ETL processes DW CTU – case study

Kristina Zolochovskaia

Department of Software Engineering
Supervisor: Ing. Michal Valenta, Ph.D.

May 11, 2023

Acknowledgements

First and foremost, I would like to express my gratitude to my academic supervisor, Ing. Michal Valenta, Ph.D., for their priceless advice, human approach, and the opportunity to write this thesis under their guidance.

Furthermore, the support of my dear colleagues has been invaluable, especially from Bc. Adam Makara and Adam Marhefka.

Last but certainly not least, I would like to extend my deep appreciation to my friends and cherished parents, whose faith and support have always encouraged me and given me the strength to overcome difficulties.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 11, 2023

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2023 Kristina Zolochevskaia. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Zolochevskaia, Kristina. *Parallelization of ETL processes DW CTU – case study*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Tato bakalářská práce se zabývá paralelizací procesů Extract, Transform, Load (ETL) v rámci datového skladu Českého vysokého učení technického (DW ČVUT) s cílem zlepšit výkon. Stávající řešení, které se opírá o sekvenční přístup, je časově náročné a omezuje efektivitu systému. Hlavním cílem této studie je navrhnout nový paralelní přístup a implementovat důkaz konceptu (POC) pomocí odlišné technologie jako alternativu k současnému nástroji Pentaho Data Integration (PDI).

Byl proveden důkladný přehled literatury, aby byly identifikovány potenciální řešení, přičemž Apache Airflow se ukázal jako moderní a spolehlivá možnost. Implementace se skládá ze dvou hlavních komponent: ručně kódovaných ETL procesů v Pythonu a Apache Airflow, který orchestruje, monitoruje, organizuje a plánuje paralelní provádění úkolů ETL. Nové řešení úspěšně snížilo čas načítání na polovinu, což dokazuje jeho účinnost při zlepšování výkonu DW ČVUT.

Hlavním přínosem této práce je vývoj efektivnějšího paralelního ETL řešení, které snižuje zatížení serverů Výpočetního a informačního centra (VIC) a uvolňuje prostředky pro ostatní procesy.

Klíčová slova paralelní ETL, datový sklad ČVUT, Apache Airflow, Python, pandas

Abstract

This bachelor's thesis addresses the parallelization of Extract, Transform, Load (ETL) processes within the Data Warehouse of Czech Technical University (DW CTU) to improve performance. The existing solution, which relies on a sequential approach, is time-consuming and limits the system's efficiency. The primary objective of this study is to propose a new parallelization approach and implement a proof of concept (POC) using different technology as an alternative to the current Pentaho Data Integration (PDI) tool.

A thorough literature review was conducted to identify potential solutions, with Apache Airflow emerging as a modern and reliable option. The implementation consists of two main components: hand-coded ETL processes in Python and Apache Airflow, which orchestrates, monitors, organizes, and schedules the parallel execution of the ETL tasks. The new solution successfully decreased the loading time by half, demonstrating its effectiveness in enhancing the DW CTU's performance.

The main contribution of this thesis is the development of a more efficient parallel ETL solution, which reduces the workload on the Computing and Information Centre (VIC) servers and frees up resources for other processes.

Keywords parallel ETL, data warehouse CTU, Apache Airflow, Python, pandas

Contents

Introduction	1
Aims	3
I Theoretical Background	5
1 Data Warehouse	7
1.1 Notions and definitions	7
1.2 Data Warehouse Architecture	8
1.2.1 Inmon's Approach	9
1.2.2 Kimball's Approach	10
1.3 Data Modelling	12
1.3.1 Data Modeling Approaches	12
1.4 Data Integration	13
1.5 Business Intelligence	14
2 Extract, Transform, Load	17
2.1 ETL Process Stages	17
2.1.1 Extract	17
2.1.2 Transform	19
2.1.3 Load	21
2.2 Parallel Optimization of ETL Processes	22
2.2.1 Parallel Processing of ETL	23
II Practical Implementation	25
3 Current State Analysis of the CTU Data Warehouse	27
3.1 CTU Data Warehouse Architecture	27

3.1.1	Staging Layer	28
3.1.2	Target Layer	29
3.1.3	Access Layer	29
3.2	Current ETL processes of CTU Data Warehouse	30
3.2.1	Source to Pre-Stage, or Export	31
3.2.2	Pre-Stage to Stage-Increment, or Transform	32
3.2.3	Stage-Increment to Target, or Load	34
3.3	Orchestration and Logging	35
4	Requirements Analysis and Specification	37
4.1	Parallelization Requirements	37
4.1.1	Functional Requirements	38
4.1.2	Non-functional Requirements	39
4.2	Selection of ETL Parallelization Part	39
5	Parallelizing ETL: Tool Research and Analysis	41
5.1	Research and Selection	41
5.2	Analysis	42
5.2.1	Apache NiFi	42
5.2.2	Talend	43
5.2.3	Apache Airflow	43
5.3	Final Assessment	44
6	Implementation	47
6.1	Introduction to Apache Airflow	47
6.2	Basic Components of Apache Airflow	48
6.2.1	Workloads	49
6.2.2	Control Flow	49
6.2.3	Executors	49
6.2.4	User Interface	50
6.3	Design of Implementation	50
6.3.1	ETL Processes Design	50
6.3.2	Airflow Configuration	51
6.4	Initial Setup	53
6.5	ETL Implementation	54
6.5.1	Database Connections	54
6.5.2	Extraction	56
6.5.3	Transformation	57
6.5.4	Loading	58
6.6	Task Parallelism Implementation	59
6.7	Scheduling and Orchestration	62
6.7.1	UI Graphs and Task Monitoring	62
6.7.2	Logging and Error Handling	62
6.7.3	Task Statuses and Dependencies	62

6.7.4	Crontab Time Defining and Scheduling	62
6.7.5	Configuration and Customization	63
6.8	Testing	63
6.9	Further Development and Optimization	64
7	Evaluation	65
7.1	Comparison	65
7.2	Requirements Evaluation	67
7.3	Result	68
	Conclusion	71
	Bibliography	73
A	List of Acronyms	77
B	Apache Airflow Screenshots	79
C	Contents of Digital Attachment	87

List of Figures

1.1	Inmon’s top-down approach [6]	9
1.2	Kimball’s bottom-up approach [6]	11
2.1	The topology of ODBC in the ETL process [3]	18
2.2	Overview of entire ETL process [17]	19
3.1	CTU DW 3.0 Architecture [32]	28
3.2	Current ETL processes in PDI [34]	31
3.3	‘TEKOSOBY_CHECK_JOB’ job in PDI [34]	31
3.4	‘J_MAKE_INCREMENT’ job in PDI [34]	32
3.5	‘t_osob_osoba_kos_stara’ transformation in PDI [34]	35
6.1	Architecture of Apache Airflow and Its Components [37, 36]	48
6.2	Final Architecture [37, 36]	52
6.3	Airflow Working Directory Tree Structure	54
6.4	Apache Airflow Connections: KOS – Oracle [37]	55
6.5	Apache Airflow DAG – Grades Parallel ETL Tasks [37]	61
B.1	Apache Airflow – DAG List with Tasks Running [37]	80
B.2	Apache Airflow – Grades DAG [37]	81
B.3	Apache Airflow – KOS DAG [37]	82
B.4	Apache Airflow – EZOP DAG [37]	83
B.5	Apache Airflow – Usermap DAG [37]	84
B.6	Apache Airflow – Logs of Grades DAG [37]	85

List of Tables

4.1	Dependencies between selected tables	40
5.1	Tool Analysis	44
7.1	Sequential and Parallel Loading: Comparison	66

Introduction

This year marks the 10th anniversary of Czech Technical University Data Warehouse (DW CTU), which originated from Stanislav Kuznetsov's master thesis, 'Faculty Data Warehouse,' at FIT CTU [1]. Over the years, multiple students' theses have contributed to its development, resulting in three versions and the establishment of a full-fledged department of Computing and Information Centre (known as VIC ČVUT). The DW CTU integrates up to seven university and faculty systems operated within the CTU in Prague. It plays a critical role in providing dependable and high-quality data that serves various purposes, including generating trustworthy reports and other outputs for decision-making by CTU leadership and individual departments, facilitating analysis and research, and supplying data to support various university web endpoints. Although the DW CTU currently covers only the Study domain, it processes around 450 GB of pure text data. As new data is added daily and new systems are integrated regularly, the uploading time has increased, highlighting the need for optimization.

This thesis aims to improve performance, increase scalability, and enable better utilization of hardware resources by parallelizing Extract-Transform-Load (ETL) processes. Currently, the DW CTU executes ETL jobs sequentially, taking almost a full day to upload data, making it inefficient. To address this issue, two concurrent bachelor theses were proposed to study the possibilities and solve the problem in different ways. While my colleague, Adam Marhefka, in their bachelor thesis 'Parallelization of DW CTU ETL processes in the Pentaho tool', FIT CTU, (Slovak: 'Paralelizácia ETL procesov DW ČVUT s využitím nástroja Pentaho', FIT ČVUT) focuses on enhancing the existing implementation in Pentaho Data Integration (PDI), this thesis will explore other tools and create parallel ETL processing of data using a selected tool identified in previous studies. The solutions will be evaluated in terms of performance, deployment requirements, and potential for further development, to determine the best path for the future growth of the DW CTU.

Aims

The aim of this bachelor thesis is to propose an applicable approach for parallelizing the current ETL processes of DW CTU and to verify the proposal on a relevantly selected example (part of the current ETL processes graph) in the form of Proof of Concept (POC). The main objectives of this thesis are:

1. To establish the requirements for parallelization of DW CTU ETL processes along with the concurrent bachelor thesis.
2. To select a suitable part of DW CTU ETL processes graph to be parallelized in the chosen technology along with the concurrent bachelor thesis.
3. To undertake research on at least two tools suitable for addressing this problem.
4. To select one of the tools from step 3, analyze its capabilities in fulfilling the requirements from step 1.
5. To propose and implement the parallelization of selected DW CTU ETL part from step 2 using the tool chosen in step 4.
6. To evaluate the effectiveness of the proposed solution in meeting the requirements outlined in step 1, as well as its potential for managing the entire ETL process of the DW CTU in the future.

The theoretical part of this thesis aims to describe the concept and purposes of data warehousing and the current state of DW CTU. It also aims to analyze methods and concepts for parallel uploading that will be used in the implementation and include additional information concerning optimization of ETL processes.

The practical part of this thesis aims to propose and implement the solution, as well as describe the steps taken to complete this implementation.

Additionally, it aims to evaluate the solution in terms of overall performance improvement and platform/framework/software support community.

The contribution of this thesis is to improve the future development of DW by providing a better solution and to increase the flexibility of ETL processes by allowing greater configuration options, rather than relying solely on existing software.

Part I

Theoretical Background

Data Warehouse

A data warehouse (DW), or enterprise data warehouse (EDW), is a large, centralized repository that stores historical and up-to-date data from various sources within a company. Its primary purpose is to facilitate efficient data querying, reporting, and analysis for decision-making. By providing a consistent, accurate, and reliable perspective of data throughout an organization, data warehouses significantly contribute to business intelligence (BI), as they enable companies to gain insights from their data, make well-informed, data-driven decisions, and propel strategic growth.

The benefits of DW lie in their ability to assemble data from different sources into a single, cohesive format, offering organizations a complete picture of their data view. They make it possible to store and analyze massive amounts of historical data, helping businesses identify trends, recognize patterns, and predict future outcomes. ETL processes ensure data quality and consistency, while the DW's design for analytical processing accelerates query and reporting tasks. Moreover, DWs can easily grow and adapt to meet changing business needs and integrate new data sources and analytic tools. Strong data security and governance measures keep the data safe and compliant with regulations, protecting its confidentiality, integrity, and availability. In essence, DW form the foundation for effective data analysis, reporting, and decision-making, enabling businesses to uncover valuable insights, drive strategic development, and stay ahead in the competitive market.

1.1 Notions and definitions

In constructing a DW, a large-scale database that stores data from multiple sources, various technologies and concepts must be taken into account. To get a better understanding of what's involved in analysis, planning, and implementing the different aspects of a DW, it is helpful to explore a collection of related terminology:

- *Data Warehouse architecture:* The architecture of a DW refers to its structure and organization, including data storage, integration, and access. Two commonly used architectures are the Kimball and Inmon approaches. The Kimball architecture focuses on building data marts first and integrating them into a larger DW over time, while the Inmon architecture focuses on building a centralized DW first and creating data marts as needed. The choice of architecture depends on the needs and priorities of the organization and the nature of the data and business processes involved. [2, 3]
- *Data Modeling:* To develop a reliable data model for a DW, it is fundamental to define and analyze the data generated by an organization. By having a clear understanding of the data and the relationships between the, it becomes easier to design a suitable model. This approach helps to create an efficient and effective way to store and access data in the DW. [4]
- *Data Integration:* The process of combining data from different sources into a single, consistent format, which can be used in a DW. It includes stages such as data modelling, cleansing, ETL mapping, and transformation. [5]
- *ETL (Extract, Transform, Load):* ETL, or ETL process, is a foundation stone of the DW architecture and its activities. The process is responsible for extracting data from source systems, transforming it into a unified format, and loading it into the DW for storage and analysis. A well-designed ETL process can ensure that the integrated data is accurate, consistent and reliable. [2]
- *Business Intelligence (BI):* Business Intelligence involves a process that includes collecting and storing data, analyzing it to generate information, and presenting it to end-users for various use-cases. It helps to monitor business results and predict future outcomes accurately.

By diving deeper into these key topics, a reader can gain a better understanding of basic concepts of data warehousing and business intelligence. The following sections and chapters will provide valuable insights into these aspects.

1.2 Data Warehouse Architecture

The origins of data warehouses can be traced back to the mid-1970s, but they peaked in popularity in the late 1980s, when two founders William H. Inmon and Ralph Kimball contributed significantly to the development of the concept and architecture of data warehouses.

1.2.1 Inmon's Approach

William H. Inmon was the first to introduce the term 'data warehouse' and define the concept in his 1990 book 'Building the Data Warehouse'. [2] Often referred to as the 'father of data warehousing,' Inmon proposed the top-down, or data-driven, approach. According to him, a data warehouse is a 'subject-oriented, nonvolatile, integrated, time-variant collection of data in support of management's decisions.' [2] This approach promotes the development of a centralized data warehouse that integrates data from all sources across an organization. It also involves creating data marts as subsets of the main data warehouse to support specific departments or organization functions.

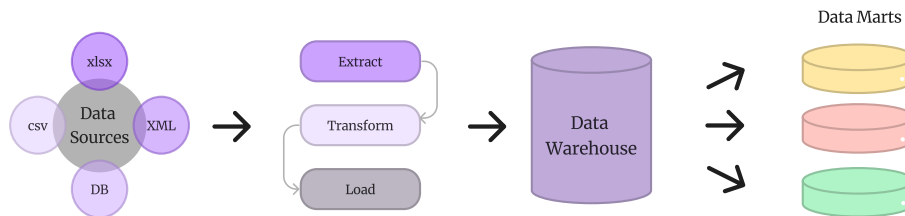


Figure 1.1: Inmon's top-down approach [6]

The *top-down approach* in this case refers to first defining and analyzing the data, followed by ETL processes and finally providing endpoint access for BI or other purposes [7, 8]. The key steps to building an Inmon-like data warehouse are:

1. Analyzing data from different sources and defining a data model to understand their data structures, relationships, and data quality. In Inmon's approach, the data warehouse is designed using a normalized, entity-relationship (ER) model to reduce data redundancy and maintain integrity. This approach results in a third normal form (3NF) schema design, which is more suitable for handling large-scale data integration and storage.
2. Data extraction, transformation, and loading (ETL). After designing the data model, ETL processes can begin to ensure data is in uniform format. The first step is to extract identified and analyzed data from sources. During the transformation process, data is cleansed, validated, and standardized to ensure quality and consistency. The normalized data is then loaded into the centralized data warehouse.
3. The data from DW is further divided into multiple data marts based on different departmental needs.

Advantages of Inmon's approach:

- One unified data source for the entire organization;
- Low data redundancy, simplifying ETL process;
- Flexible and tailored data marts: This approach enables the creation of separate data marts for different departments, providing tailored insights while maintaining a comprehensive overview;
- Greater adaptability to changes in business requirements or source data;
- Enterprise-wide reporting: This method can handle diverse reporting requirements across the organization;
- Lower maintenance costs: the overall maintenance costs are reduced due to the centralized and normalized data structure.

Disadvantages of Inmon's approach:

- Increased complexity: As more tables are added to the model, complexity increases, making it harder to manage and query;
- High initial costs: the primary setup of Inmon's approach can be costly and time-consuming, requiring high-skilled professionals which may not be feasible for all organizations.

1.2.2 Kimball's Approach

On the other hand, Ralph Kimball, another key figure in the history of data warehousing, introduced the *bottom-up approach*. In this method, the focus is on building data marts first, which are then combined to form a larger, integrated data warehouse. Kimball's approach emphasizes the importance of dimensional (denormalized) modeling and the star schema, which simplifies querying and enhances query performance. [9]

Kimball's approach focuses on designing and building data marts first, then integrating them into a larger data warehouse. The process involves the following steps:

1. Understanding and identifying the most critical business processes and needs. This helps in building a denormalized data model. The most commonly used data models are the star schema or the snowflake schema, where central fact tables are surrounded by dimensions tables.
2. Design and build ETL processes to populate the data warehouse, extracting data from the source systems and loading them into a denormalized data model.

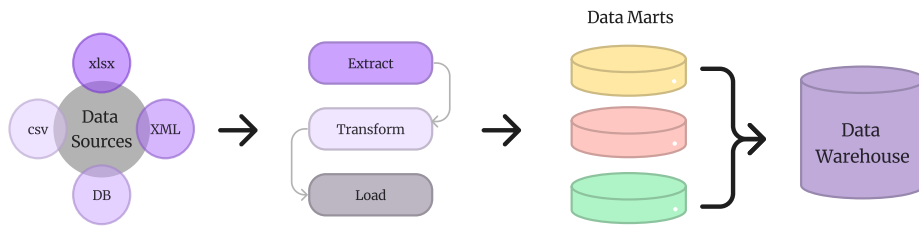


Figure 1.2: Kimball's bottom-up approach [6]

Advantages of Kimball's approach:

- Faster implementation and easier execution of the initial phase of data warehouse;
- Better performance for querying: the denormalized, dimensional model typically results in fewer joins and simpler queries, leading to better query performance for end-users;
- Focused on particular business processes, it requires less database storage and is simpler to handle.

Disadvantages of Kimball's approach:

- Potential for data redundancy: Since data is stored in a denormalized form in data marts, there may be some redundancy, which can lead to data inconsistencies and increased storage requirements;
- Increased ETL complexity: the ETL processes in Kimball's approach may be more complex due to the need to transform and aggregate data for the denormalized dimensional models;
- Less suited for enterprise-wide reporting: While Kimball's approach is well-suited for departmental or functional reporting, it may not be ideal for handling diverse, enterprise-wide reporting requirements, as it may require additional effort to integrate data from different data marts.

Summary

Both Inmon's and Kimball's architectures have played a crucial role in leading the development of data warehousing, and their methods continue to influence modern data warehouse implementation.

Choosing between Inmon's and Kimball's approaches depends on the specific needs and goals of the organization. Inmon's approach is best for a complete and consistent data warehouse, while Kimball's approach is ideal for

incremental development and achieving faster returns on investment. Eventually, the decision should be based on unique requirements, resources, and data strategy.

1.3 Data Modelling

Data modeling is a critical component in the development of a data warehouse, as it establishes the foundation for efficient data storage, retrieval, and analysis. A well-designed data model ensures that the data warehouse effectively supports an organization's requirements by presenting a consistent, accurate, and reliable depiction of its data. [4] This is usually achieved in three steps, with each step requiring a different level of abstraction:

1. *The conceptual data model* is a description of the data to be stored in a database that is independent of any specific technology. It is used by data modelers and business stakeholders to communicate their ideas. This type of model is usually presented as a diagram with additional documentation, providing a high-level view of the data entities and relationships between them.
2. *The logical model* translates the conceptual model into implementable structures within a database management system (DBMS). It typically specifies tables and columns forming the foundation of relational databases that support data warehousing. This step requires careful consideration of technical attributes – e.g. historization concerning columns – to ensure data accuracy and consistency.
3. *The physical data model* defines the physical storage and access mechanisms required for data to be stored in a DBMS, along with the specific details about the data such as field lengths, types, and relationships between the data elements. It uses tables and columns to present the data structure and is designed to ensure data integrity, consistency, and high performance while minimizing storage requirements.

1.3.1 Data Modeling Approaches

Most modern data warehouses are build on *relational data models*. However, alternative models, such as *hierarchical data models*, exist. They are primarily used to organize data in a tree-like structure with parent-child relationship between data elements.

Turning back to relational data model, it is important to specify its basic approaches:

- *The entity-relationship (ER) model* is a data modeling approach that incorporates significant semantic information from the real world. It

introduces a diagrammatic technique as a valuable tool for database design, along with implications for data integrity, information retrieval, and data manipulation. The model also serves as a foundation for unifying different views of data and resolving semantic ambiguities found in other models. [10]

- *Object-oriented data modeling* is a database design approach that supports complex and multimedia objects. The model should support object structures and interrelationships, behaviors, and dynamic constraints. In addition, it helps OODBMS to provide persistent storage and schema definition for objects, a query language, access optimization, concurrency control, authorization mechanisms, and recovery. [11]
- *(Multi-)Dimensional data modeling* is a data modeling approach that is commonly used in data warehousing (Kimball's methodology). It organizes data into dimensions and measures, where dimensions represent the descriptive attributes of the data and measures represent the numerical or quantitative data. The main objective of this approach is to facilitate complex queries and analysis of large volumes of data. In dimensional data modeling, data is organized into a star schema or a snowflake schema, which are optimized for efficient data retrieval and aggregation. [4]

Two data modeling techniques that are relevant in a data warehousing environment are ER data modeling and dimensional data modeling. [5] While ER models focus on efficient storage, dimensional models increase redundancy to simplify information reporting and retrieval, typically employed across Online Analytical Processing (OLAP) systems.

1.4 Data Integration

Data integration is an essential process for organizations building a data warehouse. It involves combining data from various sources, such as databases, applications, and file systems, into a unified and consistent view. This process aims to provide a complete and correct view of an organization's data to enable reporting, analysis, and decision-making. Nevertheless, challenges arise when attempting to integrate data from external applications or different providers, resulting in uncontrollable data replication and increased storage and update expenses.

The principal difficulty in data integration lies in addressing inconsistencies in data formats, structures, and the like while also ensuring data quality, dependability, and comprehensiveness. Frequently, organizations create and maintain data models for individual applications and an enterprise-wide data model to offer a strategy for integration. Nevertheless, reaching a high level

of data integration continues to be an important challenge for many organizations.

Data warehouses need to be more than just a gathering of records from source systems; they should be databases that ‘make sense’ on their own. This often involves defining target tables and translating, reformatting, or summarizing data to serve data marts. These constraints can complicate the data modeling and integration processes. This is exactly the point, where ETL tools prove useful in automating these processes. Efficient and properly implemented ETL techniques enable continuous and high-quality data integration. [12, 7]

Data integration systems include a global schema and multiple sources containing the actual data. The mapping between sources and the global schema is required step and can be achieved through two basic approaches: global-as-view and local-as-view. Moreover, the data integration system must not only combine data from various sources but also be able to handle queries in terms of the global schema. This requires the system to restructure the query into terms that the sources can understand and process. Inconsistencies among sources can arise, and data integration systems must address them through transformation and cleaning procedures. [13, 14]

In conclusion, proper data integration is necessary for the successful data warehouse. Careful planning and execution ensure that the data warehouse meets the organization’s requirements. The most challenging tasks to achieve an efficient and dependable data warehouse are fixing data inconsistencies, mapping sources to global schema, and processing queries.

1.5 Business Intelligence

Effective decision-making is absolutely essential for the success of any organization or business. However, this process can be complicated, as there are various factors to consider including available resources customer preferences, competitor analysis, etc. Managing these diverse data becomes difficult when it is stored across multiple systems within an organization. This is where data warehouses come into play, combining these data sources into a single, rich repository with consistent and trustworthy information. DW supply reliable data to support Business Intelligence (BI) platforms and software in evaluating the contemporary state of an organization and giving valuable basis for future growth.

BI plays a crucial role in ensuring a company’s success. It serves as both a process and a product, helping organizations to develop and apply critical information to enhance their growth and competitiveness in the global economy. BI systems combine operational data and analytical tools, providing complex and valuable insights for decision-makers. [15]

The emergence of data warehouses, advances in data cleansing, and improvements in hardware and software capabilities, along with the rise of web architecture, have all contributed to a richer BI environment. BI systems are extensively used across various industries, such as manufacturing, retail, financial services, transportation, telecommunications, utilities, and health-care. [15]

To conclude, the importance of BI in modern organizations cannot be overestimated. With data warehouses serving as a vital component for BI systems, they facilitate decision-making processes. It is essential for researchers and practitioners to persistently investigate and improve BI systems and approaches, ensuring that organizations can make well-informed and timely decisions. As a final point, this contributes to their success in a competitive global marketplace.

Conclusion

This chapter has provided a thorough overview of the core components and purposes of data warehouses, laying the foundation for a deeper exploration of the main issues addressed in this thesis. The following chapter delves into the ETL processes, their challenges, and methods for optimization. Although ETL is an key part of data warehousing, it is treated separately, as it represents the primary focus of this thesis. By building upon the knowledge gained in this chapter, readers will be better prepared to grasp the details of ETL processes and their problematics.

Extract, Transform, Load

This chapter will dive into the heart of data warehousing – the ETL processes. ETL, which stands for Extraction-Transformation-Loading, refers to the software processes that facilitate the population of data warehouses. In addition to describing each step within the ETL abbreviation, this chapter will cover the challenges faced by ETL developers and potential solutions. It will also explore techniques for ETL optimization and parallelization, including methods of orchestration.

2.1 ETL Process Stages

To better understand the details of ETL processes, it is beneficial to examine the individual steps involved. ETL processes have the responsibility of *extracting the required data* from the sources and transporting it to a specialized area (commonly called Data Staging Area, or DSA) within the data warehouse for processing; *transforming the source data* and computing new values (and, potentially records) to integrate with the structure of the DW relation that it targets, isolating and cleansing problematic tuples to ensure compliance with business rules and database constraints; and *loading the cleansed and transformed data* into the appropriate relation within the Integrated Data Layer (IDL) of the data warehouse, while refreshing its related indexes and materialized views. [16]

2.1.1 Extract

Once the appropriate data warehouse architecture and modeling have been established and configured, and the logical data map (LDM) has been defined, it is time to populate the data warehouse with data. The first step in this process is to extract data from various sources for further processing and integration.

2. EXTRACT, TRANSFORM, LOAD

Put simply, *extract* means ‘the process of selecting data from one environment and transporting it to another environment.’ [7] The data sources from which data can be extracted may cover a wide range of formats, including flat files (such as CSV, XML, JSON, etc.), as well as relational databases (such as Oracle, PostgreSQL, MySQL, etc.). While the extraction of data from flat files may not pose a significant challenge, establishing connections with relational databases can be a complex task.

Open Database Connectivity (ODBC) can be a helpful solution that ensures access to databases from multiple applications without needing to recode and recompile the application layer when the underlying database changes. ODBC drivers are available for almost every DBMS on any platform and can also be used to access flat files. [3]

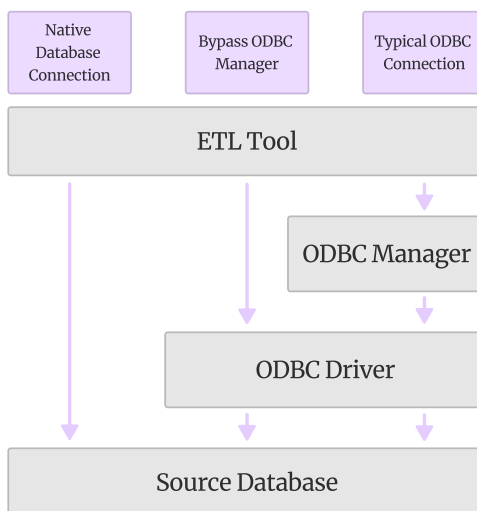


Figure 2.1: The topology of ODBC in the ETL process [3]

Despite its flexibility, ODBC comes with a performance cost due to the added layers of processing and data passing involved in data manipulation. When using ODBC in the ETL process, two layers are added between the ETL system and the underlying database: the ODBC manager and the ODBC driver (Figure 2.1). The ODBC manager maintains the connection between the application and the ODBC driver, while the ODBC driver translates ODBC SQL to the native SQL of the underlying database. Although ODBC has improved significantly over the years, it is recommended to use native database drivers for optimal performance and functionality. However, ODBC can still be a valuable tool for accessing data from sources that are otherwise difficult to extract. [3]

After successfully establishing a connection to the source database, the next step is to transfer the source data to the staging area. The staging area is a temporary storage location where data is collected, validated, cleansed, and transformed before being loaded into the target data warehouse. The following transformation step covers all these tasks.

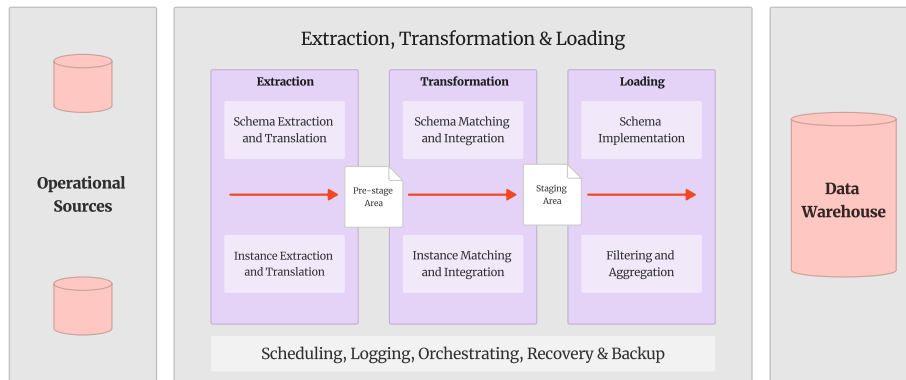


Figure 2.2: Overview of entire ETL process [17]

2.1.2 Transform

While the extraction and loading steps are typically considered straightforward, it is the transformation step that adds value to the entire ETL system. [3] ‘Transformation is a data integration function that modifies existing data or creates new data through functions such as calculations and aggregations.’ [18] In addition, the transformation step involves a variety of tasks, including data validation, cleaning, data type and value converting, handling missing values and data enrichment, computing technical values, (de)normalization, aggregating and filtering data, among others. All of these ensure that in production target data warehouse data will be reliable, consistent and accurate. This section will delve into the most significant processes from the ones mentioned above, providing detailed explanations.

Data Cleaning

Data cleaning, or data cleansing, refers to the process of identifying and correcting corrupt, inaccurate, or incomplete records within a database or record set. This step involves detecting and addressing data that is irrelevant, incorrect, or incomplete, and modifying, deleting, or replacing that data as needed. [19] Some researches include data validation to this step, which guarantees compliance with predefined business rules and constraints. In addition, data cleaning techniques can be applied to various data types, including structured

and unstructured data, in order to improve overall data quality. However it is important to note, that while data cleaning is inherently related to the ETL process, it is considered a distinct field of its own, with a primary focus on record matching and addressing challenges associated with textual attributes. [16, 20]

Data Conversion and Formatting

During the transformation step of the ETL process, data conversion and formatting are essential for ensuring accurate and consistent data representation in the target data warehouse. When determining data types, careful consideration must be given to the trade-offs between query efficiency and the category of data type used. For example, numeric data not intended for calculations, such as Social Security numbers, can be stored as either VarChar or Integer. However, integer-defined columns offer greater query efficiency compared to VarChar. [18]

Various challenges may arise in applying format masks, including different value representations (e.g., for sex: 'Male', 'M', '1') and interpretations of values (e.g., date formats: American 'mm/dd/yy' vs. European 'dd/mm/yy'). Other value-level issues involve surrogate key management, substituting constants, setting values to NULL or DEFAULT based on conditions, and utilizing common SQL operators like UPPER, TRUNC, and SUBSTR. Addressing these challenges is crucial for ensuring a smooth and effective data integration process. [21]

Composite Technical Transformations

In a data warehouse, which is an integrated storage system supporting data historization, challenges occur due to diverse production keys across data sources and processes that maintain the proper functioning of historical data storage. Managing different key systems from various data sources can be complex, as mapping data from one source to another, representing the same real-world object, is not always straightforward. To address this issue, surrogate keys are generated, replacing production keys to optimize performance and achieve semantic consistency. Surrogate keys resolve potential conflicts among data sources by allowing for globally unique, reconciled keys in the table. [3, 21]

To maintain the integrity of the historical data within the data warehouse, two preferable solutions are purposed: Slowly Changing Dimensions (SCD) and Change Data Capture (CDC). SCDs primarily manage historical changes in dimension tables within a data warehouse by providing different ways to store and track changes. SCD has three major types – Type 1, Type 2, and Type 3 – that determine how new and old values are managed. Type 1 overwrites old values and is used when history tracking is unnecessary. Type 2

retains history by creating new tuples with new surrogate keys, while Type 3 modifies existing tuples by keeping both old and new values, making it useful for history tracking. [21]

Change Data Capture (CDC) is a technique for capturing changes in the source data and efficiently applying these changes to the target data warehouse. CDC tracks and identifies insertions, updates, and deletions in the source data, and then it incrementally applies these changes to the target system. This technique reduces the need for full data loads and minimizes the latency in data updates, thereby improving the overall performance and efficiency of the ETL process. [18, 22]

Capturing changes in records can be time-consuming and performance-intensive, requiring a comparison of the previous and updated states of database records. Efficiency can be achieved through several techniques, such as calculating row hashes for comparison with the business key, using advanced frameworks (e.g., Delta Framework [23]), or applying CDC pattern.

Implementing SCD for a data warehouse implies that additional columns may be needed during the transformation phase. These columns may include status, version, hash, update timestamp, etc. [23]

Lookups and Filtering

Lookups involve the process of combining data fields from records with values from reference tables, creating an enriched dataset. This process allows tables to be expanded with attributes from other tables, providing a more comprehensive view of the data, reducing calculations during queries on target tables, and maintaining data integrity. [24, 18]

The final step in the entire process is filtering columns to be loaded into the target layer of a data warehouse. Some attributes from fetched tables originating from different sources may be duplicated or redundant. Therefore, it is considered good practice to exclude certain information and populate the data warehouse only with valuable and relevant data.

2.1.3 Load

Loading to the target database of a data warehouse represents the final step in the entire ETL process. At this stage, data is transformed and ready for uploading, as it has been cleaned, made consistent, filtered, and equipped with all necessary technical attributes for subsequent loading. Typically, loading is closely related to the implementation of Slowly Changing Dimensions (SCD), particularly Type 2. When a data warehouse is notified of a required change in an existing dimension record, a new dimension record is issued for the corresponding object at the moment of the change, rather than overwriting the existing record. Previously created technical attributes, such as status and update timestamp, support the 'perfectly partitioned history' scenario,

which is later extracted to additional target columns like version, start date, end date, and last update. This approach provides a clear overview of when and how the data changed. [3]

Furthermore, the loading stage addresses another challenge: loading techniques. [5] There are several types:

- Initial load: the first extraction of source data;
- Incremental load for each change: updated data is loaded each changed, tracked by DBMS log capture, triggered capture, or application-assisted capture;
- Periodic incremental load: changes are captured and loaded periodically (e.g., hourly, daily, weekly, monthly, etc.), which involves change tracking by timestamp-based capture or file comparison capture.

2.2 Parallel Optimization of ETL Processes

In today's data-driven world, end-users rely heavily on the information provided by data warehouses for purposes ranging from business intelligence reports to up-to-date web endpoints. As the volume of data continues to grow, the need for efficient data processing becomes increasingly important.

Data warehouses depend on ETL processes and their techniques, which are often the most time and resource-intensive aspects of data warehousing. [25] Traditionally, ETL process efficiency encounters several challenges, including:

- *Large volumes of data*: The vast amounts of operational data present significant data management problems across all three phases of the ETL process;
- *Data quality*: Data is not always clean, requiring cleansing during the ETL process;
- *Evolution of data stores*: Changes in the sources and the data warehouse can lead to frequent maintenance operations;
- *Performance issues*: The ETL process must be completed within a specific time window, demanding optimization of execution time. Any process failures must also be addressed within the established time windows to maintain data quality and efficiency.

The extraction step in ETL processes is typically straightforward, while the transformation and loading steps demand more attention. Numerous researchers have developed techniques and methods to improve ETL process efficiency, but several challenges still persist. Addressing these issues often involves revisiting the entire data warehouse design and employing advanced

techniques for enhanced data processing and storage, such as incremental data extraction, indexing, and partitioning at the target database. Parallel processing is another valuable approach that can significantly improve ETL performance.

2.2.1 Parallel Processing of ETL

Parallel computing, also known as parallel processing, is a computational approach where multiple calculations or processes occur simultaneously. [26] This method has been widely used in high-performance computing and has emerged as the leading paradigm in computer architecture, mainly through multi-core processors. [27] Parallel computing can be categorized into various forms, including bit-level, instruction-level, data, and task parallelism. In this section, the latter two forms – *data* and *task parallelism* – will be discussed.

Data Parallelism

Data parallelism is a type of parallelism that concentrates on improving the processing flow and structure of information, rather than depending on concurrent processes or tasks. The goal of data parallelism is to enhance processing throughput by breaking down the dataset into concurrent processing streams, each carrying out the same set of operations. This approach, known as domain decomposition or ‘data parallelism,’ involves assigning a single portion of data to a single process, with data portions being roughly equal in size. [28]

When portions require varying amounts of processing time, performance may be limited by the slowest process. In such cases, the issue can be moderated by dividing the data into a larger number of smaller portions. Processes then take on another portion upon completing the previous one, with faster processes being assigned more portions. This is usually achieved in SIMD mode (Single Instruction, Multiple Data mode) and can involve either a single controller directing parallel data operations or multiple threads operating similarly on individual compute nodes (SPMD). [29]

Another example of data parallelism is the parallelization of a loop without loop-carried dependencies, where processes execute the same loop body but for different loop indices and, as a result, different data. This approach centers on distributing data across various nodes in the parallel execution environment, enabling simultaneous subcomputations on the distributed data across different compute nodes. [30] Data parallelism represents a more refined form of parallelism, as it enhances performance by applying the same small set of tasks iteratively across multiple data streams.

Task Parallelism

In contrast, task parallelism emphasizes the distribution of parallel execution threads among multiple computing nodes. These threads might execute the same or different tasks. They communicate with each other through shared memory or explicit communication messages, depending on the parallel algorithm. In a task-parallel system, threads can perform entirely different tasks while coordinating to solve a specific problem. In simpler cases, all threads execute the same program, with variations in task responsibility determined by their node-IDs. [29]

Task-parallel algorithms often adopt the Master-Worker model, where a single master and multiple workers are involved. The master assigns computations to different workers based on scheduling rules and task allocation strategies. [29] In task parallelism or functional decomposition, processes are assigned to individual pieces of code, which work on the same data. For example, computing the average and standard deviation on the same data can be executed by separate processes in task parallelism. Another example is the parallelization of a loop containing an if-then-else construct, resulting in the execution of different code during different iterations. [30]

Conclusion

In conclusion, this chapter provided an in-depth exploration of the Extract, Transform, and Load (ETL) process, a key component in data warehousing. The three primary stages of the ETL process were examined, including the extraction of data from various sources, the transformation of data through cleaning, conversion, formatting, technical transformations, lookups, and filtering, and finally, the loading of transformed data into the target data warehouse.

Furthermore, the chapter delved into the parallel optimization of ETL processes as a means of improving efficiency and performance. It highlighted the importance of parallel processing in ETL and discussed two main types of parallelism: data parallelism and task parallelism. Data parallelism focuses on enhancing the processing flow and structure of information by decomposing the dataset into concurrent processing streams, while task parallelism emphasizes the distribution of parallel execution threads across multiple computing nodes. Both data and task parallelism, especially when combined, can significantly improve the performance of ETL processes, ensuring that data warehouses remain up-to-date and capable of handling the growing demands of modern data-driven environments.

By understanding the issues of the ETL process and leveraging parallel optimization techniques, data professionals can develop more efficient and effective data warehousing systems, ultimately enabling organizations to make better-informed decisions based on reliable and accurate information. [31]

Part II

Practical Implementation

Current State Analysis of the CTU Data Warehouse

While theoretical principles provide a foundational understanding of how a data warehouse should ideally function, the reality often diverges due to an organization's unique requirements, increasing data flow (which may cause the current solution to lose relevance and effectiveness over time), and evolving techniques that establish new standards. Adaptation and progress are essential, particularly when there are opportunities and areas for growth and improvement. This chapter will explore the design and overall state of the current CTU data warehouse, highlighting its strengths, aspects that may benefit from reconsideration, and potential areas for enhancement.

3.1 CTU Data Warehouse Architecture

The architecture of the CTU Data Warehouse follows Inmon's model, which entails a centralized storage system that integrates data from multiple Information Systems (IS) to serve as a single source for creating data marts and addressing Business Intelligence (BI) reporting needs. At present, the CTU DW integrates seven data sources, including Usermap, KOS (Study Information System / Komponenta Studium), Anketa, EZOP, Uvazkostroji, Grades, and Projects (as well as several flat-file data exports). These systems utilize either PostgreSQL or Oracle as their Database Management Systems (DBMS).

The entire CTU DW, encompassing all three layers – *stage*, *target*, and *access* – is stored in PostgreSQL. *The stage layer* (Data Staging Area), is a database that houses raw data extracted from the sources. This layer also contains PL/pgSQL procedures employed during the transformation processes. Once the data is transformed, it is loaded from the stage layer to *the target layer* (Integrated Data Layer), which is a centralized integrated database utilizing the third normal form (3NF).

3. CURRENT STATE ANALYSIS OF THE CTU DATA WAREHOUSE

The *access layer* is closely connected to the target layer and extracts data from it to populate data marts for end users. The primary goal of this arrangement is to facilitate efficient data management and retrieval for various analytical and reporting purposes within the organization. [32]

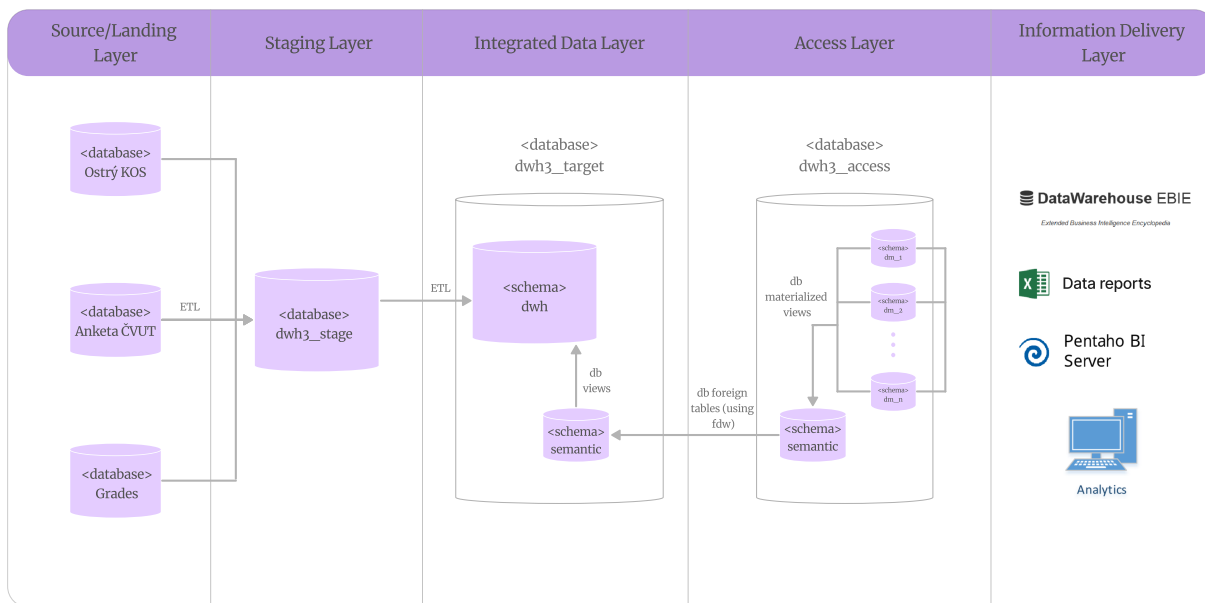


Figure 3.1: CTU DW 3.0 Architecture [32]

3.1.1 Staging Layer

The staging layer, also referred to as the stage, exists as a separate database within the CTU Data Warehouse and is named `dwh3_stage`. This layer integrates data from all seven source systems and organizes it into three groups, corresponding to various stages of transformation. These groups are:

- **ps.:** This prefix is used for the schema names of each exported domain. In this context, PS stands for 'pre-stage'. These schemas contain raw data loaded from sources without any transformations applied. As a result, the `dwh3_stage` (staging database) contains several schemas with names such as `ps_kos`, `ps_grades`, `ps_ezop`, and so on, which hold table copies from the source databases.
- **psc.:** This prefix denotes schemas that store cleansed data (PSC represents pre-stage clean). Data is loaded here after being processed through various transformations, such as removing data inconsistencies, duplicates of keys, unnecessary columns, etc. Currently, these transformations are applied using PL/pgSQL procedures, which are triggered by

PDI (the data integration tool currently in use). It is important to note that not all domains are stored here, as some source systems are reliable and their data is consistent and meets the requirements (e.g., there are no `psc_grades` or `psc_anketa` tables). The data in this stage is cleaned and ready for further processing.

- **si_**: SI stands for stage increment, and this name partially explains the data transformation process at this stage. Transformations applied include computing state flags for new, modified, and deleted data. This transformation is discussed in more detail in the following section (3.2). In brief, it compares newly loaded data with the data since the last refreshment of the data warehouse.

3.1.2 Target Layer

The target layer in the CTU DW, named `dwh3_target`, serves as the central repository where data is loaded after the final transformations are processed. This layer is organized into several schemas, with the most important ones for this thesis being:

- *ciselniky* (English: code lists) contains tables with codes for reporting. These tables usually remain unchanged, so they are not updated regularly;
- *dwh* functions as the integrated data storage, where transformed data from the stage layer is uploaded. At present, it houses 185 tables designed to closely adhere to the third normal form (3NF). This schema not only includes data from all provided sources but also historical data. The structure of this schema is discussed in detail in Section 3.2;
- *semantic* is a schema used for creating and storing data views, primarily to combine data from multiple target tables for improved comprehensibility. Once views are prepared, they are imported into the access layer for endpoint use. By employing the semantic agent user and PostgreSQL Foreign Data Wrapper technology, views from the `dwh3_target`'s semantic schema are connected to the `dwh3_access`'s semantic schema as foreign tables using the `IMPORT FOREIGN SCHEMA` command. [33]

3.1.3 Access Layer

The third and final layer of the CTU Data Warehouse is formed by the `dwh3_access` database, which serves as the primary point of access for users. Users can connect to this layer either directly through SQL queries or indirectly via reporting tools. Unlike the stage and target layers, this layer contains schemas with materialized views that store aggregated data, offering

a more user-friendly representation of the underlying information. Materialized views offer rapid access to data by eliminating the need to repeatedly execute complex queries.

3.2 Current ETL processes of CTU Data Warehouse

In the context of the CTU Data Warehouse, the ETL processes play a crucial role in managing the data flow between various databases and schemas. These processes are responsible for extracting, transforming, and loading data to ensure smooth operation and data consistency. To provide a clear understanding of the CTU Data Warehouse ETL processes, this section will serve as an introduction to the primary transformations and tools employed in the system.

The entire data flow between these databases and schemas is facilitated by ETL processes. These processes are either implemented using PL/pgSQL scripts or tools' transformations; however, both types are ultimately triggered or processed by PDI, the data integration tool currently employed in the system. This section will explore the essential transformations necessary for the proper functioning of the CTU Data Warehouse.

Currently, the Pentaho Data Integration (PDI) tool holds the position of the master-chief in all ETL processes within CTU Data Warehouse. PDI, also known as Kettle, is an open-source data integration and ETL (Extract, Transform, Load) tool developed in Java for processing and transforming large amounts of data. Consequently, it is platform-independent and compatible with any operating system that supports Java Virtual Machine (JVM). PDI offers a wide range of features, including data extraction from various sources, data transformation, and data loading into target systems such as databases or data warehouses.

In Figure 3.2, the entire ETL routine is presented using the Kettle job named 'J_DWH_routine'. Generally, all tasks in PDI are either jobs or transformations. The main distinction is that transformations focus on data processing, while jobs concentrate on workflow orchestration and coordination of multiple data processing steps. Due to this separation, all PDI tasks are prefixed with 'J_' (job) or 'T_' (transformation). The 'J_LOAD_PRE_STAGE' box is responsible for data extraction, including certain transformations, as well as the 'J_MAKE_INCREMENT' box. The final loading process is ensured by the 'J_IDL_LOAD' box.

3.2. Current ETL processes of CTU Data Warehouse

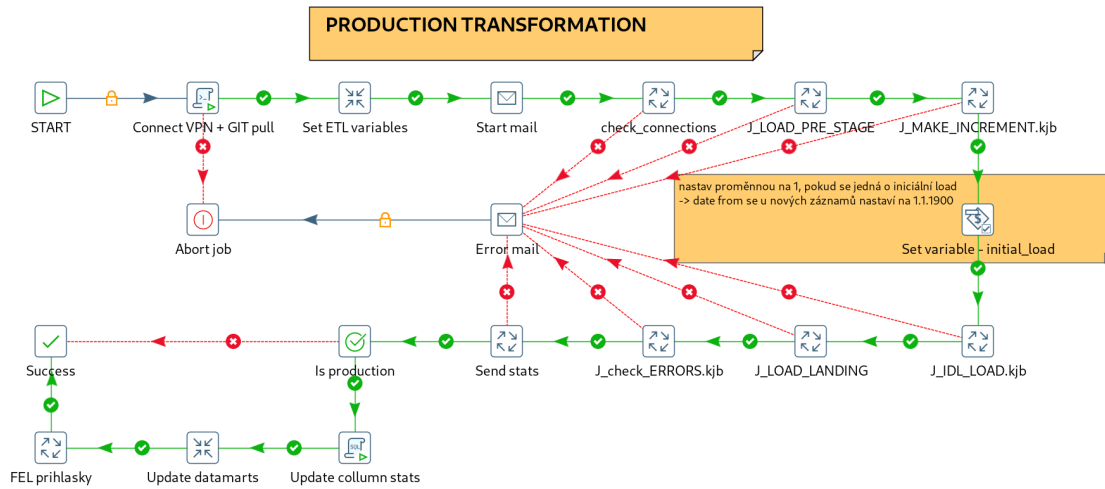


Figure 3.2: Current ETL processes in PDI [34]

3.2.1 Source to Pre-Stage, or Export

The ‘J_LOAD_PRE_STAGE’ box hides the loading of all data source systems accessible to CTU Data Warehouse administrators. As previously mentioned, it extracts data from seven ISs. The extraction process occur *sequentially*, meaning each domain is loaded one after another. A more detailed view of the extraction from source and subsequent loading into the pre-stage area is illustrated in Figure 3.3

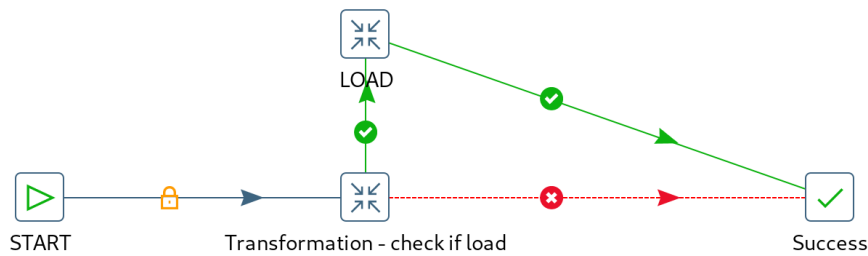


Figure 3.3: ‘TEKOSOBY_CHECK_JOB’ job in PDI [34]

Most of the processes are similar to one another, so the analysis will focus on describing the target table ‘t_osob_osoba.kos_stara’, which is loaded and transformed from KOS (the name of the original table is ‘TEKOSOBY’).

The first step in the graph is ‘Transformation - check if load’ utilizes PL/pgSQL function `fc_should_table_be_loaded_to_ps`. This function relies

3. CURRENT STATE ANALYSIS OF THE CTU DATA WAREHOUSE

on a metadata table to determine whether or not to load the table into the pre-stage area. The following ‘LOAD’ box extracts data from the source, appends an MD5 hash (calculated from the row’s values with PDI function ‘Add a checksum’) to each row, and loads the resulting data (with PDI PostgreSQL Bulk Loader) into the pre-stage area within the corresponding schema (in this case, it would be ‘ps_kos’). MD5 hash, along with the business key, helps catch changes. Although it is not collision-resistant, it still proves valuable and efficient within this context. [32]

3.2.2 Pre-Stage to Stage-Increment, or Transform

The data transferring from pre-stage area to stage increment area is done in the ‘J_MAKE_INCREMENT’ box shown in the Figure 3.2. The content of this step is illustrated in Figure

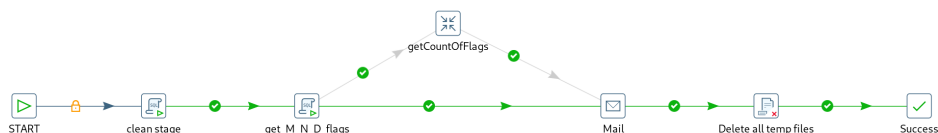


Figure 3.4: ‘J_MAKE_INCREMENT’ job in PDI [34]

In the first step, PDI triggers PL/pgSQL script using the ‘Execute SQL script’ function, named ‘clean stage’ in Figure 3.4. The script is as follows:

```
VACUUM;  
ANALYZE;  
  
BEGIN;  
SELECT public.create_clean_pre_stage();  
SELECT public.inc_clear_state_flag();  
COMMIT;
```

Listing 3.1: Clean Stage Box Script

`VACUUM` reclaims unused space, and `ANALYZE` updates statistics about tables to improve query performance. Subsequently, it executes predefined functions (specified in `public.create_clean_pre_stage`) to clean and filter data. These functions additionally check whether a record is deleted in the source system, ensure business keys are free of duplicates, create indices for faster querying, and load the data into a pre-stage clean area, ensuring its cleanliness and accuracy for further processing. As previously mentioned, data from some sources is reliable enough to skip this step. The `inc_clear_state_flag` function removes the state flags (‘N’ – New record, ‘M’ – Modified record, or

‘D’ – Deleted record) that were computed during the previous data warehouse refreshment to avoid interfering with new, upcoming changes in records. Each function is executed in sequence, as before.

The next step is crucial, as it detects changes and assigns an ‘N’, ‘M’, or ‘D’ flag/state to the corresponding case. PDI triggers the SQL script below:

```
BEGIN;  
  
SELECT inc_find_modified_in_pre_stage();  
SELECT inc_find_new_in_pre_stage();  
SELECT inc_find_deleted_in_pre_stage();  
  
COMMIT;
```

Listing 3.2: get_M_N_D_flags Box Script

Changes are captured by applying CDC technique described in the theoretical part of this thesis, in which data is downloaded into a pre-stage area, changes are identified relative to the previous state, and altered records are marked accordingly. Three types of changes are monitored: new records (marked with ‘N’), modified records (marked with ‘M’), and deleted records (marked with ‘D’). Changes are identified using two parameters: an MD5 hash of the record and a predetermined business key. During data loading, an MD5 hash is created for each record and added as a text column.

After data is loaded into the stage database, change detection is performed using three database procedures mentioned in Listing 3.2. The number of modified records is found, and these counts are sent to the administrator via email along with the transformation log. To find these records, dynamic SQL is used, sequentially processing each pre-stage table, executing prepared commands, and propagating changes to the stage-increment with appropriate change markings.

To find new, deleted, and modified records, database joins are utilized with business keys. For each table, the process iterates through and inserts records into the stage-increment that meet certain conditions. [32]

- **New records** are those present in the pre-stage but not in the stage-increment;
- **Deleted records** are identified as those present in the stage-increment but not in pre-stage. The `inc_find_deleted_in_pre_stage` function also updates their state to ‘D’ (deleted) and active to 0;
- **Modified records** are identified by comparing hashes and selecting those not in the stage-increment, then comparing business keys of these records and the stage-increment to exclude new records.

3.2.3 Stage-Increment to Target, or Load

The final process of CTU Data Warehouse ETL is loading the changed data from the stage-increment area into the target. Continuing the description of the process using the ‘t_osob_osoba.kos_stara’ table from KOS, the concluding stage is presented in Figure 3.5.

The process begins with the ‘Table input’ step, where data is extracted from the stage-increment area using an SQL query with a specific constraint: `WHERE si_kos.tekosoby IS NOT NULL`. This ensures that only changed records are loaded. Before proceeding to ‘Select values’ step, the last transformations are applied. These may include database lookup, row filtering, value replacement, or adjusting strings, as in this case. In the ‘Select values’ step, data types are fixed, column names are changed, and fields are chosen from the input data. Then data is then separated by their state (‘N’, ‘M’, or ‘D’). Data with each state is loaded differently. Loading new data is the simplest, as it only requires inserting values using PDI Bulk Loader. ‘Loading’ deleted records is slightly more complex and is implemented using a different technique. PDI executes the following SQL script:

```
UPDATE dwh.{table_name}
SET date_to = CURRENT_TIMESTAMP
WHERE osoba_peridno_bk = '{business_key}'
AND (date_to = '2199-12-31 00:00:00');
```

Listing 3.3: SQL Script for Loading Deleted Rows

This script essentially finds the last version of the deleted record and sets the `date_to` column to `CURRENT_TIMESTAMP`, signifying that the validity of the record expired at the moment of loading. The script is executed for one record at a time, processing all deleted rows individually.

The last branch of the loading process is updating and inserting modified records. This is the most complex and time-consuming operation. It is also implemented using an SQL script, which calls the `public.inc_historize_case_m` procedure. The procedure takes four parameters: one row, with values concatenated using the ‘\x1f’ separator; the table name to load into; and two values from stage-increment table: `active` and `last_update`. In simple terms, it terminates the validity of the last version of the modified record (similarly to the deleted case) and inserts a valid record with an incremented version number. Although every table is loaded sequentially, this step has been optimized to execute in 30 parallel processes.

3.3. Orchestration and Logging

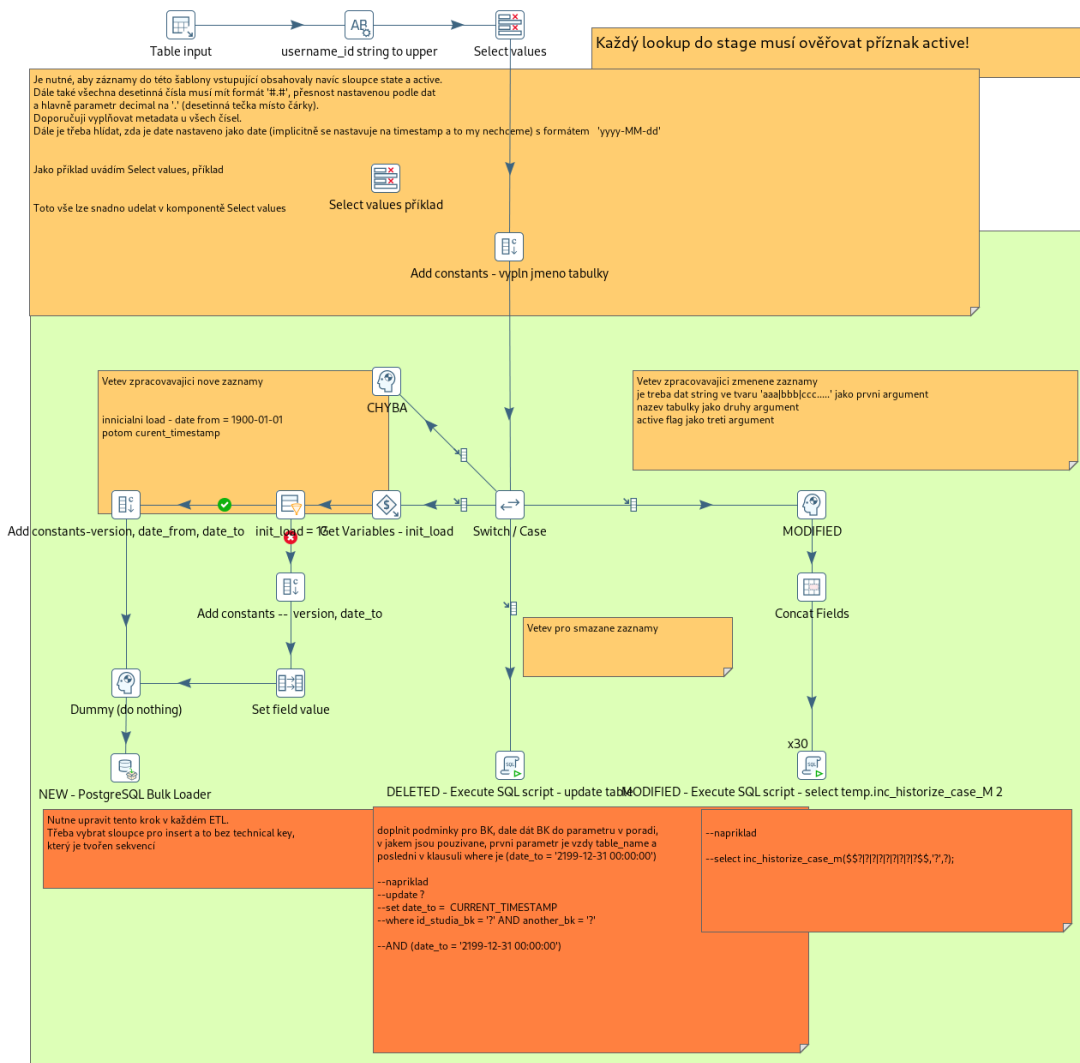


Figure 3.5: 't_osob_osoba.kos.stara' transformation in PDI [34]

3.3 Orchestration and Logging

In Pentaho Data Integration (PDI), processes can be orchestrated using Jobs and Transformations. Jobs manage high-level control flow and coordinate multiple data processing steps, while Transformations focus on detailed data manipulation tasks. Jobs and Transformations can be organized hierarchically, with Jobs calling other Jobs or Transformations, and Transformations calling sub-transformations. This hierarchical organization allows users to create complex workflows by breaking them down into smaller, manageable components.

PDI provides built-in logging capabilities to track and monitor the progress of Jobs and Transformations. Logging settings can be configured to capture various levels of detail, ranging from basic execution statistics to fine-grained debugging information. The logs can be stored in text files, databases, or sent to remote log servers for centralized management.

Conclusion

Throughout the analysis of the CTU Data Warehouse and, primarily, its ETL processes, several hidden issues were discovered. Among them are:

- Redundant state counting: Some tables in the stage area are used for lookups only, making it unnecessary to find changes in them. Cleaned data from sources is sufficient;
- False-positive modified flags: In cases where not all columns are uploaded to the target, MD5 hash is still computed for unfiltered data. If a value (that is not integrated into the target) changes, the entire row is marked as modified, but the filtered row remains the same;
- Some transformations are implemented with errors.

Addressing these issues will lead to more efficient processing and improved data quality.

The current ETL processes are governed by Pentaho Data Integration (PDI). PDI is a powerful tool for working with data migration, offering rapid and straightforward design as well as versatility. Despite its many advantages, PDI has some drawbacks, such as limited support, optimization options, upgrade migration difficulties, poor scalability, occasional random behavior, and issues with proper functioning on Linux OS systems (based on the author's personal experience). To fulfill the requirements for parallelizing ETL processes, author's colleague has been tasked with developing a custom extension to address table dependencies and other challenges.

This thesis aims to explore alternative tools for migrating the ETL processes and addressing the identified problems in a different manner. The requirements for such a tool are structured and analyzed in the following chapter.

Requirements Analysis and Specification

In this chapter, the focus will be on the Requirements Analysis and Specification for the implementation of parallel ETL processes in the CTU Data Warehouse. This work is part of a collaborative effort between two bachelor theses, both focusing on improving the current ETL solution for CTU DW. One thesis, ‘Parallelization of DW CTU ETL processes in the Pentaho tool’, FIT CTU, (sk.: ‘Paralelizácia ETL procesov DW ČVUT s využitím nástroja Pentaho’, FIT ČVUT) authored by Adam Marhefka, is dedicated to improving the existing solution, while the other thesis, to which this chapter belongs, is centered on investigating alternative approaches to implementing parallel ETL processes. The authors have worked together to develop the requirements and goals of both theses.

Given the shared context and goals of the two theses, this chapter on Requirements Analysis and Specification will be identical in both works. The following sections will present a detailed discussion of these requirements, which have been jointly identified by the authors. These requirements will serve as a foundation for the subsequent exploration and evaluation of potential parallel ETL implementations for the CTU Data Warehouse.

4.1 Parallelization Requirements

In order to base both bachelor theses on the same requirements and provide a clear goal for comparing the solutions found in the proof of concept (POC) in the conclusion, the authors have jointly defined the following functional and non-functional requirements (functional – F, non-functional – N).

4.1.1 Functional Requirements

F1. Dependency Management between Tasks

The solution must provide dependency management between ETL tasks to maintain the correct order of loading required by the CTU Data Warehouse's database architecture. The solution must emphasize adherence to dependencies since inconsistencies in the data warehouse's data history may arise if they are not maintained.

F2. Loading of a Single IDL Table with History

Currently, it is only possible to run a complete data warehouse loading process, which loads all tables contained in the IDL. The solution must enable loading of a single table or a list of tables, allowing separate loading of pre_stage/pre_stage.clean tables and the history processes only for the necessary stage_increment tables for the correct loading of data into the chosen IDL table or tables.

F3. Loading from Various Source Systems

The solution must allow loading from different types of source systems. Currently, only PostgreSQL and Oracle database systems are used, and they must be included in the solution. Other database systems should be easily added if needed in the future.

F4. Parallelization of Components Using Data or Task Parallelization

Independent ETL processes should be executed in parallel, which corresponds to the definition of task parallelization. Data parallelization is also an option for processes that handle large amounts of data or where it makes sense. Parallelization should be performed on a single server, which the CTU DW has available for ETL loading.

F5. Clarity of Logging

When running processes in parallel, it is not possible to write log information to a single file due to context switching between different threads or processes. The solution must ensure easy retrieval of log information for specific loading of one or more tables or the complete data warehouse loading process.

4.1.2 Non-functional Requirements

N1. Scalability of Parallelization

The solution will allow for the scalability of parallelization, whether it is data or task parallelization. Scalability can be changed by assigning more or fewer resources to parallelization processes.

N2. Portability of the Solution

The created POC must be executable on various types of systems to simplify its use by future users.

4.2 Selection of ETL Parallelization Part

ETL processes currently load nearly all tables from the target database during each loading process. This involves approximately 180 tables in the `dwh3_target` database, which use a similar number of tables in the `dwh3_stage` database. Given that the implementation of the entire ETL process would be considerably time-consuming, a smaller subset of tables (a part of the ETL process) has been selected for demonstrating the proposed solutions. In Table 4.1, the interdependencies among the chosen tables in the stage and target databases are explicated. The selection was made to cover all possible relationships in loading between stage and target tables, namely 1:1, 1:N, N:1, and N:M. Additionally, two tables with a large volume of data and frequent changes (resulting in extended loading times) in the source system (tables `tcitation_affiliations`, `tcitation_authors`) were selected. To ensure completeness, a table without history has been added, featuring a different loading mode that does not use stage tables (table with the `_nohist` suffix).

Table 4.1: Dependencies between selected tables

Source DBMS	Source system	Stage table	Target table	Use of PSC	
PostgreSQL	Grades	classification_text	t_klas_klasifikace	No	
		classification_user_classification	t_klas_klasifikace, t_klas_klasifikace_student		
		student_classification	t_klas_klasifikace_student		
	Usermap	boolean_student_classification			Yes
		number_student_classification			
	Oracle	KOS	string_student_classification		No
			osoby	t_osob_osoba	
		EZOP	tusers		
			tkontakt	t_koud_adresa t_koud_email t_koud_telefonni_cislo	Yes
			tekns	t_orgj_organizacni_jednotka	
torganizations		t_orgj_organizacni_jednotka_externi			
EZOP		tcitation_affiliations	t_extermiorganizacni_jednotka_extermicitaceautor_rel	-	
	tcitation_authors	t_vvvs_externi_citace_autor			
	-	t_vvvs_vedecky_vysledek_bibl_indik_nohist			

Parallelizing ETL: Tool Research and Analysis

This chapter focuses on researching and analyzing various tools and platforms to identify the most suitable solution for implementing parallelized ETL processes. The analysis will be conducted in three main stages:

1. **Research and selection:** Initially, a few prominent ETL tools and platforms will be chosen from the vast range of available solutions for in-depth examination.
2. **Analysis:** The selected tools will be compared based on their ability to meet the functional and non-functional requirements outlined in the previous chapter.
3. **Final Assessment:** The most suitable tool will be chosen not only based on its ability to fulfill the requirements but also considering other factors such as flexibility, community support, ease of integration with existing systems, adaptability to future changes, convenient orchestration, and documentation quality.

A thorough exploration of various ETL tools and platforms is conducted with the goal of identifying the most suitable solution for implementing parallelized ETL processes within the scope of the CTU Data Warehouse.

5.1 Research and Selection

The IT market offers a wide range of solutions concerning data warehousing and beyond. The research conducted during this thesis suggests that most tools suitable for this case can be divided into three categories: data integration tools, ETL tools, and data flow management tools.

- *Data Integration Tools*: These tools focus on combining data from different sources and making it available for further analysis or processing. They handle data extraction, transformation, and loading (ETL), but their scope may also include other tasks like data cleansing, deduplication, and data synchronization. Examples include Talend, Apache Nifi, and Microsoft SQL Server Integration Services (SSIS).
- *ETL Tools*: ETL tools are a subset of data integration tools, specifically designed for the extraction, transformation, and loading of data. They provide a means of transferring data from various sources into a data warehouse or another central repository, where it can be analyzed and transformed. Apart from data warehousing, ETL tools are often used in other field, such as machine learning, etc. ETL tools often include features such as data profiling, data quality management, and data mapping. Examples include Informatica PowerCenter, IBM InfoSphere DataStage, and Microsoft SSIS.
- *Data Flow Managers*: Data flow managers focus on controlling and orchestrating the flow of data between different processing stages and components in a data processing pipeline. They provide a means of managing dependencies, scheduling tasks, and monitoring the execution of data processing tasks. Data flow managers often include features for handling failures, retries, and parallelization of tasks. Examples include Apache Airflow, Apache NiFi, and Luigi.

Given the circumstances, it is important to mention that only open-source software can proceed to the next step of analysis. Therefore, deeper exploration of the following tools will be done: Apache NiFi, Apache Airflow, and Talend. Luigi was not included in this list, as after further researching it appeared to share very similar functionality with Apache Airflow.

5.2 Analysis

The comparison of Apache NiFi, Apache Airflow, and Talend in Table 5.1 shows that how all three solutions meet the functional and non-functional requirements specified for the ETL parallelization process.

5.2.1 Apache NiFi

Apache NiFi is a data integration and data flow management tool that offers a wide range of features and capabilities. However, upon further analysis, it appears that it may not be the most suitable option for this specific project. The potential limitations of NiFi include:

1. **Dependency Management between Tasks (F1):** While NiFi allows for the configuration of data flow dependencies, its primary focus is on data flow management rather than specifically managing ETL task dependencies. Ensuring proper dependency management between ETL tasks might necessitate additional work or custom solutions.
2. **Clarity of Logging (F5):** NiFi provides comprehensive logging information, primarily centered around data flow components. Although it is possible to access log information for specific ETL tasks, this process might not be as straightforward as with other dedicated ETL tools. Additional effort may be required to guarantee clear and easily retrievable logging for parallel processes.

5.2.2 Talend

Talend is a comprehensive data integration and ETL tool that offers a wide range of features for various data management scenarios. In the context of the previously outlined requirements, Talend presents a viable option.

1. **Dependency Management between Tasks (F1):** Offers robust support for managing ETL task dependencies, with potential challenges in complex scenarios.
2. **Loading Single IDL Table (F2):** Supports loading individual or multiple tables, although requiring manual configuration.
3. **Various Source Systems (F3):** Provides strong support for multiple source systems, including PostgreSQL and Oracle.
4. **Parallelization (F4):** Features parallel processing capabilities, but performance may not be optimal compared to specialized tools.
5. **Clarity of Logging (F5):** Offers detailed logging information; however, accessing logs for specific parallel processes could be challenging.
6. **Scalability and Portability (N1 & N2):** Meets scalability and portability requirements.

5.2.3 Apache Airflow

Apache Airflow is a platform developed by Airbnb for programmatically creating, scheduling, and monitoring data pipelines. Similar to Taled, it appears to fulfill the requirements:

1. **Dependency Management between Tasks (F1):** Excels at managing ETL task dependencies through its Directed Acyclic Graph (DAG) structure, ensuring proper execution order.

5. PARALLELIZING ETL: TOOL RESEARCH AND ANALYSIS

2. Loading Single IDL Table (F2): Supports loading individual or multiple tables, with a flexible approach for defining specific tables to load.
3. Various Source Systems (F3): Provides extensive support for multiple source systems, including PostgreSQL and Oracle, and can be easily extended to accommodate others.
4. Parallelization (F4): Offers native parallel processing capabilities, optimizing performance for both task and data parallelization.
5. Clarity of Logging (F5): Delivers clear and easy-to-retrieve logging information, even for parallel processes, simplifying monitoring and troubleshooting.
6. Scalability and Portability (N1 & N2): Meets scalability and portability requirements, allowing for flexible resource allocation and compatibility with various systems.

Both Talend and Apache Airflow seem to be the leaders of this analysis. The next section will dive deeper into details to define the winner.

Table 5.1: Tool Analysis

Requirement	Apache NiFi	Apache Airflow	Talend
F1. Dependency Management	✓/✗	✓	✓
F2. Loading Single IDL Table	✓	✓	✓
F3. Various Source Systems	✓	✓	✓
F4. Parallelization	✓	✓	✓
F5. Clarity of Logging	✓/✗	✓	✓
N1. Scalability of Parallelization	✓	✓	✓
N2. Portability of the Solution	✓	✓	✓

5.3 Final Assessment

After a thorough comparison of Talend and Apache Airflow in the context of the requirements outlined earlier, it is evident that both tools have their unique strengths and limitations. However, it seems that Apache Airflow offers a more profitable solution for the specific needs of this CTU DW.

In terms of support, Apache Airflow has a more extensive community and better support compared to Talend. Additionally, Airflow is entirely free, whereas Talend offers a paid version for advanced features such as cloud storage integration. This aspect may limit the potential for future development and expansion when using Talend.

While Talend's documentation can be difficult to navigate, Airflow's documentation is more user-friendly, making it easier for developers to work with

the tool. Furthermore, Airflow's scheduling and orchestration capabilities are superior, which is particularly important for the management of ETL processes.

It is important to note that Talend is a ready-to-use software solution, while Airflow requires the creation of hand-coded ETL processes. Although this may initially seem more time-consuming, it ultimately results in greater flexibility and customization, which can be advantageous for the specific needs of this project.

However, it is also necessary to acknowledge that the hand-coding aspect of Airflow may lead to a longer debugging process. It is essential to remember that no tool is perfect, and each comes with its own set of trade-offs.

In conclusion, while Apache NiFi, Talend and Apache Airflow are viable options, Apache Airflow stands out as the most suitable choice for CTU DW, considering the fulfillment of the previously established requirements and further analysis. This evaluation should not be seen as a dismissal of other tools, as they may still be appropriate for different use cases and scenarios.

Implementation

In the previous chapter, Apache Airflow emerged as the most suitable solution for implementing the required data flow in this thesis. This chapter aims to offer a comprehensive understanding of Apache Airflow’s concepts and architecture, followed by an in-depth discussion of its practical application within the context of this thesis. Once the desired results are obtained, the implementation will undergo testing, and suggestions for possible improvements will be put forward.

6.1 Introduction to Apache Airflow

Apache Airflow is an open-source platform for data engineers to build, schedule, and monitor batch-oriented workflows. [35] Airflow’s extensible Python framework enables seamless integration with various technologies, and its web interface helps in managing workflow states. The platform can be deployed in diverse ways, from a single laptop process to large-scale distributed setups.

One of the key characteristics of Airflow workflows is their definition in Python code, enabling ‘Workflows as code.’ This approach serves multiple purposes, such as dynamic pipeline generation, extensibility, and flexibility. It also allows workflows to be stored in version control systems, facilitating collaboration and testing. [36]

Apache Airflow is particularly suitable for orchestrating processes, managing connection pools, and establishing dependencies between tasks. Its primary use case is the development of data pipelines. Airflow supports parallel execution, focusing on task parallelism, and its high configurability allows for hand-coded ETL processes, providing data parallelism capabilities. The platform’s user interface offers comprehensive views of pipelines and individual tasks, as well as insightful logging and statistics.

Additionally, Airflow has a vibrant open-source community, ensuring continuous development, testing, and knowledge sharing among users worldwide.

6.2 Basic Components of Apache Airflow

Apache Airflow is a versatile platform designed for creating and managing workflows, which are represented as Directed Acyclic Graphs (DAGs). These DAGs consist of individual Tasks, organized according to their dependencies and data flows.

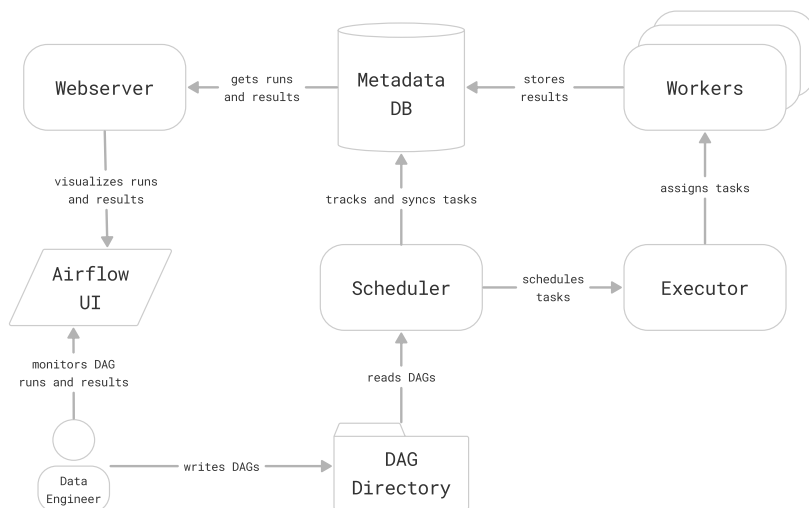


Figure 6.1: Architecture of Apache Airflow and Its Components [37, 36]

An Airflow installation typically includes the following components:

- A *scheduler* that triggers scheduled workflows and submits Tasks to the executor for execution.
- An *executor* responsible for running Tasks. While the default installation runs Tasks within the scheduler, production-ready executors generally utilize worker processes for task execution.
- *Workers*, which are separate processes or machines responsible for executing the Tasks assigned by the executor. They ensure better resource utilization and parallelism in Airflow.
- A *webservice* providing a user interface for inspecting, triggering, and debugging DAGs and Tasks.
- A *directory containing DAG files*, which is accessed by the scheduler, executor, and any workers.
- A *metadata database* used by the scheduler, executor, and webservice to store state information.

Airflow is capable of orchestrating and running a wide range of Tasks, either through high-level support via providers or directly using Shell or Python Operators.

6.2.1 Workloads

In Airflow [36], Tasks are the fundamental units of execution within a DAG and can be categorized into three types:

- **Operators:** Predefined Tasks that can be quickly combined to form most parts of a DAG. Along with Airflow’s basic Operators (such as BashOperator, PythonOperator, HttpOperator, etc.), it is possible to write custom Operators with more complex code.
- **Sensors:** A specialized subclass of Operators that wait for external events to occur.
- **TaskFlow-decorated `@task`:** Custom Python functions packaged as Tasks.

Derived from the BaseOperator, Tasks are organized into a DAG with specified upstream and downstream dependencies to determine their execution order. Tasks can have various states during their lifecycle, such as `scheduled`, `queued`, `running`, and `success`. Other possible states include `failed`, `skipped`, and `up_for_retry`, which help manage error handling and task retries. In a well-functioning DAG, Tasks progress from the initial state of `none` to the final state of `success` by satisfying their dependencies and executing without errors. [36]

6.2.2 Control Flow

In Apache Airflow, DAGs (Directed Acyclic Graphs) are workflows that define a sequence of Tasks and their dependencies. Each DAG represents a pipeline where Tasks are executed in a specific order, ensuring that all dependencies are met before proceeding to the next task.

DAGs are designed for multiple, parallel runs and are parameterized to include an interval for data processing, as well as other optional parameters. Tasks have dependencies on each other, determining the order in which they are executed. Data can be passed between Tasks using XComs, file uploads/downloads from storage services, or TaskFlow API with implicit XComs.

6.2.3 Executors

Airflow can be installed in various configurations, from a single machine (easy to set up but not scalable) to multiple machines (requires more initial work

but offers horizontal scalability). The execution mode depends on the chosen executor, which can be `SequentialExecutor` (default), `LocalExecutor`, `CeleryExecutor`, or `KubernetesExecutor`.

`SequentialExecutor` is the simplest and runs `Tasks` sequentially, suitable for testing and demo purposes but limited to a single machine. `LocalExecutor` runs multiple `Tasks` in parallel on a single machine using a Python First-In-First-Out (FIFO) queue, supporting up to 32 parallel processes by default.

For distributing workloads across multiple machines, the options are `CeleryExecutor` and `KubernetesExecutor`. These are suitable for cases where resource limits are reached, redundancy is needed, or faster workloads are desired. `CeleryExecutor` uses Celery for queuing `Tasks`, with workers processing `Tasks` from the queue. It supports RabbitMQ, Redis, and AWS SQS as brokers and includes a monitoring tool called Flower. The main difference between `CeleryExecutor` and `LocalExecutor` is that the former can distribute tasks across multiple machines. [38]

6.2.4 User Interface

Airflow includes a user interface for monitoring the status of the entire installation and individual DAGs, triggering DAG runs, viewing logs, and performing limited debugging and problem resolution.

6.3 Design of Implementation

With the essential components explained, the next step is to propose an implementation design. This involves defining the configuration for Apache Airflow and outlining the ETL processes, as they will be triggered and executed by Airflow.

Airflow does support data passing between tasks using `XComs`, but with a 512MB limitation, it is unsuitable for managing large data volumes. As a result, data storage between processes is still required. In order to preserve the existing data warehouse architecture, the primary implementation strategy will involve creating hand-coded ETL processes in Python, leveraging Pandas for data processing. Airflow will then be employed to parallelize and orchestrate these processes more effectively.

6.3.1 ETL Processes Design

Preserving the existing CTU DW architecture allows for the reuse of PL/pgSQL procedures with minor modifications. The basic flow remains the same: extracting from source to pre-stage area; cleaning data, counting flags, and loading it to the stage-increment area; transforming data and eventually loading it to the target database. The responsibility of executing Python code for data extractions, transformations, and loading, managing connections, and running

database scripts has now shifted to Airflow, instead of relying on PDI as it was previously.

As previously mentioned, the Pandas module will be utilized for data processing. Although more complex and potentially superior modules exist, Pandas is deemed sufficient for the current proof-of-concept implementation.

Due to the large volume of data, processing it in chunks is recommended. This involves dividing data into equal-sized parts and processing them iteratively.

Each Task in a DAG represents a separate stage of the ETL process. By configuring Airflow, dependencies between these Tasks will be resolved. Whenever possible, one DAG will be created for each source system to provide a clearer view of the whole processing. In cases of interdependencies between multiple source systems, the `TriggerDagRunOperator` will be utilized to address the issue.

6.3.2 Airflow Configuration

It is crucial to establish platform configurations tailored to each specific case. These configurations include: setting the maximum connection pool size, as both source and target databases have limits on the number of connections; setting the maximum number of parallel processes; selecting and configuring a specific executor; configuring connections; and addressing DAG management and error resolution (`dag_import_timeout`, `default_task_retry_delay`, etc.). These configurations are set in a config file called `airflow.cfg`.

Another important aspect is defining DAGs correctly, ensuring proper import and build. For this, a well-defined template is required, which Airflow parses and approves for further use. Proper Task/Operator definitions and dependencies must also be established.

The `LocalExecutor` is suitable for development or testing, but for deployment in a production environment, the `CeleryExecutor` or `KubernetesExecutor` is typically recommended. In this implementation, the `CeleryExecutor`, along with the `RabbitMQ` queue broker, will be applied. The final architecture is depicted in Figure 6.2. It consists of several components, also mentioned in Section 6.2:

It consists of several components also mentioned in Section 6.2:

- **Workers** – Execute the assigned Tasks;
- **Scheduler** – Responsible for adding the necessary Tasks to the queue;
- **Webserver** – HTTP Server provides access to DAG/Task status information;
- **Database** – Contains information about the status of Tasks, DAGs, Variables, connections, etc.;

- **Celery** – Queue mechanism.

It is worth noting that the Celery queue consists of two components:

- **Broker** – Stores commands for execution;
- **Result Backend** – Stores status of completed commands.

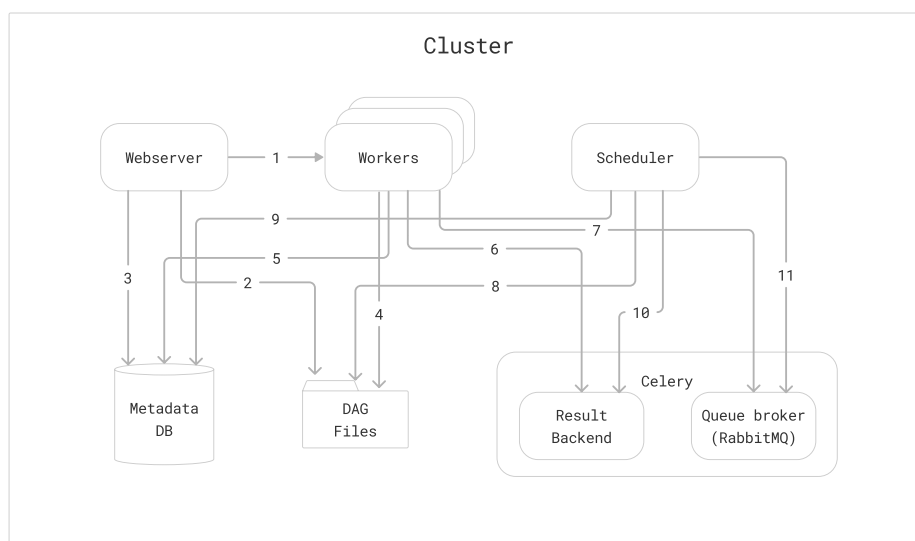


Figure 6.2: Final Architecture [37, 36]

The components communicate with each other at several points, as illustrated in Figure 6.2:

1. **Webserver** → **Workers**: Fetches Task execution logs;
2. **Webserver** → **DAG files**: Reveals the DAG structure;
3. **Webserver** → **Database**: Fetches the status of the Tasks;
4. **Workers** → **DAG files**: Reveals the DAG structure and execute the Tasks;
5. **Workers** → **Database**: Gets and stores information about connection configuration, variables and XCOM;
6. **Workers** → **Celery's result backend**: Saves the status of Tasks;
7. **Workers** → **RabbitMQ**: Stores commands for execution;

8. **Scheduler** → **DAG files**: Reveals the DAG structure and executes the Tasks;
9. **Scheduler** → **Database**: Store a DAG run and related Tasks;
10. **Scheduler** → **Celery's result backend**: Gets information about the status of completed Tasks;
11. **Scheduler** → **RabbitMQ**: Put the commands to be executed.

The complete implementation will entail rewriting ETL processes in Python using Pandas as Tasks in DAGs and setting up and configuring Apache Airflow so that these DAGs are parallelized and assume control over the entire process. [36]

6.4 Initial Setup

To start working with Apache Airflow, it is essential to understand the prerequisites [36]. This implementation will be executed using the following specifications: Python 3.10, PostgreSQL 14, 16GB RAM, and Fedora 36. These specifications are compatible with the latest Apache Airflow version at the time of writing – 2.5.3.

The installation process is relatively straightforward if one adheres to the instructions provided in the official documentation [36]. Some aspects that require specific attention include:

- **Metastore database**: The default database for Airflow is SQLite, which does not support parallel task processing. As parallelism is a core focus of this thesis, an alternative DBMS is recommended. In this case, PostgreSQL 14 running in Docker is used;
- **DAG directory**: It is crucial to create a folder named 'dags' to store DAGs. Additionally, proper directory organization is vital. The structure employed for this project, shown as a tree in Figure 6.3, diverges from the one provided in the Airflow documentation. The path to the home directory (`airflow`) must be stored as a variable: `export AIRFLOW_HOME=~/.airflow`.
- **Webserver and Scheduler**: These are two distinct components. To execute DAGs from the webserver, it is necessary to start the Scheduler in a separate terminal window. The Scheduler is responsible for parsing DAGs and checking for errors. The same applies to the CeleryExecutor and its auxiliary components;
- **CeleryExecutor**: When implementing the CeleryExecutor, the initial step is to set up the following components: Celery, Flower UI, and a

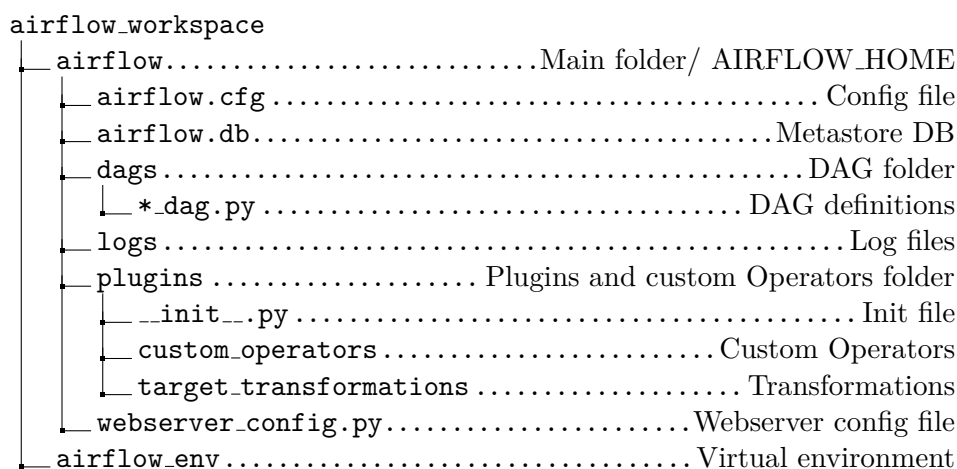


Figure 6.3: Airflow Working Directory Tree Structure

queue broker (RabbitMQ in this instance). Finally, modify the configurations in `airflow.cfg`.

6.5 ETL Implementation

In this section, the focus is on the implementation of ETL processes for the DW CTU using hand-coding methods. The discussion covers all stages of the process, from extracting data from various sources to data cleansing, formatting, and ultimately loading into the Integrated Data Layer (IDL). The aim is to provide a comprehensive, step-by-step overview of the ETL process, shedding light on best practices and pitfalls to avoid.

6.5.1 Database Connections

Before delving into the transformations, it is necessary to obtain source data. As outlined in the Requirements Analysis chapter (see Chapter 4), raw data from four source systems will be processed: Grades, Usermap, KOS, and EZOP. While Grades employs the PostgreSQL DBMS, the others utilize Oracle. For this purpose, SQLAlchemy, a popular Object Relational Mapper (ORM) in Python, is required. SQLAlchemy supports a wide range of databases and employs specific drivers for each database system. Depending on the database, it is essential to install the appropriate driver package using the following pip commands:

```

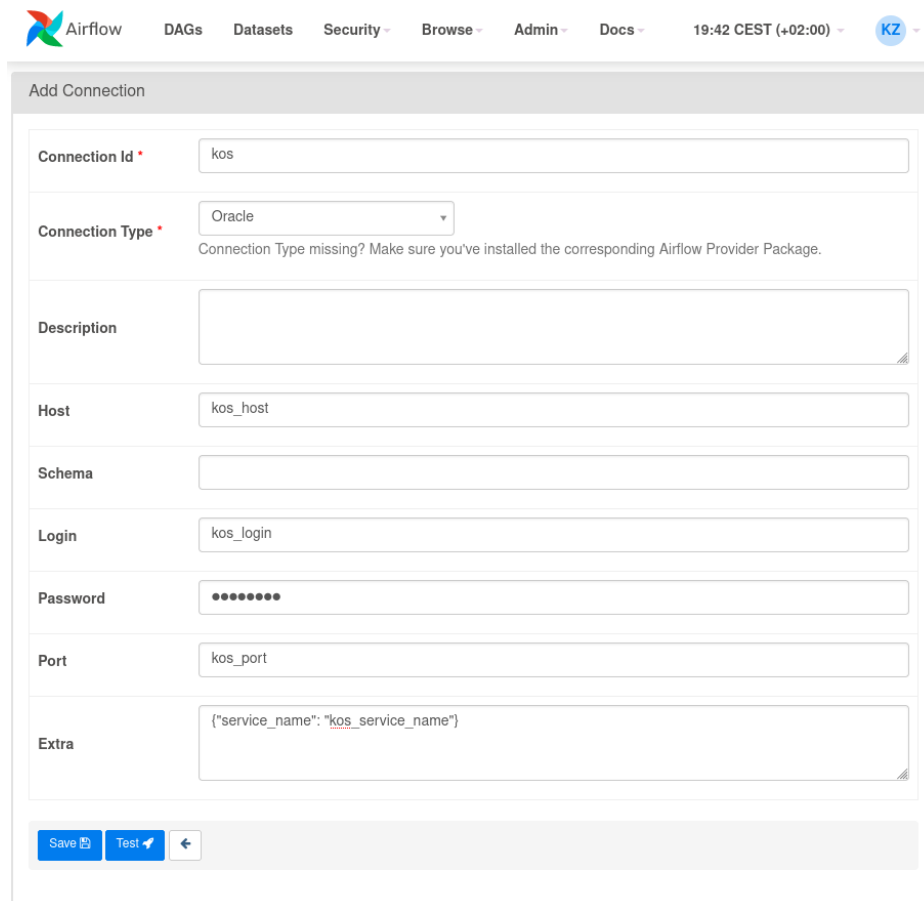
(venv)$ pip install psycopg2 # for PostgreSQL databases access
(venv)$ pip install cx_Oracle # for Oracle databases access

```

Listing 6.1: PIP commands to install database access drivers

SQLAlchemy is installed along with Apache Airflow. In Apache Airflow, database connections are managed using the Connections feature, which offers a built-in mechanism for managing and retrieving these connections in a centralized and secure manner.

Connecting to PostgreSQL databases did not pose any issues. However, the Airflow documentation does not provide clear guidance on connecting to an Oracle database. Since PostgreSQL and Oracle have different connection parameters, challenges arise. PostgreSQL uses a URL-based connection string format, whereas Oracle employs a more complex format that includes a service name or SID (System Identifier). It was not immediately clear where to place the service name or SID, but the trial-and-error method helped resolve this issue:



The screenshot shows the 'Add Connection' form in Apache Airflow. The form is titled 'Add Connection' and contains the following fields:

- Connection Id ***: Text input field containing 'kos'.
- Connection Type ***: Dropdown menu set to 'Oracle'. Below it is a note: 'Connection Type missing? Make sure you've installed the corresponding Airflow Provider Package.'
- Description**: Text area field.
- Host**: Text input field containing 'kos_host'.
- Schema**: Text input field.
- Login**: Text input field containing 'kos_login'.
- Password**: Password input field with masked characters (dots).
- Port**: Text input field containing 'kos_port'.
- Extra**: Text area field containing a JSON dictionary: `{"service_name": "kos_service_name"}`.

At the bottom of the form, there are three buttons: 'Save' (with a save icon), 'Test' (with a test icon), and a back arrow button.

Figure 6.4: Apache Airflow Connections: KOS – Oracle [37]

In cases where the SID is used, it should also be entered in the Extra field as a JSON dictionary.

Next, a connection within the code needs to be created. As Airflow allows for connection management, utilizing ‘hooks’ offers a more convenient approach. In Apache Airflow, the `PostgresHook/OracleHook` is a hook class that simplifies interactions with PostgreSQL/Oracle databases. The following example demonstrates how to use hooks to establish a connection with `PostgresHook`:

```
from airflow.providers.postgres.hooks.postgres \
    import PostgresHook

def create_connection_pg(pg_conn_id: str, db: str):
    pg_hook = PostgresHook(
        postgres_conn_id=pg_conn_id,
        database=db
    )
    return pg_hook.get_conn()
```

Listing 6.2: Establishing connection with `PostgresHook`

6.5.2 Extraction

Before delving into the data extraction process, it is crucial to note that data is processed in chunks, which are equal-sized subsets of the initial dataset. This approach helps prevent running out of random access memory (RAM). A for loop is employed to process data in a single transaction so that in case of an error, the previously uploaded data is rolled back. Data is fetched using the `cursor.execute(select_stmt)` and `cursor.fetchmany()` functions, and subsequently stored in a NumPy array (since loading to stage does not require any transformations) before being uploaded to the staging area. Additionally, an MD5 hash is calculated using the `hashlib` module. The data upload process is facilitated by the `COPY` statement, which enables efficient bulk data loading through its optimized binary data format. `Psycopg2`'s `copy_expert` required string iterator. The `get_data_iterator` function takes an array as input, converts it to a list of tuples, and returns a `StringIteratorIO` object. This object is an iterator that allows reading the data row by row as strings, with fields separated by the delimiter, making it suitable for use with the `copy_expert` function for data loading. The following code snippet illustrates a simplified implementation of the extraction process:

```
def load_table(self, **kwargs):
    # Connect to staging database and create cursor
    with create_connector(self.stg_conn_id,
                        self.stg_db) as stg_conn:
        try:
            src_conn = create_connection(
```

```
        self.src_conn_id,
        self.src_db)
    stg_cursor = stg_conn.cursor()

    # Load data in chunks into staging database
    self.load_table_in_chunks(
        src_conn=src_conn,
        stg_cursor=stg_cursor)

    # Commit changes to staging database and close cursor
    stg_cursor.close()
    stg_conn.commit()
except Exception as error:
    # Rollback changes and raise the error
    stg_conn.rollback()
    print("Error: ", error)
    raise
finally:
    # Close source database connection
    src_conn.close()
```

Listing 6.3: Loading table in chunks

It is important to mention that these functions are methods defined within the `LoadToStageOperator` class. The `PythonOperator` is not suitable in this scenario, as it is typically used for executing simpler Python code. This custom operator allows for greater flexibility and, ultimately, results in a more efficient and faster parsing of the DAG with Airflow.

6.5.3 Transformation

This stage of the ETL process is extensive and, as a result, is divided into separate parts, each implemented in the `MakeIncrementOperator` and `LoadToTargetOperator`.

`MakeIncrementOperator` implements methods that call stored procedures. `public.inc_clear_state_flag_schema_table_args`, the first stored procedure in the stage database, performs the same tasks as stored procedure `public.inc_clear_state_flag` described in Section 3.2.2. However, it has been slightly modified to process specific tables rather than all tables, taking schema and table names as arguments to determine which table needs processing. The next function calls a set of three procedures:

- `public.inc_find_modified_in_pre_stage_schema_table_args`,
- `public.inc_find_new_in_pre_stage_schema_table_args`,
- `public.inc_find_deleted_in_pre_stage_schema_table_args`.

Their functionalities were also described in Section 3.2.2 and underwent similar modifications.

Before the final data loading, the `LoadToTargetOperator` performs several transformations on the data. For most tables, the data withstands minimal transformation, such as replacing Python’s `NaN` and `NaT` with `NULL`. However, some dependent tables require database lookups and individual transformations. As a result, the `LoadToTargetOperator` is implemented in a general manner, allowing developers to provide only necessary data information (e.g., database, schema, table names, columns, data types, etc.) without further coding. When custom transformations are required, they can be implemented in separate files (in `target_transformation` folder shown in Figure 6.3) and then passed as higher-order functions to the `LoadToTargetOperator`. These processes are also applied by dividing data into smaller chunks.

6.5.4 Loading

The subsequent implementation phase of the `LoadToTargetOperator` is focused on the data loading process. A similar approach to the extraction process is applied, which involves connecting to the staging database and processing data in chunks. After all transformations, the dataset is divided into three groups: new data – ‘N’, modified data – ‘M’, and deleted data – ‘D’. Each group is processed differently, as described in Section 3.2.2

A similar bulk loading implementation is used to upload new records, utilizing the same `COPY` statement concept as in the extraction stage. Processing deleted records remains the same as in the existing solution, executing the SQL script shown in Section 3.2.3.

However, a challenge arises when loading modified data. This task is performed by calling the `public.inc_historize_case_m` stored procedure in the target database. Executing this procedure one row at a time is time-consuming, indicating the need for data parallelism. To address this issue and improve performance, data parallelism is employed. This approach involves creating n connections and using the `ThreadPoolExecutor` to create n threads to handle the data processing simultaneously. The implementation ensures efficient processing and enhances the overall performance of the data loading process.

The following code snippet demonstrates a basic data parallelism implementation to mitigate this issue:

```
def load_modified(self, modified_data, tg_cursor):
    # ...
    def process_rows_in_parallel(modified_data, num_conns):
        # Divide the DataFrame into smaller DataFrames
        data_splits = np.array_split(modified_data, num_conns)
```

```
# Create connections and process records in parallel
# using ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=num_conns) as exec:
    # Submit tasks and wait for their completion
    futs = [exec.submit(process_rows, data_split, conns[i])
            for i, data_split in enumerate(data_splits)]
    for future in futs:
        future.result()

# Close connections after processing all rows
# ...
# ...

# Define the number of connections
num_conns = 10
process_rows_in_parallel(modified_data, num_conns)
```

Listing 6.4: Data parallelism implementation

1. `load_modified` function takes the modified data and a target cursor as input parameters.
2. Inside `load_modified`, the modified data is first concatenated using the `concat_fields` function.
3. The `process_rows_in_parallel` function is called, which does the following:
 - Splits the modified data into smaller DataFrames based on the number of connections.
 - Creates a list of connections.
 - Utilizes `ThreadPoolExecutor` to process the data splits in parallel, with each worker handling a data split and connection.
 - Submits the tasks to the executor and waits for completion.
 - Closes the connections after processing.

This implementation improves the performance of the data loading process by processing rows simultaneously, using multiple connections.

6.6 Task Parallelism Implementation

After analyzing the ETL processes, it became evident that the Producer-Consumer pattern is the most suitable solution in this case. The Producer-

Consumer pattern is a concurrency design pattern where one component (producer) generates data, and another component (consumer) processes it, typically using a shared buffer or queue. This pattern helps to decouple task generation from task execution, allowing for better scalability and parallelism. Using the Celery executor in Airflow implements a distributed task queue pattern, which can be viewed as a variation of the Producer-Consumer pattern. In this context, the Airflow scheduler acts as the producer, while the Celery worker nodes act as consumers.

The producer (Airflow scheduler) creates tasks and puts them into the task queue (message broker like RabbitMQ or Redis). The consumers (Celery worker nodes) fetch tasks from the queue and execute them in parallel. This pattern allows efficient distribution of tasks across multiple workers, enabling parallel processing and load balancing.

To achieve effective task parallelism, it is crucial to analyze table dependencies and, according to them, build a proper Directed Acyclic Graph (DAG). As an example, the Grades tables will be analyzed.

Implementing the loading of this domain implies that there will be 7 stage tables, which will subsequently be loaded into 2 target tables only. During the loading stage, those 2 tables make database lookups on the remaining ones. This means that the target table `t_klas_klasifikace`, as a primary source, will use the stage table `classification`, and for database lookup, `classification_text` is used. As for the second target table, referred to as `t_klas_klasifikace_student`, the stage table is `student_classification`; lookup tables are `classification_user`, `boolean_student_classification`, `number_student_classification`, and `string_student_classification`. Interestingly, thanks to parallel processing and dependency resolution between tables, there is a great possibility to avoid redundant flag calculation for lookup tables since lookups are held on the pre-stage tables.

Through this, it is rather evident that primary and lookup tables should be loaded and processed before final loading to the target table. This can be achieved by declaring dependencies between DAG's Tasks. There are two ways to do so:

1. `>>` and `<<` operators as shown in Listing 6.5
2. `set_upstream` and `set_downstream` methods:

```
first_task.set_downstream(second_task, third_task)
third_task.set_upstream(fourth_task)
```

The Airflow UI webserver shows the following:

```
task_start_grades >> [task_load_boolean_student_classification,
                      task_load_classification,
```

```

task_load_classification_text,
task_load_student_classification,
task_load_classification_user,
task_load_number_student_classification,
task_load_string_student_classification
]
task_load_classification >> task_make_increment_classification
task_load_student_classification >> task_make_increment_student_classification
[task_make_increment_student_classification,
task_load_boolean_student_classification,
task_load_number_student_classification,
task_load_string_student_classification,
task_load_classification_user] >> task_load_t_klas_klasifikace_student
[task_make_increment_classification,
task_load_classification_text] >> task_load_t_klas_klasifikace

```

Listing 6.5: Task Dependencies – Grades DAG

The Airflow UI webserver shows the following result:

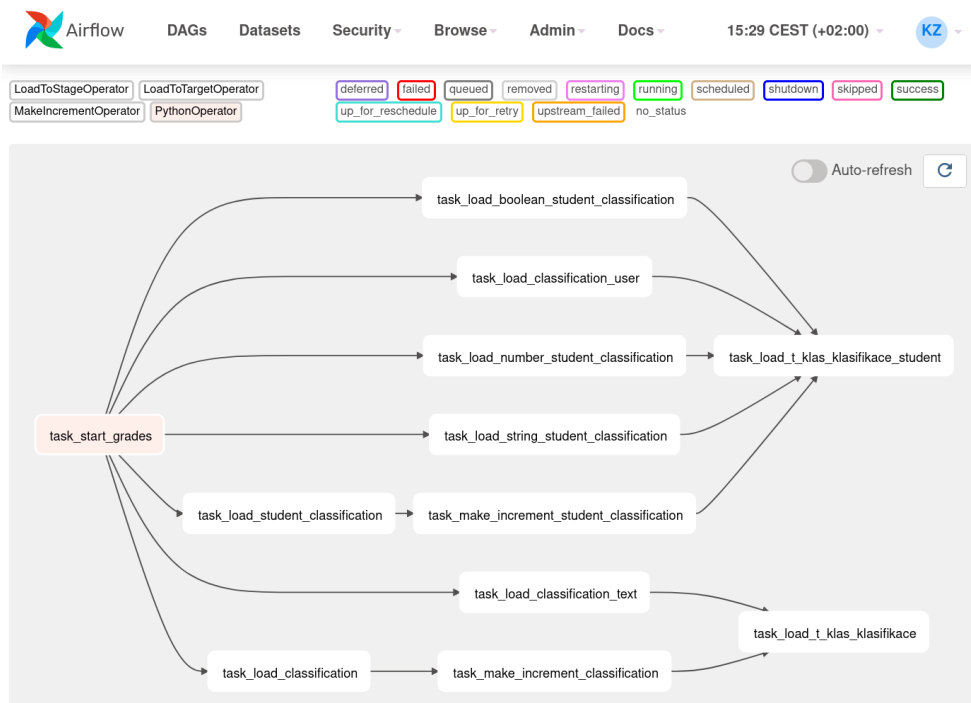


Figure 6.5: Apache Airflow DAG – Grades Parallel ETL Tasks [37]

6.7 Scheduling and Orchestration

Upon successful implementation of the ETL processes and the construction of the DAG, it becomes necessary to address the scheduling and orchestration of the entire process. As Apache Airflow is widely employed for the programmatic authoring, scheduling, and monitoring of workflows, it facilitates a practical approach to accomplishing these tasks.

6.7.1 UI Graphs and Task Monitoring

Airflow's web-based user interface (UI) enables users to visualize DAG structure and progress. It includes multiple views like tree view, graph view, and Gantt chart view, offering different perspectives on DAG execution. Users can monitor task statuses, view logs, and manage DAG runs through the web interface. The UI also allows users to trigger DAG runs manually and manage user roles and permissions.

6.7.2 Logging and Error Handling

Airflow provides robust logging capabilities that help users track and monitor workflows. Task logs are stored in a specified directory (Figure 6.3), and log storage can be customized using Python's logging module. Airflow supports log aggregation using services like Elasticsearch. Error handling mechanisms include email notifications, task retries, retry intervals, and custom error handling functions.

6.7.3 Task Statuses and Dependencies

Each Task has a status indicating its progress within the workflow, such as 'queued', 'running', 'success', 'failed', 'up_for_retry', and 'skipped'. Task statuses can be viewed in the Airflow UI. Users can view the status of each Task within the DAG through the Airflow UI, which uses color coding to visually represent each status. Task statuses provide valuable information for identifying issues and bottlenecks within the workflow. Users can also set up task dependencies, ensuring that Tasks are executed in the correct order.

6.7.4 Crontab Time Defining and Scheduling

Airflow uses a cron-like syntax for scheduling DAGs. Users can specify frequency, interval, and exact times for executing workflows. Airflow supports advanced scheduling features like dynamically generated schedules and relative time scheduling.

6.7.5 Configuration and Customization

Airflow provides numerous configuration options, such as task parallelism, timeouts, and retries. Users can also extend its functionality using custom operators, sensors, and hooks, allowing for integration with various external systems.

6.8 Testing

Testing is a crucial aspect of this thesis. The primary challenge lies in the large volume of data that undergoes the ETL processes. Additionally, new input data is often unpredictable, making it difficult to determine when the transformation processes may fail. In consultation with the supervisor, the following testing methodology was agreed upon:

1. Create all necessary tables for testing in the staging and integrated data layers, and copy the data from the tables processed and loaded by the existing solution into the newly created tables.
2. Run the DW refreshment on the original tables using the existing solution and run the DW refreshment on the copied tables using this implementation.
3. Compare the data in the tables after the refreshments have been completed.

This methodology is also applied in evaluating performance comparisons. Specifically, it measures the time taken by the existing PDI implementation to load the tables and compares it to the time taken by the solution presented in this thesis. This comparison helps assess the efficiency and effectiveness of the new implementation relative to the existing one.

The results obtained show that the target tables have the same number of new, modified, and deleted data records for each tested table. There are, however, some non-critical differences in the data:

1. Different MD5 hashes in stage-increment tables: This discrepancy is due to variations in data types and methods for calculating hashes between PDI and Python's `hashlib` library.
2. Different types of boolean values: PDI converts `True/False` to `Y/N`, while this implementation maintains the source values. However, this issue can be easily resolved.

3. Different value formats: In the IDL table `t_klas_klasifikace_student`, there is a column `hodnota` (English: value) of type TEXT, which contains three types of values: boolean, string, and number. The number values are of type double precision in the source table. However, PDI distinguishes between integers and floats and loads them accordingly, while this implementation always loads them as float numbers. This means that if the number is an integer, it is loaded as a float, such as 7 being loaded as 7.0.

6.9 Further Development and Optimization

Based on the current ETL implementation using Apache Airflow, there are several opportunities for further development and optimization to enhance performance, maintainability, and scalability. The following points outline some potential improvements:

- **Optimize data processing with more efficient Python libraries:** While the current implementation leverages Pandas, which is a popular and versatile library, there are other libraries that can potentially offer better performance and more efficient memory usage. Future work could involve exploring alternatives, such as Dask, Vaex, or Modin, to further optimize data processing.
- **Parallelize processing of new, modified, and deleted branches:** The loading process could benefit from parallelizing the execution of new, modified, and deleted branches in terms of code and database capability. This would require analyzing the possibilities for concurrent execution and making necessary adjustments to the implementation to ensure correct and efficient processing.
- **ETL code refactoring:** As with any software project, continuous refactoring of the ETL codebase can lead to more maintainable and modular code. This includes identifying and addressing redundancies, simplifying complex logic, and improving code readability and documentation.
- **Address issues discussed in previous sections:** Some issues were mentioned in Section 3.3, which could be further investigated and resolved. By addressing these concerns, the overall stability, performance, and maintainability of the ETL pipeline can be enhanced.

By pursuing these further development and optimization efforts, the ETL implementation using Apache Airflow can continue to evolve, offering improved performance, scalability, and ease of maintenance. This, in turn, will provide a more robust and efficient data processing pipeline for the CTU Data Warehouse.

Evaluation

This chapter presents the results of the implementation and a comparison between the existing sequential solution in PDI and the proposed solution in terms of performance.

7.1 Comparison

The primary goal of this thesis is to enhance the performance of DW refreshment by parallelizing ETL processes, thereby reducing the overall loading time. The methodology for this evaluation is described in the previous chapter in Section 6.8.

DW refreshment is a highly time and resource-consuming process. Consequently, the performance of the sequential loading currently used in production was measured once and for both concurrent bachelor theses to establish a baseline. The following are the characteristics of the computing device used for this evaluation:

- Processor: Intel Core i7-8565U 1.8GHz (8th Generation) featuring 4 cores and 8 threads;
- Memory: 16GB LPDDR3 RAM;
- Transfer speed: 90-100 MB/s.

The sequential loading process took 8 hours and 27 minutes to complete. In contrast, the solution proposed in this thesis was executed on a machine with the following characteristics:

- Processor: Intel Core i5 7200U 2.5 GHz (7th Generation) featuring 2 cores and 4 threads;
- Memory: 16GB LPDDR3 RAM;
- Transfer speed: 90-100 MB/s.

7. EVALUATION

The overall DW refreshment time for the proposed solution was 3 hours and 52 minutes.

Table 7.1: Sequential and Parallel Loading: Comparison

Loading Task	Sequential	Parallel
boolean_student_classification	0:00:04	0:00:18
classification + SI	0:00:03	0:00:20
classification_text	0:00:01	0:00:08
classification_user	0:00:01	0:00:04
number_student_classification	0:00:16	0:01:03
string_student_classification	0:00:03	0:00:18
student_classification + SI	0:00:26	0:02:30
osoby + PSC + SI	0:01:05	0:01:40
tusers	0:00:30	0:00:13
tkontakt + PSC + SI	0:02:18	0:02:44
tekns + PSC + SI	0:00:03	0:00:03
torganizations + PSC + SI	0:00:04	0:00:04
tcitation_affiliations + PSC + SI	3:16:47	1:50:31
tcitation_authors + PSC + SI	3:46:58	3:52:13
t_klas_klasifikace	0:02:25	0:00:50
t_klas_klasifikace_student	0:16:41	0:28:48
t_osob_osoba	0:00:11	0:03:27
t_koud_adresa	0:17:40	0:15:04
t_koud_email	0:09:41	0:03:12
t_koud_telefonni_cislo	0:09:50	0:03:44
t_orgj_organizacni_jednotka	0:00:01	0:00:05
t_orgj_organizacni_jednotka_externi	0:00:01	0:00:01
t_externiorganizacnijednotka_externicitaceautor_rel	0:00:27	0:00:18
t_vvvs_externi_citace_author	0:01:15	0:00:36
t_vvvs_vedecky_vysledek_bibl_indik_nohist	0:00:28	0:00:16
In total	8:07:19	3:52:13

In the selected portions of ETLs, the tables `tcitation_affiliations` and `tcitation_authors` are the largest, with over 65 million and 75 million records, respectively. Consequently, the loading time for these tables varies significantly compared to the others. The loading time for the new parallel solution, which includes prestage-clean and stage-increment operations, is determined by the time taken to load the data into the staging area (0:58:55) and the time spent counting flags (0:51:36) using a database stored function. Due to differences in MD5 hashing between PDI and Python's `hashlib`, the loading time increases for the first load with this new solution. However, subsequent calculations are expected to take less time.

It is a noteworthy observation that the performance of the parallel solution is slightly lower in some smaller tables compared to the sequential PDI implementation. These differences can primarily be attributed to the following factors:

- The sequential solution was tested on more efficient hardware, which could have contributed to its better performance in comparison to the parallel implementation.
- The Python implementation employs distinct connections and data fetching technologies, such as `cx_Oracle` and `psycopg2`, as opposed to the JDBC connections utilized by PDI.
- The new MD5 hashes calculated in the Python implementation differ from those in PDI, resulting in an increased amount of time required to compute new, modified, and deleted flags during the stage-increment process.

These factors collectively explain the observed performance disparities between the parallel and PDI implementations. Further optimization and refinement of the parallel solution may help bridge this performance gap.

7.2 Requirements Evaluation

Functional Requirements

1. **F1. Dependency Management between Tasks:** The implemented solution effectively manages dependencies between ETL tasks, ensuring the correct order of loading as required by the CTU Data Warehouse's database architecture. By adhering to these dependencies, the solution prevents inconsistencies in the data warehouse's data history.
2. **F2. Loading of a Single IDL Table with History:** The solution enables the loading of a single table or a list of tables, allowing separate loading of `pre_stage/pre_stage_clean` tables and history processes only for the necessary `stage_increment` tables for the correct loading of data into the chosen IDL table or tables.
3. **F3. Loading from Various Source Systems:** The solution supports loading from different types of source systems. It currently supports PostgreSQL and Oracle database systems and is designed to easily accommodate other database systems in the future, if needed.
4. **F4. Parallelization of Components Using Data or Task Parallelization:** Independent ETL processes are executed in parallel, corresponding to the definition of task parallelization. Parallelization is

performed on a single server, which the CTU DW has available for ETL loading.

5. **F5. Clarity of Logging:** The solution ensures easy retrieval of log information for specific loading of one or more tables or the complete data warehouse loading process, even when running processes in parallel. This is crucial as it is not possible to write log information to a single file due to context switching between different threads or processes.

Non-functional Requirements

1. **N1. Scalability of Parallelization:** The solution allows for the scalability of parallelization. Scalability can be adjusted by assigning more or fewer resources to parallelization processes, ensuring that the system can adapt to changes in workload.
2. **N2. Portability of the Solution:** The implemented POC is executable on Linux and macOS systems. For execution on Windows machines, a virtual environment is required.

7.3 Result

The overall result of this work demonstrates that the implementation of task parallelism for ETL processes using Apache Airflow as the core platform for constructing, managing, orchestrating, and scheduling the loading was successful. The implementation utilized Pandas and NumPy libraries, and the code is structured in a way that simplifies the migration to this solution. The following are the strengths and weaknesses of the solution and its potential for future use.

Strengths

- The solution provides flexibility in the ETL process, allowing for better adaptation to changes in data sources and requirements.
- The code is designed in a way that requires only minimal configuration (such as connection settings and table information), making it easier to set up and use.
- The implemented solution outperforms the existing ETL process in terms of speed and efficiency.
- The user-friendly UI offers an intuitive way to control the process, with clear visualizations and loggings for effective monitoring.
- Apache Airflow is a modern and highly supported open-source software with a robust ecosystem.

Weaknesses

- The implementation process can be challenging and time-consuming due to the complexity of the solution and the need for hand-coding ETL tasks. Thus, it takes more time than using other drag-and-drop, ready-to-use software.
- The solution may still be somewhat unstable, as multiple testing is challenging to perform. This may result in further refactoring requirements.
- Longer initial loading.
- Like any coded project, it requires proper maintenance.

Despite the weaknesses, the solution is functional and recommended for future use, provided that the necessary resources are allocated for its development and maintenance. The evaluation of whether the bachelor's thesis assignment has been fulfilled is as follows:

1. The requirements for parallelization of the DW CTU ETL processes were specified in Chapter 4.
2. A suitable part of the DW CTU ETL processes to be parallelized was selected and represented in Table 4.1.
3. Research on two or more tools suitable for use in this problem was undertaken in Chapter 5.
4. The tool was selected and analyzed in Section 6.2.
5. The parallelization of the selected DW CTU ETL part using Apache Airflow was proposed and implemented in Chapter 6.
6. The solution was evaluated in terms of meeting the requirements and in terms of its potential for future use in managing the entire DW CTU ETL process.

Conclusion

In conclusion, the primary objective of this thesis was to propose and implement parallel ETL processes for the CTU Data Warehouse with the aim of achieving improved performance and faster processing times. A proof of concept (POC) was successfully developed and implemented, focusing on a selected portion of the entire ETL process for the DW CTU. The solution leverages the power of Apache Airflow, providing a user-friendly UI, easy-to-understand code structure, and flexibility in handling ETL tasks.

The main contribution of this work is the optimization of the DW refreshment process, significantly reducing the workload on VIC servers. By minimizing the time required for the entire DW refreshment process, additional resources are made available for other tasks within the CTU Data Warehouse environment, improving overall efficiency.

For future development, it is recommended to explore the use of more efficient packages and libraries for enhanced data processing. Additionally, further refactoring and applying the developed parallel ETL approach to the remaining DW CTU tables would be beneficial. The implementation presented in this thesis serves as a solid foundation for these future improvements, ultimately leading to a more efficient and performant Data Warehouse for the CTU.

Bibliography

1. KUZNETSOV, S. *Datový sklad fakulty*. Prague, 2013. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology.
2. INMON, William H. *Data architecture: The information paradigm*. QED Information Sciences, Inc., 1992.
3. KIMBALL, Ralph; CASERTA, Joe. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Indianapolis, IN: Wiley, 2004. ISBN 978-0764579233.
4. SIMSION, Graeme; WITT, Graham. *Data modeling essentials*. Elsevier, 2004.
5. BALLARD, Chuck. *Data Modeling Techniques for Data Warehousing*. Upper Saddle River, NJ: Prentice Hall PTR, 2000. ISBN 9780130673046.
6. EKANAYAKE, I. *Inmon vs. Kimball: The Great Data Warehousing Debate* [online]. 2021. [visited on 2023-04-24]. Available from: <https://medium.com/cloudzone/inmon-vs-kimball-the-great-data-warehousing-debate-78c57f0b5e0e>.
7. INMON, W. H. *Building the Data Warehouse*. 4th ed. Foster City, CA: Hungry Minds, 2005.
8. ARIYACHANDRA, Thilini; WATSON, Hugh J. Which data warehouse architecture is most successful? *Business intelligence journal*. 2006, vol. 11, no. 1, p. 4.
9. KIMBALL, Ralph; ROSS, Margy. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. 3rd. John Wiley & Sons, 2013.
10. CHEN, Peter Pin-Shan. The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)*. 1976, vol. 1, no. 1, pp. 9–36.

11. BERTINO, Elisa; MARTINO, Lorenzo. Object-oriented database management systems: concepts and issues. *Computer*. 1991, vol. 24, no. 4, pp. 33–47.
12. BRUCKNER, Robert M; LIST, Beate; SCHIEFER, Josef. Striving towards near real-time data integration for data warehouses. In: *Data Warehousing and Knowledge Discovery*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 317–326. Lecture notes in computer science.
13. ZIEGLER, Patrick; DITTRICH, Klaus R. Data integration—problems, approaches, and perspectives. *Conceptual modelling in information systems engineering*. 2007, pp. 39–58.
14. CALVANESE, Diego; DE GIACOMO, Giuseppe; LENZERINI, Maurizio; NARDI, Daniele; ROSATI, Riccardo. Data integration in data warehousing. *International Journal of Cooperative Information Systems*. 2001, vol. 10, no. 03, pp. 237–271.
15. NEGASH, Solomon. Business intelligence. *Communications of the association for information systems*. 2004, vol. 13, no. 1, p. 15. Available from DOI: 10.17705/1cais.01315.
16. VASSILIADIS, Panos. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)*. 2009, vol. 5, no. 3, pp. 1–27. Available from DOI: 10.4018/jdwm.2009070101.
17. FATIMA, Anosh; NAZIR, Nosheen; KHAN, Muhammad Gufran. Data cleaning in data warehouse: A survey of data pre-processing techniques and tools. *Int. J. Inf. Technol. Comput. Sci*. 2017, vol. 9, no. 3, pp. 50–61.
18. GIORDANO, Anthony David. *Data integration blueprint and modeling: techniques for a scalable and sustainable architecture*. Pearson Education, 2010.
19. WU, Shaomin. A review on coarse warranty data and analysis. *Reliability Engineering & System Safety*. 2013, vol. 114, pp. 1–11. Available from DOI: 10.1016/j.ress.2012.12.021.
20. CASERTA, Joe; KIMBALL, Ralph. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, 2013. ISBN 978-0764579233.
21. VASSILIADIS, Panos; SIMITSIS, Alkis. Extraction, Transformation, and Loading. *Encyclopedia of Database Systems*. 2009, vol. 10.
22. BALA, Mahfoud; BOUSSAID, Omar; ALIMAZIGHI, Zaia. A Fine-Grained Distribution Approach for ETL Processes in Big Data Environments. *Data & Knowledge Engineering*. 2017, vol. 111, pp. 114–136.

23. KUKREJA, Manoj. *Handling Slowly Changing Dimensions (SCD) using Delta Tables* [online]. 2023. [visited on 2023-04-24]. Available from: <https://towardsdatascience.com/handling-slowly-changing-dimensions-scd-using-delta-tables-511122022e45>.
24. BHARGAV, Nikhil. *Lookup Table in Databases*. 2023. Available also from: <https://www.baeldung.com/cs/lookup-table-in-databases>.
25. VASSILIADIS, Panos; SIMITSIS, Alkis. Near real time ETL. In: *New trends in data warehousing and data analysis*. Springer, 2008, pp. 1–31.
26. ALMASI, George S; GOTTLIEB, Allan. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., 1994. ISBN 978-0-8053-0177-9.
27. ADVE, S; ADVE, Vikram S; AGHA, Gul; FRANK, Matthew I; GARZARÁN, MJ; HART, JC; HWU, W-m; JOHNSON, RE; KALE, L; KUMAR, R, et al. Parallel computing research at Illinois: The UPCRC agenda. *Urbana, IL: Univ. Illinois Urbana-Champaign*. 2008.
28. LOSHIN, D. High-Performance Business Intelligence. In: *Business Intelligence*. 2013, pp. 211–235. Available from DOI: 10.1016/b978-0-12-385889-4.00014-4.
29. SITARAM, D.; MANJUNATH, G. Paradigms for Developing Cloud Applications. In: *Moving To The Cloud*. 2012, pp. 205–253. Available from DOI: 10.1016/b978-1-59749-725-1.00005-6.
30. VITOROVIĆ, A.; TOMAŠEVIĆ, M. V.; MILUTINOVIĆ, V. M. Manual Parallelization Versus State-of-the-Art Parallelization Techniques. In: *Advances in Computers*. 2014, pp. 203–251. Available from DOI: 10.1016/b978-0-12-420232-0.00005-2.
31. ALI, Syed Muhammad Fawad; WREMBEL, Robert. From conceptual design to performance optimization of ETL workflows: current state of research and open problems. *The VLDB Journal*. 2017, vol. 26, no. 6, pp. 777–801.
32. KOTLÁŘ, Robert. *Datový sklad ČVUT - způsoby datové integrace*. 2017. Master’s thesis. Czech Technical University in Prague, Faculty of Information Technology.
33. MAKARA, Adam. *Návrh a implementace klientské části a rolí systému Reports*. 2022. Bachelor’s thesis. Czech Technical University in Prague, Faculty of Information Technology.
34. *Pentaho Data Integration* [<https://sourceforge.net/projects/pentaho/>]. 2015. Accessed: April 23, 2023.
35. THOMAS J., Tatarczak E. *Airflow Survey 2022* [online]. 2022. [visited on 2023-04-24]. Available from: <https://airflow.apache.org/blog/airflow-survey-2022/>.

BIBLIOGRAPHY

36. APACHE SOFTWARE FOUNDATION. *Apache Airflow Documentation* [<https://airflow.apache.org/docs>]. 2023. Accessed: 19 April 2023.
37. *Airflow* [<https://airflow.apache.org/>]. 2023. Accessed: April 23, 2023.
38. HARENSLAK, Bas P; RUITER, Julian de. *Data Pipelines with Apache Airflow*. Simon and Schuster, 2021.

List of Acronyms

- 3NF** Third Normal Form
- BI** Business Intelligence
- CTU/ČVUT** Czech Technical University in Prague (Czech: České vysoké učení technické v Praze)
- DBMS** Database Management System
- DW** Data Warehouse
- EDW** Enterprise Data Warehouse
- ER** Entity-Relationship
- ETL** Extract, Transform, Load
- EZOP** Electronic Study Support System (Czech: Elektronická Základna Odborných Prací)
- FIFO** First In, First Out
- FIT** Faculty of Information Technology
- IDL** Integrated Data Layer
- JSON** JavaScript Object Notation
- KOS** Study Components (Czech: Komponenta studium)
- OODBMS** Object-Oriented Database Management System
- ODBC** Open Database Connectivity
- OLAP** Online Analytical Processing

A. LIST OF ACRONYMS

PL/pgSQL Procedural Language/PostgreSQL

POC Proof of Concept

SCD Slowly Changing Dimensions

SID System Identifier

SQL Structured Query Language

UI User Interface

VIC Computing and Information Centre (Czech: Výpočetní a Informační Centrum)

Apache Airflow Screenshots

This appendix presents a series of screenshots showcasing the implementation of the ETL processes in Apache Airflow. These images provide a visual representation of the developed DAGs and tasks, highlighting the organization and parallelization of the ETL processes. In addition to the screenshots, a README file is provided in the attachments, which includes detailed instructions on the installation and execution of the developed ETL solution using Apache Airflow.

B. APACHE AIRFLOW SCREENSHOTS

The screenshot displays the Apache Airflow DAG List interface. At the top, the time is 17:05 CEST (+02:00) and the user is KZ. The navigation menu includes Airflow, DAGs, Datasets, Security, Browse, Admin, and Docs. The main content area is titled 'DAGS' and features a search bar, a filter by tag, and an auto-refresh button. Below the search bar, there are four DAGs listed, each with a status indicator (Active or Paused), an owner, a schedule, a last run time, a next run time, a recent tasks count, and a links column. The DAGs are: ezop_dag (Active, 4 tasks), grades_dag (Active, 12 tasks), kos_dag (Active, 6 tasks), and usermap_dag (Active, 16 tasks). The recent tasks counts are: 4, 1, 3, and 2 respectively. The interface also shows a pagination bar at the bottom indicating 'Showing 1-4 of 4 DAGs' and a footer with version information: Version: v2.5.1, Git Version: .release:2.5.1+49867b60b6231c1319969217bc619177cf9829.

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks	Actions	Links
ezop_dag	zolickri	1	00***	2023-05-06, 17:03:39	2023-05-06, 02:00:00	4	▶	...
grades_dag	zolickri	12	00***	2023-05-06, 17:00:17	2023-05-06, 02:00:00	1	▶	...
kos_dag	zolickri	6	00***	2023-05-06, 17:00:42	2023-05-06, 02:00:00	3	▶	...
usermap_dag	zolickri	16	00***	2023-05-06, 17:01:14	2023-05-06, 02:00:00	2	▶	...

Figure B.1: Apache Airflow – DAG List with Tasks Running [37]

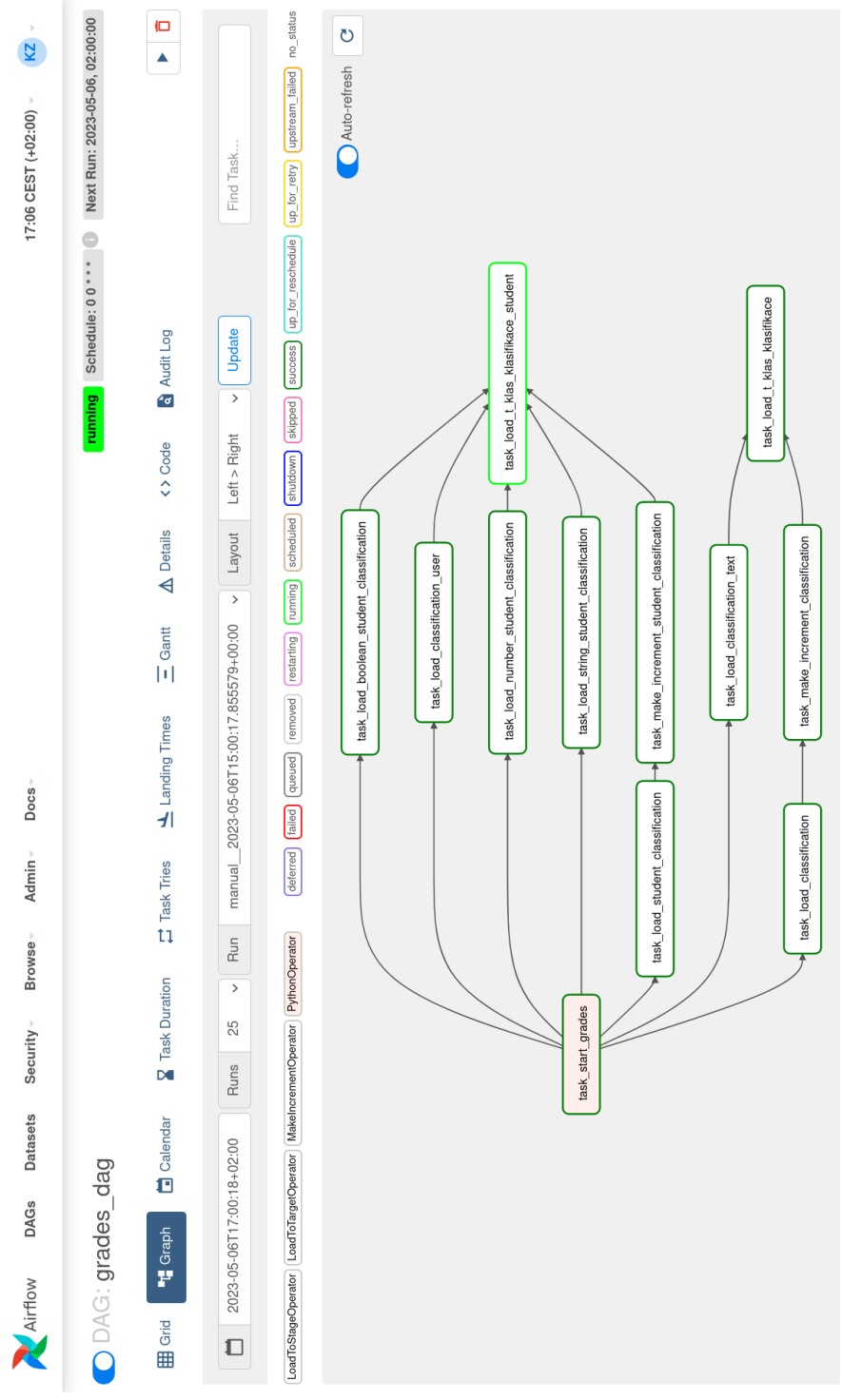


Figure B.2: Apache Airflow – Grades DAG [37]

B. APACHE AIRFLOW SCREENSHOTS

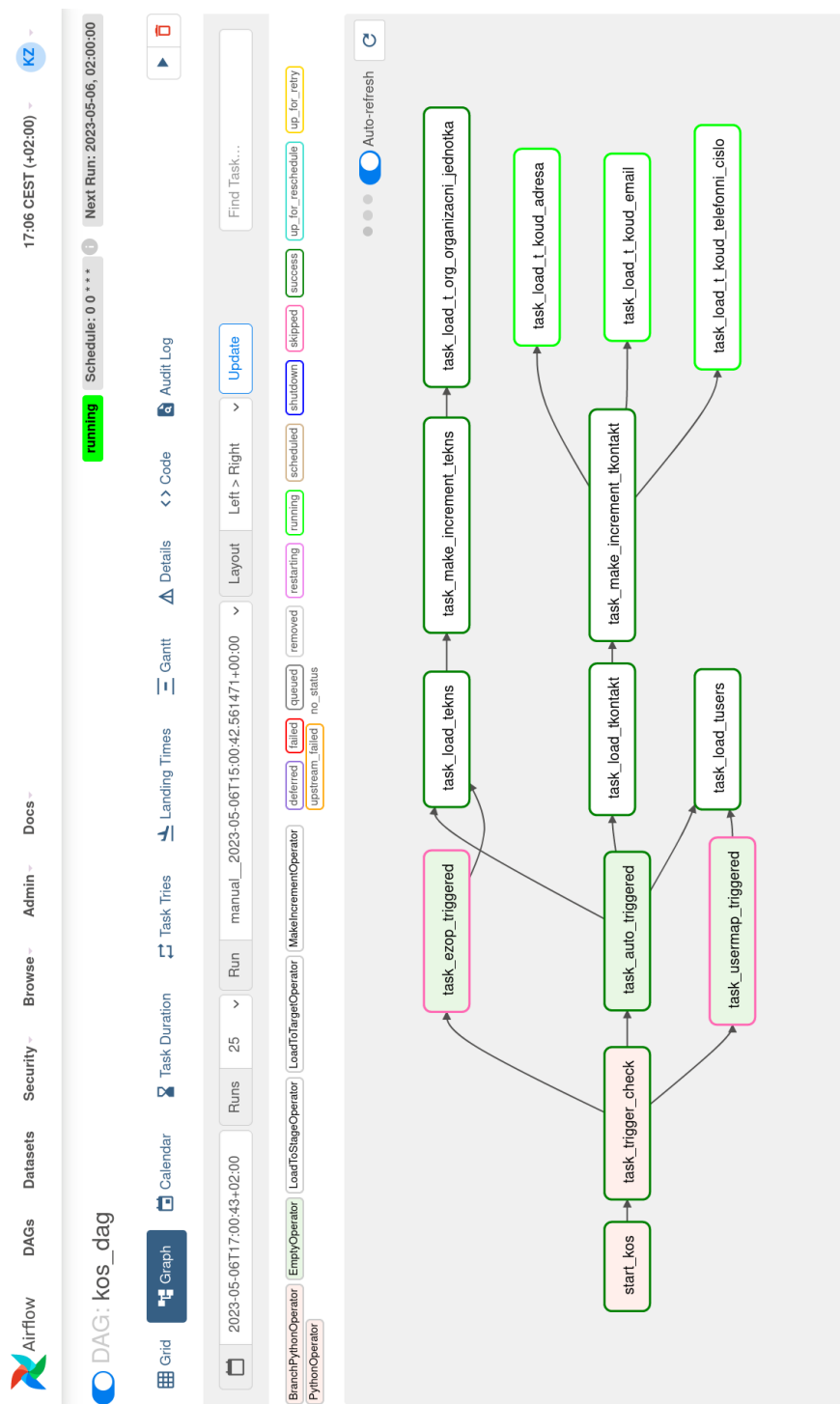


Figure B.3: Apache Airflow – KOS DAG [37]

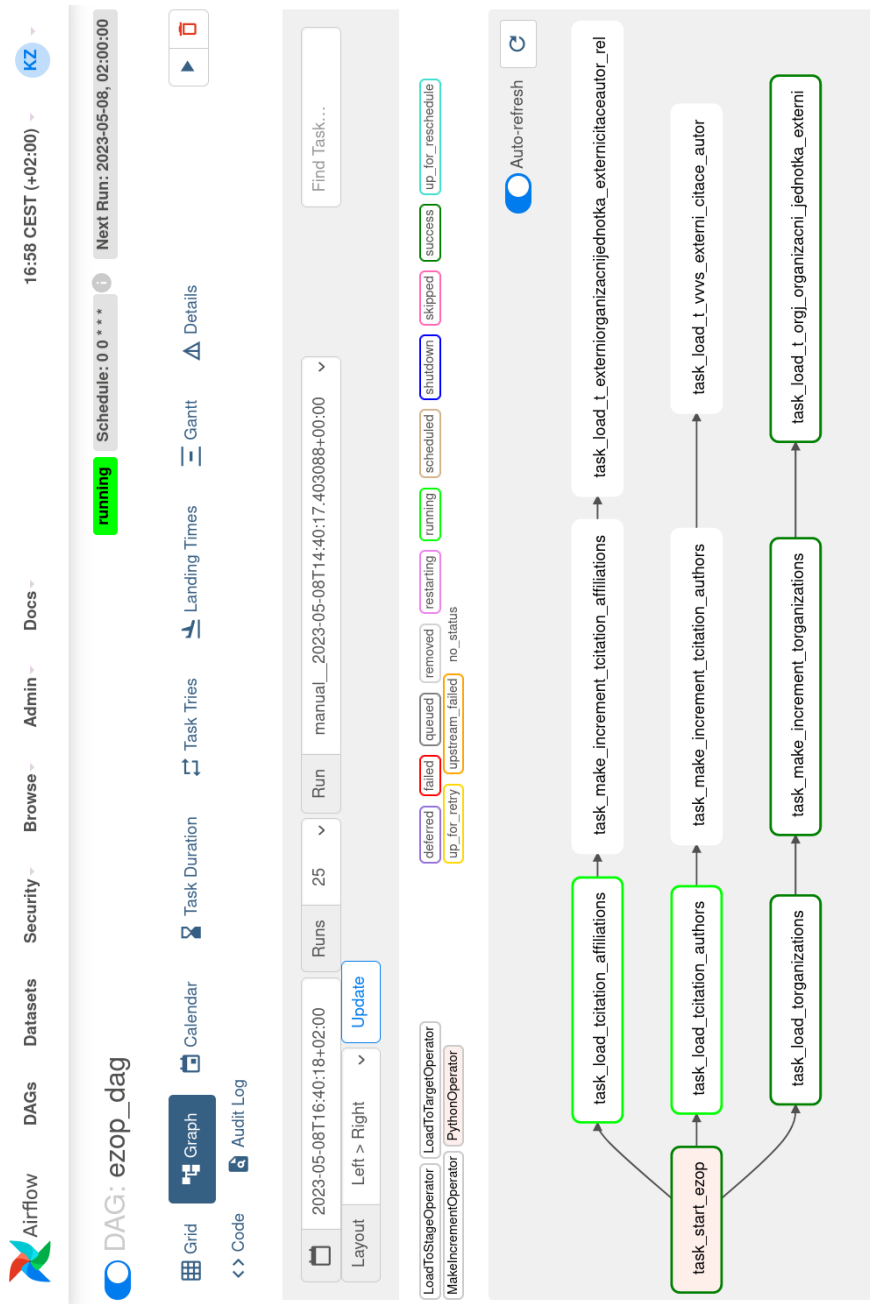


Figure B.4: Apache Airflow – EZOP DAG [37]

B. APACHE AIRFLOW SCREENSHOTS

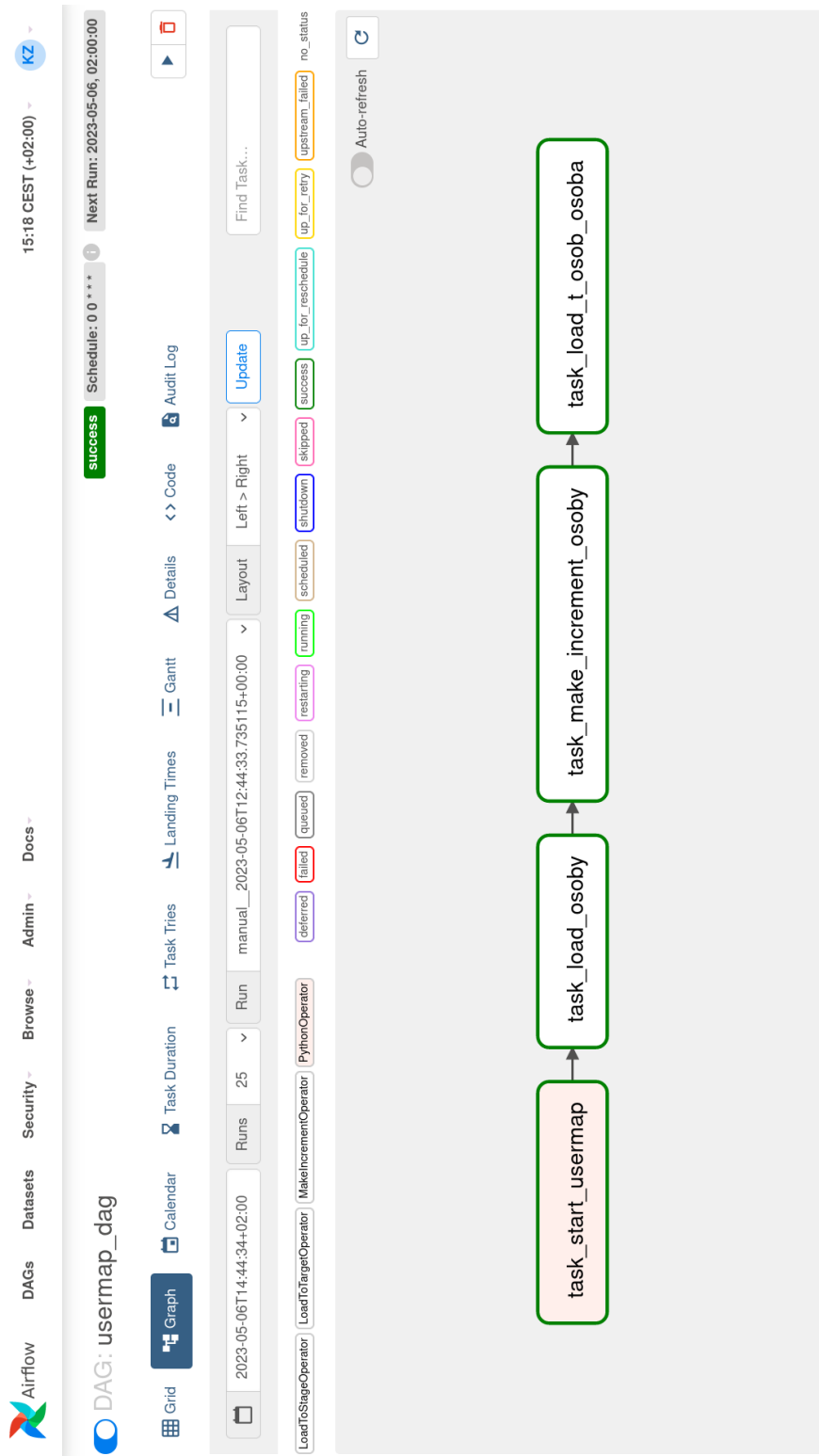


Figure B.5: Apache Airflow – Usermap DAG [37]

09/05/2023, 16:59:24 17:00 CEST (+02:00) KZ

09/05/2023, 16:59:24 25 All Run Types All Run States Clear Filters

Auto-refresh

Run DAG grades_dag / ► 2023-05-06, 02:00:00 CEST / task_make_increment_student_classification

Task Instance Details Rendered Template Log XCom List Instances, all runs Filter Upstream

Details Logs

(by attempts)

Task	Duration	Status
task_load_number_student_classification	00:00:00	Success
task_load_string_student_classification	00:00:00	Success
task_load_student_classification	00:00:00	Success
task_make_increment_student_classification	00:00:50	Success
task_load_t_klas_klasifikace	00:00:00	Success
task_load_t_klas_klasifikace_student	00:00:00	Success

task_make_increment_student_classification Status: success Started: 2023-05-06, 17:00:51 CEST Duration: 00:00:50

```

[2023-05-06, 17:02:38 CEST] [taskinstance.py:1300] INFO - Executing <Task(MakeIncrementOperator): task_make_increment_student_classification> [standard_task_runner.py:82] INFO - Started process 2746 to run task
[2023-05-06, 17:02:38 CEST] [standard_task_runner.py:82] INFO - Running: ['airflow', 'tasks', 'run', 'grad
[2023-05-06, 17:02:38 CEST] [standard_task_runner.py:83] INFO - Job 1627: Subtask task_make_increment_stud
[2023-05-06, 17:02:38 CEST] [task.command.py:386] INFO - Running <TaskInstance: grades_dag.task_make_incre
AIRFLOW_CTX_DAG_OWNER=zo1ockr1
AIRFLOW_CTX_DAG_ID=grades_dag
AIRFLOW_CTX_TASK_ID=task_make_increment_student_classification
AIRFLOW_CTX_EXECUTION_DATE=2023-05-06T15:00:17.855579+00:00
AIRFLOW_CTX_TRY_NUMBER=1
AIRFLOW_CTX_DAG_RUN_ID=manual_2023-05-06T15:00:17.855579+00:00
[2023-05-06, 17:02:38 CEST] [base.py:73] INFO - Using connection ID 'dvs' for task execution.
[2023-05-06, 17:02:41 CEST] [logging_mixin.py:137] INFO - State Flags for table si_grades.student_classifi
[2023-05-06, 17:02:41 CEST] [base.py:73] INFO - Using connection ID 'dvs' for task execution.
[2023-05-06, 17:02:41 CEST] [logging_mixin.py:137] INFO - Calling inc_find_modified_in_pre_stage_schema_ta
[2023-05-06, 17:02:54 CEST] [logging_mixin.py:137] INFO - State Flags M for table ps_grades.student_classi
[2023-05-06, 17:02:55 CEST] [logging_mixin.py:137] INFO - State Flags N for table ps_grades.student_classi
[2023-05-06, 17:02:57 CEST] [logging_mixin.py:137] INFO - State Flags D for table ps_grades.student_classi
[2023-05-06, 17:02:57 CEST] [taskinstance.py:1318] INFO - Marking task as SUCCESS. dag_id=grades_dag, task

```

Figure B.6: Apache Airflow – Logs of Grades DAG [37]

Contents of Digital Attachment

readme.md	the file with attachment contents description
src	the directory containing the source code
├─ airflow.....	the directory containing the DAGs and transformations
│ source code	
└─ readme.md.....	the file containing installation and execution guidance
src_thesis.....	the directory of \LaTeX source codes of the thesis
└─ BP_Zolocheskaia.2023.pdf	the thesis text in PDF format