JOHANNES KEPLER
UNIVERSITY LINZ

Author
Katharina Sternbauer, **BSc**

Submission
Institute for Application-oriented Knowledge Processing (FAW)

Thesis Supervisor
A.Univ.-Prof. DI Dr. Josef Küng

February 2025

Adaptive ETL processing for Business Intelligence applications with Data Warehouses

Master's Thesis
to confer the academic degree of
Diplom-Ingenieurin
in the Master's Program
Computer Science

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at

Abstract

Data is everywhere, and the more we digitalize our life and work, the more data our actions generate and tell about us. Capturing this data, cleaning, storing and analysing it has become a vital task of many companies. They do so not to be unique, but to have an edge over their competitors, or simply compete with them, in a more complex, globalized market than it was in the last century. The field of business intelligence aims to address these challenges. Without solid data sources, however, the believed insights are incomplete at best, or devastating for a company, in the worst case.

Large amounts of data are typically stored in an organized way, very often a data warehouse or its most popular variant, the less-structured data lake. A cornerstone to an effective data storage is an efficient loading process into it, with data stemming from various source systems. The standard process to do this, which comes in many variations, adaptations and various commercial tools, is ETL, short for Extract, Transform, Load. It is a powerful set of techniques, stemming from decades of experience, to continuous collect and store data as efficiently as possible and enable meaningful analytics to be performed on the captured and processed data.

Effective ETL processing is an exhaustive task to set up, configure and maintain, even when supported by software frameworks and specialized tools. Especially when new data sources are added to an existing system, non-automated and only hard-to-automate tasks such as credential management, data mappings, necessary transformations, code scripting, and much more take up a lot of time. They can be tedious to perform regularly.

While this thesis does not aim to, nor can it, solve all the above-mentioned tasks, it does make a solid attempt to give the basis for automated ETL code generation. When done manually, this step needs to combine a lot of information from multiple areas, like the before mentioned mappings and transformation, which is complicated. The thesis proposes a way within the scope of a domain to automate this step and therefore reduce the potential of errors. Each step, such as transformation, mapping, can be looked at independently by the ETL engineer, while not having to worry about the complex task of combining them manually.

This thesis' system is composed of multiple components, such as a data storage for information about the ETL processing and the code generator software. When ingested with the information on how data is moving between the sources and target destination, it automates the code generation step and provides the user with debug-able script files. These basic components may be embedded into a larger software system, which handles the data ingestion into the generator system and the ETL script execution, to, in the end, aid the engineer with their ETL processes.

Acknowledgement

I am deeply grateful to have had the opportunity of working with my supervisor, Professor Josef Küng. He has not only provided ongoing support for basically any question I ever had surrounding this thesis, he also helped me to continue this thesis when it was on the brink of collapse. I wouldn't be able to write these lines if he hadn't believed in my work and the idea of this thesis.

This thesis originally started in cooperation with Primetals Technologies Austria, and for the opportunity to work on this subject with them, I am grateful. It would not be this interesting of a topic if not for them. Special thanks to Helmut Ortmayer, Benjamin Skall and Daniel Fuchshuber, whom all were very supportive and appreciative of my work, even when it didn't work out as we expected it to.

Studying is not just another step in one's educational journey, which to me never truly ends, but also a very formative time in a person's life. Foremost, being able to study, and especially to focus on my passion, wasn't always an easy task to do, but especially not without the support from my parents. Second, I have met many passionate people during my studies, all of whom have shaped me and influenced me in one way or another. I would like to especially mention Michael Leichtfried, Alrun Lindner, Julia Walchetseder, and Elisa Fischer. You have not only been fellow students, you have been friends and advisors.

When I started my study, I was still living at home with my parents. And now, as it ends, I have moved out, started a job already and got a lot more independent. One could say, I really grew up. I could not have done that without support during this time from my parents. Irmi, Wolfgang, thank you for all the support throughout the years, both emotionally and financially.

Lastly, my biggest thanks and my heart rightfully goes to my partner Vanessa. When this thesis nearly failed, and I wasn't even sure that this thesis could ever be done, let alone formed into code and this very document, you supported me, lifted me up and snatched the therapy cat for me (even though I have pretty sure the cat wasn't that passionate about it). You didn't give up on me. Thank you.

> You will see. You will all see!
>
> — The inventor of glasses

Contents

Images

Tables

Code

Abbreviations

ETL	Extract, Transform, Load
DWH	Data Warehouse
KB	Knowledge Base
DB	Database
RDB	Relational Database
RDBMS	Relational Database Management System
CRUD	Create, Read, Update, Delete
REST	Representational State Transfer
API	Application Programming Interface
KPI	Key Performance Indicator
DTO	Data Transfer Object
SCD	Slowly Changing Dimension
RDF	Resource Description Framework
OWL	Web Ontology Language

1. Introduction

1.1. Background

Data warehouses are essential for supporting decision-making processes by integrating data from various sources into a unified, subject-oriented, and time-variant repository [1]. The ETL process, which involves extracting, transforming, and loading data, is critical for populating these warehouses. It, as many other complex systems, faces several challenges, especially with its data's variety, as well as the heterogeneity [2]. The variety of data refers to the different formats and structures of data coming from multiple sources. Data heterogeneity arises from the structural and semantic differences between data sources. Traditional ETL processes also require significant human involvement and most often are costly in terms of time and resources [3]. Additionally, the complexity of ETL workflows and the lack of precise metadata can hinder automation.

To address these challenges, semantic technologies, such as ontologies, offer a possible approach [1]. Ontologies provide a formal and explicit specification of a shared conceptualization, offering a vocabulary to describe a specific domain and its concrete elements within [4], [5]. By integrating ontologies into the ETL process, data integration may be enhanced by providing common definitions and relationships of concepts. Ontologies capture the semantics of a domain, are an applicable tool to resolve semantic-related issues, as well as facilitate data integration [4], [6]. Furthermore, ontologies can enable automated ETL design by specifying the semantics of data sources and the data warehouse schema, leading to automated transformations [2], [4], [7].

Semantic technologies also provide more flexibility in data integration by mapping data from source databases to ontology classes [4]. Specifically, the use of semantic web technologies in the development of data warehouses can improve the management of data through ontology-based data mapping [6]. Moreover, ontologies support the construction of a unified model that integrates the various business requirements. Semantic technologies can also address heterogeneity problems, both semantic and structural, by providing formal specifications of inter-element relations [2].

1.2. Problem statement

This thesis focusses on automated SQL generation for ETL processing on a practical level. It both discusses a possible solution, implements a working prototype, a script generator, and evaluates the same.

The more data sources there are for a data warehouse system, the more informative analysis on the stored information becomes. With an increasing number of sources comes a dramatically increasing workload on the engineers to handle all the ETL processing and adapt existing solutions to new requirements by hand. This still holds true even when deploying more complex ETL frameworks, as the SQL code itself still needs manual configuration and editing. This becomes costly and difficult when working with many external partners for such a data warehouse.

The derived goals of this thesis are therefore:

- the implementation of a SQL script generator for ETL processing
- the design and development of a knowledge storage about the meta information required to perform the ETL-generation processes
- evaluate and discuss the implementation.

Explicit non-goals are the automated metadata extraction from ETL sources and targets, as well as the initial support for all varieties of different SQL dialects and database system. The program will also not offer a fully-fledged API or user interface for out of the box seamless integration with established frameworks.

1.3. Structure

The first chapters discuss all the basics and theoretical background to the described problem and adds further context to the concepts. This is followed by the presentation of the design, implementation, and evaluation. Last, a conclusion is given, alongside an outlook on further research and development topics.

In *Chapter 2*, we provide the theoretical background on the many terms and concepts that surround the topic. *Chapter 3* introduces related works, including state-of-the-art works, projects with similar ideas and related technologies in action. The next chapter, *Chapter 4*, shows the proposed solution and the considerations behind the chosen design. Implementation and elaborations on it are to be found in *Chapter 5*, whereas *Chapter 6* evaluates the implementations and discusses possible extensions. We conclude the thesis with *Chapter 7*, providing both a summary of the previous work and giving an outlook on future work and possible usage of the implementation.

2. Concepts and Definitions

As with many things, there isn't only one correct definition for any of these terms. We will therefore often consider more than one perspective, combining different aspects into a given explanation.

2.1. Data Warehouse

A data warehouse (or DWH, in short form) can be described as a "subject-oriented, integrated, time-variant, and non-volatile collection of data" [8] to support decision-making processes [6]. This comes with some crucial characteristics. First of, the data is organized in a subject-oriented manner in fact tables and dimensional tables [1], more on them in an upcoming section, for its decision-making purpose, meaning the structuring revolves around key business subjects and not operational processes [9].

Second, data comes from various differing sources and is "re-stored" [10] in the data warehouse, where inconsistencies must be resolved to establish a unified view [11]. A warehouse can track time-variant data and provides a non-volatile storage, containing the historical data for analysis [9]. The data inside the warehouse is populated using ETL processes [6].

Quantitative measurements of business-related events are stored in fact tables. The dimensional tables, on the other hand, provide context to the facts, most often descriptive attributes to support filtering and grouping [12]. This so-called dimensional model comes in various styles, such as a star schema, snowflake schema or a multidimensional OLAP cube [6], depending on the type of database system used for the data warehouse. Typically, OLAP cubes are built from star schemas [12].

2.2. ETL

The acronym **ETL** stands for the three-phase process of **E**xtract, **T**ransform, **L**oad and is a central component in data warehousing. This flow of data is considered to be the most important step in the data warehouse lifecycle [13]. It does involve gathering the data from various sources, transforming it to meet the required, unified format and loading the processed data into the data storage [2].

There are many ways to perform this complicated procedure, the most notable laid out by Kimball [14], who has also been very active with designing data warehouses themselves [9]. The process is often tool supported [15], much contributed by the necessity to accommodate many forms of sources [5]. These include internal ones like other databases, XML, CSV files, and external sources, web applications or social media [2].

The three parts of ETL each have their individual characteristics. During extraction, the previously mentioned different types of sources have to be identified and, more influential, the relevant content extracted [10], [16]. Data extraction can be done in full or incremental to already existing data in the destination [2]. The items are transformed into a common format, guided by a ruleset based on business demand [10], [16]. The transformations include aggregations, joins, data conversion and filtering, typically completed in the staging area [2]. Last, the transformed data is propagated to the data warehouse, or data mart [16].

2.3. Business Intelligence

The term "Business Intelligence" was coined in the mid-1990s and used as an umbrella term referring to "applications, infrastructures, tools, and practices used to improve and optimize decision-making and performance" [1], done so by analysing available data sources and retrieving information from them. In essence, these tools produce actionable insights from raw data [3], enabling decision-making on a strategic level [17].

Business Intelligence strategies often use a data warehouse as its central data storage, accessed by dedicated BI tools [1], [6]. The ETL process enables the data ingestions into the data warehouse and therefore represents a further essential tool for BI applications [3], [16]. It could be concluded that Business intelligence builds on top of the concepts of data warehouses and ETL processing and processes the data to fit the user's requirements [1], [2].

2.4. Semantics

The term semantics, in the area of Computer Science, refers to the meaning of data in a domain[4], and not its syntax [18]. For an item to be semantically sound, it requires a formal, explicit and shared definition, often achieved through the use of an ontology [7].

As an example, the Java snippet `int x = 4` does assign the numeric value '4' to a variable named `x` from a program's perspective, but does not say anything whether 4 is a lot of what it means. By adding the semantic context of `x` stating the number of children a person has given birth to, this would typically mean plenty of children in most modern western cultures. However, in countries where the average person has more children, 4 children could also be perceived as perfectly inline with the average and 200 years ago even as relatively few descendants. This difference in perception arises the issue of semantic heterogeneity, where the sole value is interpreted in different ways, depending on other factors in its environment [17]. Therefore, ontologies aim to construct a common vocabulary used by all parties evolved, giving consistent meaning to the raw data [1], [4].

Furthermore, the semantics do not only define the meaning and add context to a single item, but also describe relationships between different terms within the domain [4]. Semantics can also be used to represent (dis)similarity between elements [15].

Semantics are useful with data integration from heterogeneous sources due to the provided meaning of data [19]. This holds especially true for transformation, to appropriately map from different sources to a common target [15] and aid automated ETL processes [4], [5]. Adding semantics to data may also be beneficial for data analysis, enhancing analytical capabilities for example, although this comes with its own set of challenges [6].

The term "Semantic Web" references an extension to the current web by adding machine-readable semantic metadata [6]. To some authors, the Semantic Web is the practical implementation of semantics [1]. With this technology, some authors aim to further automate and elevate the ETL process [1], [5], [17].

2.5. Ontology in Computer Science

An ontology in the field of computer science can be described as a "formal and explicit specification of a shared conceptualization" [5], i.e. an abstract model containing the fitting concepts of a given topic [1]. Due to the explicit specification, these concepts and the constraints on them are clearly defined by shared vocabulary, to avoid semantic ambiguity (see *Section 2.4*) [4]. When an ontology is only concerned about covering knowledge from a specific area, labelled the domain, and specialises in its concepts, it is also called a "Domain Ontology" [10].

In the context of working with data, ontologies enable data exchange based on the meaning [17] by said standardized vocabulary taxonomy of concepts in a domain [19]. Typically, the knowledge representation comes in terms of classes, properties, and relationships [4]. Ontologies and semantics turn out to be an essential aid in integrating heterogenous data from multiple sources during ETL processing.

There exists more than one way and language to define an ontology, yet there is a rather popular choice. The 'Resource Description Framework' (RDF) builds the foundation [1]. It provides a machine-readable XML format as triplets with subject, predicated (or property), and object [1], [17]. The 'Web Ontology Language' (OWL) is built on-top of RDF and provides a richer vocabulary and increased expressiveness [1]. OWL often uses the RDF syntax for ontology creation and is an W3C standard [4]. Combining OWL with a reasoning engine enables automated knowledge inference.
As it will be shown later in *Chapter 4*, an ontology can also be stored in other forms as well, in this case representing the domain knowledge in and by a relational database.

Ontologies may be used, due to their semantic enrichment of data, to help with ETL processing in multiple ways. One presented approach uses an ontology inside the database's metadata to map data records to ontology classes, which then aids with record selection and rule creation for data transformation [10]. Another way would be to fully automate ETL transformation creation from an ontology [2]. Ontologies help to mitigate semantic conflicts and ambiguities [4]. Within Business Intelligence, ontologies can support analytical functionalities by providing semantic information [6] and improve the "quality of insights in BI systems" [1].

2.5.1. Knowledge Graph

A 'Knowledge Graph' is a representation of information as a network of nodes and relationships between them, therefore in a graph format [11]. In "Towards a Definition of Knowledge Graphs", which also discusses the ambiguity of the term rather well, the authors define it as follows:

> A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge.
>
> — Towards a Definition of Knowledge Graphs [20]

The nodes represent the entities / concepts, and the edges are the relations between the elements. An ontology provides the underlying structure for a knowledge graph [21] and introduces the semantic layer, representing the meaning of contained data. Furthermore, this enables inferencing and reasoning on the information by a dedicated reasoning engine [19], [20].

3. Related Work

As discussed in *Chapter 2*, adaptive extraction, transformation, and loading are complicated tasks with many differing aspects. For this thesis, both storing meta information about ETL, the semantics of what it does, and script generation are the main focus. There are, however, numerous further perspectives which are relevant to painting a more complete picture of ETL processing, especially when in overlap with semantics and ontologies.

For added structure, we split overarching themes into separate sections. Each paper finds its space in a subsection, which discusses its content in a short, yet precise manner.

3.1. Literature review

The paper “Incorporation of Ontologies in Data Warehouse/Business Intelligence Systems - A Systematic Literature Review” [1] examines how semantic web techniques, especially ontologies, enhance Data Warehouse (DWH) and Business Intelligence (BI) systems by addressing semantic heterogeneity, improving interoperability, and enriching analytical capabilities. Using standards like the Web Ontology Language (OWL), ontologies are applied in key DW/BI lifecycle tasks, including dimensional modelling, ETL processes, and BI application design. The study highlights that integrating ontologies into DW/BI systems bridges traditional database methods and semantic web advancements, improving both efficiency and analytical depth.

In dimensional modelling, the literature review finds ontologies to simplify schema design and enable semantic integration across diverse data sources, aligning with data warehousing’s aim to unify data into a consistent format. Ontologies are expected to enhance ETL processes by automating mappings, resolving inconsistencies, and representing workflows semantically, augmenting traditional ETL methods with greater precision and interoperability.

For BI application design, ontologies provide semantic reasoning and allow users to query and analyse data more effectively. This builds on the strengths of data warehouses in multidimensional analysis while adding inference capabilities for more profound insights. Ontology-based frameworks, like RDF Data Cubes, extend OLAP capabilities and foster knowledge extraction.

3.2. Semantics

3.2.1. State of the art

“Semantic ETL – State-of-the-art and open research challenges” [15] shows a comprehensive survey of data integration approaches combining traditional ETL processes with semantic web technologies. The paper examines how semantic understanding elevates the traditional ETL pipeline, while identifying current limitations and future research directions.

Traditional ETL tools excel at extracting data from various sources, transforming it according to business rules, and loading it into data warehouses. However, these tools lack semantic understanding of the data they process. Semantic ETL frameworks like Karma and SETL address this limitation by incorporating ontologies and semantic web standards, enabling semi-automatic mapping of structured sources to semantic web formats while preserving meaningful relationships between data elements.

The relationship between semantic ETL, traditional ontologies, ETL processes, and data warehouses shows both similarities and complementary aspects. Like traditional ontologies, semantic ETL uses formal representations of domain knowledge, but applies them specifically to guide data transformation and integration rather than just knowledge representation. The core ETL process remains rather unchanged, but gains semantic understanding that helps automate mappings and ensure consistency across sources. Similar to how data warehouses consolidate information into unified schemas, semantic ETL creates semantically enriched data stores that preserve both data and its relationships.

The work finds that major challenges persist in automating semantic mapping processes and handling data quality issues. It also concludes that these are areas where traditional ETL tools have struggled and still face major obstacles. The integration of semantic technologies with established ETL practices represents a promising direction for addressing increasing data integration complexity in modern enterprises, though full automation remains an open research challenge.

3.2.2. Semantic approach to ETL

The paper “A semantic approach to ETL technologies” [16] presents a tool that enhances ETL processes by combining semantic integration techniques with data analysis for automated mapping and transformation of data sources in data warehousing. The authors extend two existing systems - MOMIS for data integration and RELEVANT for data analysis - to support semi-automatic definition of semantic mappings between source and target schemas, while also enabling semantic transformation of attribute values.

The tool addresses key ETL challenges through semantic enrichment of schema descriptions, automated cluster generation of related elements, and transformation function identification. In the work’s case study with a beverage and food logistics company, the system demonstrated effective automation of mapping identification and data transformation, achieving approximately 85% accuracy compared to manual expert mappings.

The approach shares significant commonalities with other ontology-based data integration and traditional ETL systems, while introducing novel semantic elements. Like other ontology-based systems, it leverages a reference ontology, in this case “WordNet”, a “large lexical database of English” [22][*], for annotating schema elements and establishing semantic relationships. However, it extends this by combining schema-level semantics with instance-level analysis through clustering techniques. The system’s transformation capabilities parallel traditional ETL tools with standard functions like `CONVERT` and `AGGREGATE`, but innovates through a function that creates semantic clusters of related attribute values. This semantic categorization enables a more flexible data warehouse population while preserving original values for detailed analysis.

The work attempts to bridge the gap between semantic integration approaches, focused purely on schema mapping, and traditional ETL systems concerned primarily with data transformation. It demonstrates how semantic technologies may enhance the data warehousing process while maintaining compatibility with existing ETL frameworks and practices.

3.2.3. Semantic Web and ETL

The paper “Designing ETL Processes Using Semantic Web Technologies” [23] explores the use of Semantic Web technologies to enhance the design of ETL processes for data warehouses. The paper highlights the challenges posed by structural and semantic heterogeneity in data sources. It proposes using ontologies to automate the construction of ETL workflows, which can significantly improve efficiency and reduce errors at the same time. It therefore attempts roughly the same things as this thesis.

The authors describe how to construct an ontology to model the domain of discourse, based on the schemas of the data warehouse and data sources. This ontology helps resolve semantic conflicts and identify the necessary transformations for data integration. By formally and explicitly specifying the semantics of the data source schemas and the data warehouse schema, the ontology-based approach facilitates the automation of ETL workflow creation.

The paper outlines a method to automatically derive the required ETL transformations based on the constructed ontology. This includes identifying the relevant data sources, establishing inter-attribute mappings, and specifying the necessary transformations. The authors argue that using Semantic Web technologies can significantly automate the ETL workflow creation process, making it more efficient and less error-prone.

The work highlights the similarities between ontologies, ETL, and data warehouses, as all three involve the integration and transformation of data from various sources to create a unified and coherent data model. Ontologies provide a structured framework for understanding the semantics of data, ETL processes handle the extraction, transformation, and loading of data, and data warehouses serve as the central repository for integrated data. By leveraging Semantic Web technologies, the authors show how

[*]https://wordnet.princeton.edu/

these components can work together to streamline the ETL process and improve data quality.

3.2.4. Semantic Framework

“A Domain Independent Framework for Extracting Linked Semantic Data from Tables” [21] discusses a toolkit designed to interpret and represent the intended meaning of tables as Linked Data. The framework aims to address the challenges posed by the vast amounts of information encoded in tables found in documents, on the Web, and in spreadsheets or databases. By using Semantic Web technologies, the framework seeks to make the implicit meaning of tables explicit in a semantic representation language like RDF.

The framework is grounded in graphical models and probabilistic reasoning to infer the semantics associated with a table. It leverages background knowledge from resources in the Linked Open Data cloud to jointly infer the semantics of column headers, table cell values (e.g., strings and numbers), and relations between columns. This inferred meaning is then represented as a graph of RDF triples, capturing the table’s meaning by mapping columns to classes in an appropriate ontology. The representation continuous by linking cell values to literal constants, implied measurements, or entities in the linked data cloud, and discovering or identifying relations between columns.

The paper highlights the usefulness of ontologies, as they provide a structured framework for understanding the semantics of data. By using Semantic Web technologies, the framework integrates all the components to streamline the ETL process and improve data quality. The approach ensures that the data’s intended meaning is preserved and made accessible for integration and search purposes.

3.3. Ontologies

3.3.1. Differences to Databases

The work “Ontology versus Database” [24] explores the differences and similarities between ontologies and databases. Starting with a brief history lesson, the document explains that ontologies originated as a philosophical discipline, focusing on adding meaning and comprehension, while the modern database evolved from file cabinets used for data storage, designed for efficient data retrieval.

A major difference lies in the way of knowledge representation. Ontologies use the Open World Assumption (OWA), allowing for incomplete information, whereas databases rely on the Closed World Assumption (CWA), where missing information is considered false. Ontologies are often built upon already existing representatives, using their design patterns for consistency, while databases are created from scratch, focusing on normalization to eliminate redundancy.

Despite these differences, both ontologies and databases aim to represent a segment of the real world, featuring entities (or classes) and attributes (or properties) to describe data. Both systems can impose restrictions to ensure data consistency and integrity. The document also discusses methods for transforming a database into an ontology and vice versa, highlighting the benefits of combining the strengths of both systems for enhanced data integration and interoperability.

The paper concludes by emphasizing the potential for communication and integration between ontologies and databases. By leveraging the structured framework of ontologies and the efficient data storage capabilities of databases, the integration can streamline data processes and improve overall data quality.

3.3.2. Domain Ontologies for ETL

"A Domain Ontology Approach in the ETL Process of Data Warehousing" [10] approaches the integration of a domain ontology into the ETL (Extract, Transform, Load) process to address data heterogeneity in data warehouses. The paper highlights the challenges posed by structural and semantic heterogeneity in enterprise information systems and proposes using an ontology to tackle these issues. By embedding it in the metadata of the DWH, data records can be mapped from the database to the corresponding ontology classes.

The authors describe the construction of a domain ontology based on domain standard documentation and database schemas. The created ontology helps to define the rules for data transformation and ensures that data is integrated semantically correct. The ETL process is guided by the ontology, which facilitates the identification of data sources, transformation of data formats, and cleansing of data. The paper presents a multi-level metadata model for ETL, which includes resource definition, domain dictionary, and mapping levels.

The document also discusses the practical application of this approach in a hospital data warehouse project. The results indicate that the ontology method plays a crucial role in data integration by providing common descriptions of concepts and relationships of data items. The use of domain ontology is said to enhance the flexibility and efficiency of data warehousing, making it a feasible approach for complex and data-intensive environments.

3.3.3. Ontology based ETL processing

The paper "Ontology based ETL process for creation of ontological data warehouse" [17] introduces a novel approach to streamlining ETL processes through ontology-based automation. The authors developed a methodology that uses a master ontology to guide and automate key ETL activities, and therefore addressing common challenges in data warehouse integration.

The proposed solution consists of four primary modules: meta-information extraction, meta-information addition, logical model generation, and physical model generation. This framework enables automated data integration while using ontologies to resolve heterogeneity issues across different data sources. Through semantic mapping and automation, the approach reduces the manual intervention required, as well as maintaining data consistency and meaning across diverse source systems.

The authors validated their methodology through a case study involving multiple data sources, including MySQL and PostgreSQL databases, and XML files. The implementation achieved 95% accurate data integration within a one-week timeframe. Their architecture employs a layered approach, encompassing data sources, information extraction, abstraction and generalisation, model generation, and information integration. This structure, combined with ontological mapping, hints the potential for semantic technologies to bridge the gap between heterogeneous data sources and target data warehouses while automating core ETL processes.

3.3.4. ETL processing for (semi)-structured data

The work titled "Ontology-Based Conceptual Design of ETL Processes for Both Structured and Semi-Structured Data" [5] focusses an ontology-based approach to facilitate the conceptual design of ETL. The paper highlights the challenges posed by structural and semantic heterogeneity in data sources and proposes using Semantic Web technologies to annotate data sources and the data warehouse. This approach aims to resolve heterogeneity issues by creating a suitable application ontology to annotate the datastores. A reasoner is then employed to infer semantic correspondences and conflicts among the datastores, suggesting conceptual operations for transforming data from the source datastores to the data warehouse.

The authors describe the use of a graph-based representation as a conceptual model for the datastores, allowing both structured and semi-structured data to be handled uniformly. The ontology-based approach includes creating an application ontology to model the domain and requirements, mapping and annotating the datastores, and using automated reasoning techniques to infer correspondences and conflicts. This method facilitates the identification of relevant data sources, the transformation of data formats, and the cleansing of data, ultimately improving the efficiency and accuracy of the ETL process.

The work talks about the similarities between ontologies, ETL processes, and data warehouses. By leveraging Semantic Web technologies, the authors demonstrate how these components can work together to streamline the ETL process and enhance data quality.

3.4. Dynamic ETL

The authors of the work “Dynamic-ETL: a hybrid approach for health data extraction, transformation, and loading” [25] outlines a novel method for harmonizing heterogeneous US health data to support clinical research networks. It faces many of the same hurdles as this thesis. The “Dynamic-ETL (D-ETL)”“ approach integrates concepts from ontologies, traditional ETL processes, and data warehousing. It addresses the challenges of transforming electronic health records (EHRs)—often stored in proprietary, fragmented formats—into standardized schemas such as the OMOP common data model. Similar to ontologies, D-ETL emphasizes semantic alignment, employing reusable and human-readable rules to map diverse healthcare terminologies. This alignment ensures consistency and interoperability, a foundational principle shared with ontology-based systems.

D-ETL’s structure mirrors traditional ETL pipelines, extracting data from diverse sources, transforming it to conform to a target schema, and loading it into a unified repository. Its emphasis on task automation through a rule-based engine, combined with manual editing when necessary, enables the handling of complex healthcare data requirements, such as resolving duplicate and conflicting records.

Implemented in the ‘ROSITA’ system, the paper claims the D-ETL system demonstrated scalability and adaptability, enabling partners to standardize and link EHR and financial claims data. It reduced technical barriers by allowing domain experts with limited technical knowledge to create transformation rules while maintaining flexibility for intricate operations. Though effective, the approach requires enhancements, according to the authors, including cross-platform compatibility and tools for error-checking. Overall, the authors faced many of the same difficulties and challenges as this thesis, however in a different field and with a dissimilar toolset.

3.5. Conclusion

The presented works provide many interesting and diverse approaches to ETL processing, with some addressing similar topics or problem statements, whereas others broaden the view on the many difficulties in this subject. Especially, works presented in *Section 3.3* state many of the same troubles and attempt similar approaches as this thesis does, however often with some differences in the used tools and scope. Nevertheless, they were more than once helpful for getting inspiration for our approach to adaptive ETL processing with ontologies, as presented in the next chapters.

4. Solution Design

We have already discussed the general aspects of adaptive ETL processing in *Chapter 2*, from which stems the main motivation for this thesis, and on the broader picture in *Chapter 3*, where elements related to the core topic are looked at. This chapter is dedicated to the ideas and attempts made to implement an adaptive code generator for ETL scripts, and how the whole system built to support these efforts.

4.1. Scenario

For a more profound understanding of what the system should solve, a scenario as described in the following paragraph is created. Imagine the Ministry of Education in Austria has the goal of getting more insight into the many universities located within the borders of the country. The federal IT department decides that the various requirements, including long-time data storage, support for analytical processing and pre-processed data for faster result computation, are best supported by a data warehouse. To start off, they are only interested in adding one location. Later on, the ministry plans to add the data of more and more of Austria's universities to their warehouse. The IT department must be very cautious with spending taxpayer's money, and therefore aim towards minimal configuration overhead when adding new data sources to cut the operating expenses.

4.2. Objectives

This scenario covers the general concepts and definitions from *Chapter 2* pretty well. Automated code generation for data transfer from a data source to the target data warehouse structure, based on information stored about these components, is the desired outcome. Nonmanual, or at least semi-automated knowledge deduction from databases, like data transformations, element mapping, joins, and so on would be a nice addition, but for the pure functionality of code generation, they are out of scope. For demonstration, it is sufficient to fill these records by hand, although some SQL scripting can support the process quite efficiently.

4.3. Requirements

To complete the set of goals, it is essential to derive some required attributes for the system. First off, it must be able to store information about multiple sources and access them independently of each other. The way the system stores this knowledge is best to be kept separate from both the data sources and the target. It must hold all information relevant to performing ETL, such as the structure of a single source and the targets. It must be aware of data mappings and transformations, how data is related with each other (expressed as joins in most relational databases) and under which conditions data must or must not be selected. An engineer should be able to manually add, modify, delete these entries in the database when needed and start the information extraction process, if present.

On the side of non-functional requirements [26], also commonly referred to as the '-ilities' of a system, generally favourable aspects such as performance, scalability, availability, and many more, take a back seat in this work. Due to the demonstrative nature of this work, the design is not specifically targeting them, but this does not mean, at all, that good practices are straight out ignored. Especially reusability and maintainability are of importance, such that the combination with other systems and extensions to the core functionality can be done with comparable ease.

4.4. System design

Following the boundaries on what is to achieve, it becomes clear that the proposed solution requires multiple interacting components. One of these elements is the semantic data storage, which will be called "KnowledgeBase", or KB for short. Furthermore, a code generation software is needed, which takes the information from the knowledge repository and turns it into valid SQL code. There should also be endpoints for the engineer to access and modify the data.

4.4.1. Knowledge encoding

The encoding of knowledge is separated into two parts. First off, the ontological knowledge of this scenario - for example students and courses in a university context - are stored by encoding them in the table layout of the adaptive ETL target. In this case, considering the data flow, this references the upstream database before the data warehouse. More on this in the upcoming sections. Second, the information specific to individual data sources, target and ETL operations is stored in KnowledgeBase.

4.4.2. Layout overview

Let us start off with a look at the completed system, and then break it down into its individual components. *Figure 1* shows the completed design of the system, including source systems, generation software, KnowledgeBase and data warehouse. The image has some naming that references the scenario outlined in *Section 4.1*. For example, the sources are labelled with the names, or the respective short forms, of different Austrian universities. It does also show some of the more specific elements of ETL processing, more on these in later on.

The next sections will introduce the various components of the system in more detail. More specific, it focusses on KnowledgeBase, what the intermediary TemplateDB between the sources and the data warehouse does, how the generation software comes into play and how the pieces interact with each other to form this bigger picture.

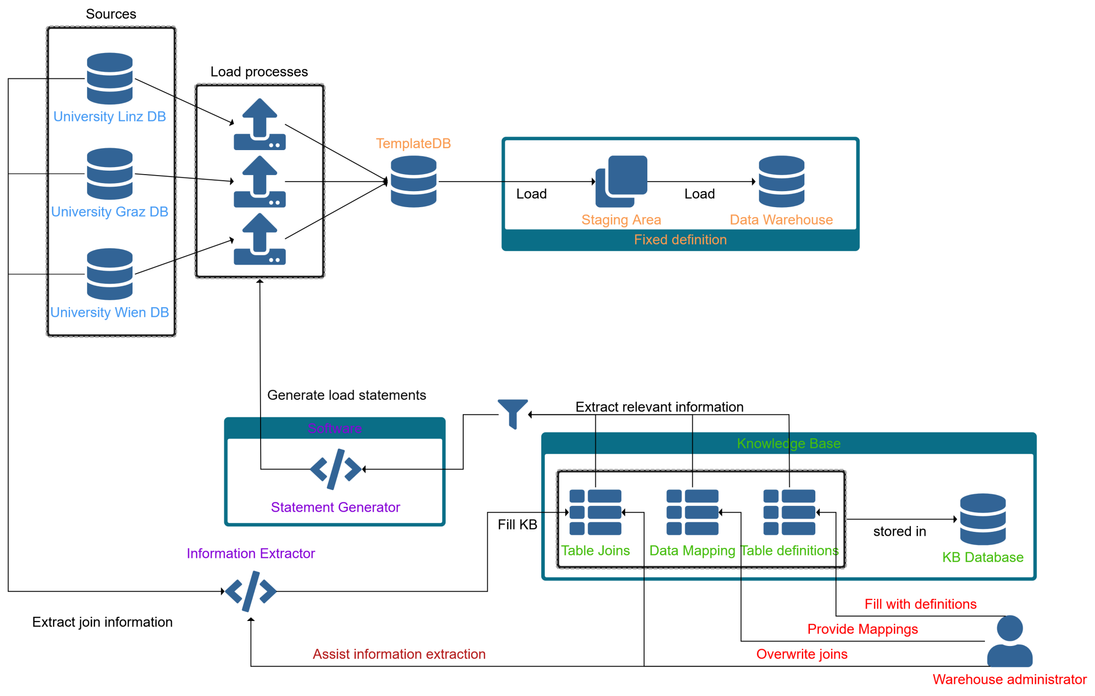


Figure 1: An schematic overview of system design.

4.4.3. TemplateDB

Ontologies need to both encode the domain knowledge of a system and actual knowledge for a concrete system. To put this into more precise terms, when looking at the scenario as laid out in *Section 4.1*, TemplateDB encodes the domain knowledge about universities, where things such as courses, students, grades, credits, locations and many more exists. The information about specific systems is then stored in KnowledgeBase.

Having the template component specified explicitly, helps tremendously with ETL processing. The fact that the domain knowledge is not encoded in the warehouse itself, which can be of various forms, but in an upstream database, simplifies the engineering

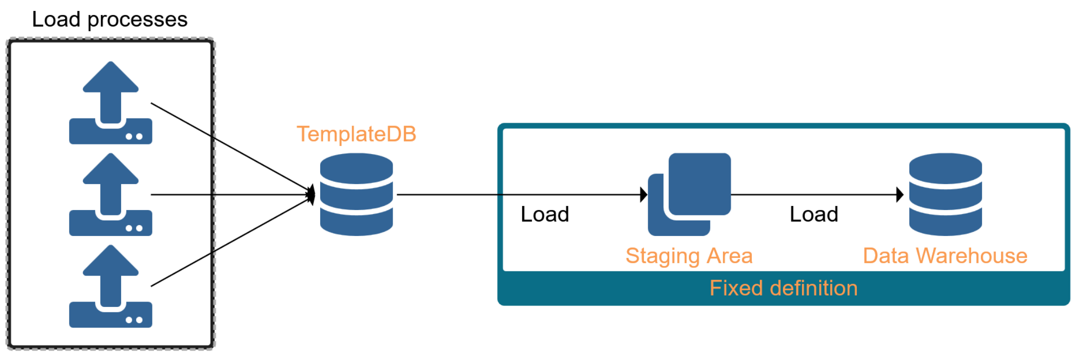


Figure 2: Detailed view of the integration of TemplateDB, in interaction with the ETL process and the Data Warehouse.

side. Each source now does not need to match against the data warehouse itself, but must only do so against TemplateDB, i.e. the storage of domain information. From this intermediate step, a second ETL processing routine, which does not need to be adaptive by design, can take place and load data into the warehouse.

When the warehouse itself is modified, for example to support a new key performance indication (KPI), the template will also be in need of an update. This is required as the domain knowledge needs to be reflected in the internal layout of TemplateDB. As Data Warehouses are most likely implemented in relational database systems, adding additional information is most likely carried out by implementing more columns. Furthermore, an adaptation of the pre-defined ETL script for data transfer between TemplateDB and DWH is needed. However, this holds no influence over the automated script generation, as it is not affected by adding new sources, but only when changing the final target.

As different sources will not need to be operated on at the same time, each source as an independent loading process from its system to the template database. This means, further on, that the generator can produce code tailored towards a single source at a time. This is reflected in system design by the individual symbols for ETL processes.

The decision to not use a traditional ontology language like OWL comes with various upsides. For ETL scrip generation, first and foremost, not leaving the realm of relational databases avoids switching languages and technologies. Second, tools and support for RDBMs are rather mature and well-documented. Third, the separation of domain knowledge and instances into two different locations would have been the same with OWL as well, to avoid unnecessary clutter and promote reusability.

4.4.4. KnowledgeBase

The core element of this system is called KnowledgeBase. It stores data concerned about the adaptive ETL processing. It will need to store information about various elements in the data sources and the data destination “TemplateDB”. More on why this intermediate step in front of the warehouse is necessary and the specifics, are available in *Section 4.4.3*.

The type of information stored in KnowledgeBase is trifold. First, it must persist the different elements definitions from both sources, these are corresponding to different universities in our scenario, and the ETL target, which in this case will be TemplateDB. Second, the mappings and transformations from one end to the other are established. Third and last, to store the relation between different data items, enabling data items to be fused when needed.

The first type of data stored is information about all the elements in source and targets. This would reflect tables and columns in a relational database. With this information, KnowledgeBase gets access to the domain knowledge encoded in TemplateDB.

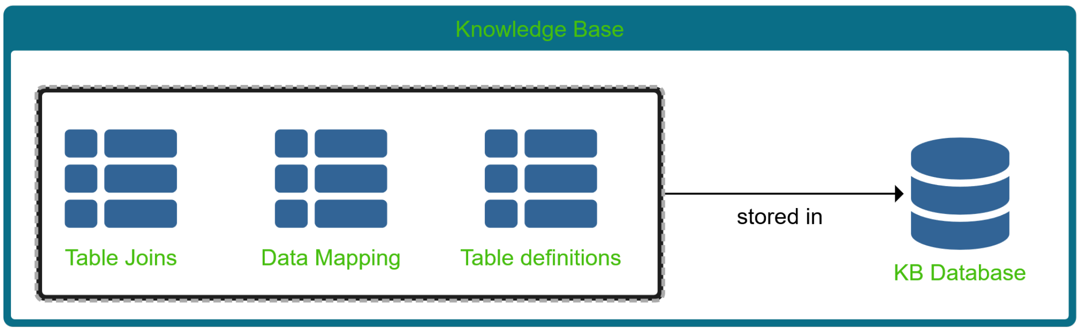


Figure 3: Detailed view of KnowledgeBase with the different types of stored data and its storage location.

Second, mappings, and transformations work by enriching the basic definitions of the first type with additional context. It is encoded how elements from the sources need to be combined, split, modified, transformed into other data types, and much more, to be compatible with their destination location. Another example from the reigns of RDBs could be a date field in the source, which is encoded as a date element, but the target uses a simple string, following some ISO formatting standard. This means that both a mapping from the source to target date is needed, and a transformation from the native date format to a formatted string.

Third and last, the relation between elements is mostly concerned with the sources. Typically, pieces of information have some semantic relation with each other. As an example, consider that a student is studying some subject, could be Computer Science, at some university, maybe Johannes Kepler Universität Linz. Then there is a relation between the student, a subject, and the university. In the realm of relational databases, these connections are often encoded as foreign key constraints, although some authors debate that RDBs in general lack true relationships [27]. Nevertheless, these connections are useful, and essential, when pieces of data, which belong to one logical unit in the destination, are stored separately at the source level. It is the way to let the generation software know how it can piece different elements together and transform them into script code.

4.4.5. Generator software

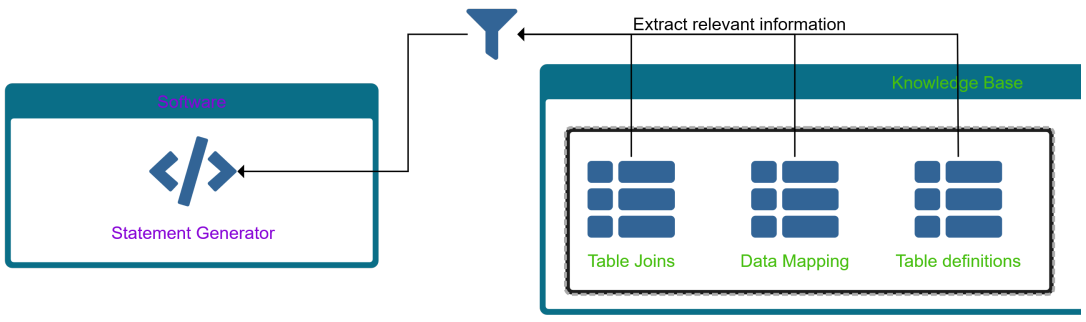


Figure 4: Detailed view of the integration of the statement generator software, in accordance with KnowledgeBase and its stored data.

With all the knowledge stored in KnowledgeBase, including the domain knowledge from TemplateDB, adapted ETL scripts for each database may be generated. The software will need access to the stored information and some additional input on which source is to be scripted. The latter can be provided either automated by a larger software system using the generator or stem from a user, selecting one of the available configurations. Upon selection, only the relevant bits of knowledge are then used to generate suitable ETL code. How the generator may access KnowledgeBase is entirely up to the implementation of this component. For most types of databases, a connector exists for the major programming languages, sometimes even database specific, or some sort of API.

4.4.6. Human interaction

While the system's aim is to reduce the required human interaction to a minimum, and avoid it during code generation in general, some human assessment will be needed to assure plausibility and data quality. Manual or guided meta information extraction from a freshly attached sources requires the engineer's interaction to ensure proper results. Furthermore, as *Figure 5* clearly shows, performing create, update and delete statements on KnowledgeBase itself must be possible as well. Be it to let the human fix machine-made errors or reflect source updates, among many other possible use cases. These interactions are analogues to CRUD capabilities in a database system [28].

The interface towards the human interactor can be of any kind deemed fitting. These can be a CLI, web-frontend via a REST-API, embedded into other programs, commercial solutions via adapters, or whichever other possibility and combination of tools is up to the provided task.

4.5. Component interplay

As a first impression, *Figure 1* does a wonderful job. Yet, explaining how components interact with each other before explaining their individual doings in details, would have

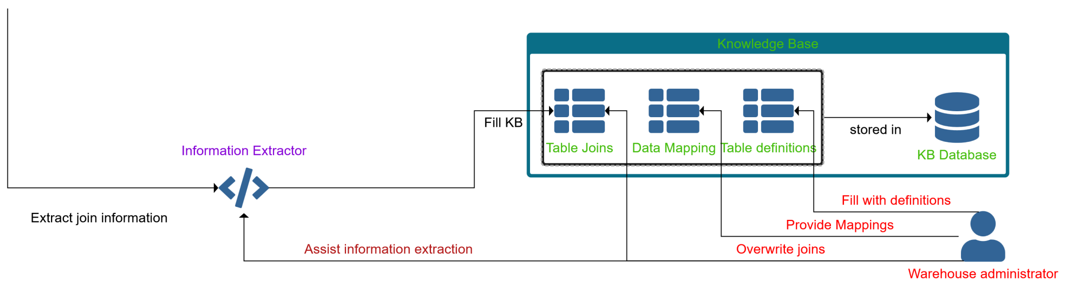


Figure 5: Possible interactions of the human engineer with the interaction, especially focused on the possibilities to interact with the loading process and stored data in KnowledgeBase.

been of little help to the understanding. Therefore, after careful commentary on each part, it is time to paint the whole picture. This section is explained from the end-user's perspective, therefore, from the first point of interaction when adding a new source via script generation until the point of executing the ETL script.

The flow of data starts when a new source is attached. Next, the metadata extractions starts and details about the layout and the relation between elements are extracted. This is reflected by the arrow on the left side of the image, running from the sources via a code block to KnowledgeBase. Mappings and transformations are generally more complicated, where techniques such as semantic mappings and data type analysis can help to find suitable matches. As this is not strictly part of this work, these ideas will not be implemented and the records are manually constructed by hand. More on manual data ingestion in *Chapter 5*.

Once KnowledgeBase is filled with information, script generation may start. After selecting the appropriate source, the generator will query for these items. The information is transformed into executable database code. This is reflected in the image by the line running from KnowledgeBase, via the software section, to the loading process area.

Once data is in TemplateDB, it has only to be passed on once more, into the data warehouse itself. This last step is not dynamically generated, as there is no influence through any of the data sources. The system overview reflects this, along with the written part dedicated to TemplateDB (*Section 4.4.3*), by another set of arrows, symbolising the flow direction of data through the system.

4.6. Design conclusion

This chapter handled the delicate topic of constructing a solution to support automated code generation for ETL processing. The approach introduces an information storage named KnowledgeBase, which holds insights about the sources and target as well as needed transformations, mappings, among other items. A code generator uses these further on to construct the ETL code. In the following chapters, we will discuss how this approach is both implemented and evaluated.

5. Implementation

This chapter will discuss the concrete implementation of the before explained solution. It starts with a general overview of the different implemented system components, before introducing the core aspects of each part and how they are interconnected with one another. This does follow the proposed design of *Chapter 4* and adopts it with tool-specific elements where appropriate.

As this thesis originates in a cooperation with a company, the utilized languages and tools were set by said corporation. This includes Java [29] as the main programming language and Microsoft SQL Server [30] or Oracle Database [31] as the favoured relational database management systems. After the cooperation ended, MSSQL Server and Java remained the RDBMS and, respectively, programming language of choice, as both were familiar already and also readily available for academic use.

This chapter will start off by covering the database aspects. This includes an explanation of the different database implementations, including KnowledgeBase, TemplateDB and designed sources as well as the target data warehouse. Afterwards, it will turn towards the code generator and discuss its elements in detail. For each section, the commentary will also look into the used version of software and, when noteworthy, their specific setup steps.

5.1. Databases

For the database, we picked Microsoft SQL Server 2022 as a pre-packaged Docker[*] container, to run it locally on the development machines without having to install the whole server package itself. Microsoft themselves already provide a docker image for x86-based machines, ready to be pulled and spun up [32].

Configuration-wise, no specialized options had to be chosen that were not already provided by the container image authors themselves, apart from a unique password and an added container name for easier recognition among all running services. This configuration assumes that standard locations for data transport, like port 1433, are available on the local machine. If this is not the case, the corresponding elements must be changed. The used run-command looks like the following excerpt:

```
docker run --name "SQL-Server" -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=aStrong(!)Password"
-p 1433:1433 -d mcr.microsoft.com/mssql/server:2022-latest
```

Listing 1: Docker command to set up and run the SQL Server

As by personal preference, the command can also be run in detached mode by adding `-d` and provided in the form of a docker compose file. This is, however, out of the scope

[*]www.docker.com

of the thesis. Docker will then, as expected, pull the image, supply the parameters and spin up the container.

When the following sections mention data types, they are supposed to be their equivalent on T-SQL. Typically, this is already done. A common example for a type that is typically known under a different name than it is implemented on a Microsoft SQL Server is a boolean element. While many programming languages, and also other SQL dialects, offer some bool/boolean value using the very name, SQL Server intends the engineer to use BIT, which is either 0 or 1 (or null).

The reference implementation uses Microsoft's 'Azure Data Studio'[*], built upon Visual Studio Code, and 'Database Projects' to set up the storage areas. Each project contains all SQL files, including potential dummy data and different schemas, organized into various directories. A very neat feature of the project format is its deployability, where each database item can be deployed to a server from Azure Data Studio and filled with data with only a very limited number of clicks and overhead. For some forms of change, it does support incremental updates by default, although only to a limited extent. For example, changing the column's data type, where the table already contains data, will result in an error, whereas an empty table allows the update process to pass. This can be mitigated with pre-deployment scripts, which reset the tables, but adds additional complexity to the deployment process.

Data Studio supports Jupyter Notebooks with additional kernels apart from Python, which comes in very handy for SQL scripts. The manual implementation of the ETL processed for testing is done in a notebook, as well as the data ingestion into KnowledgeBase for sources in testing and validation. The execution can be attached to a SQL server connection and run all its content at once or cell by cell, where the last option helps with debugging and table updates.

5.1.1. Goals

The specified target is loading data from any source to the data warehouse. With the proposed system design at hand, it is sufficient to only automate the section between a source and TemplateDB, whereas the way between TemplateDB and the data warehouse is not influenced by the selected source. From this follows that the latter task has no requirement to change, except when the data destination changes as well, and is coded statically as a Jupyter notebook, and can be run independently of the generated ETL code. The generated part comes in the form of a SQL script file and can be run on any SQL server with network access to the source and target database. This separation of concerns is further illustrated in *Figure 6*. Whenever this chapter speaks of code, it means the dynamic part and therefore the way to TemplateDB and not further onwards.

[*]https://azure.microsoft.com/de-de/products/data-studio

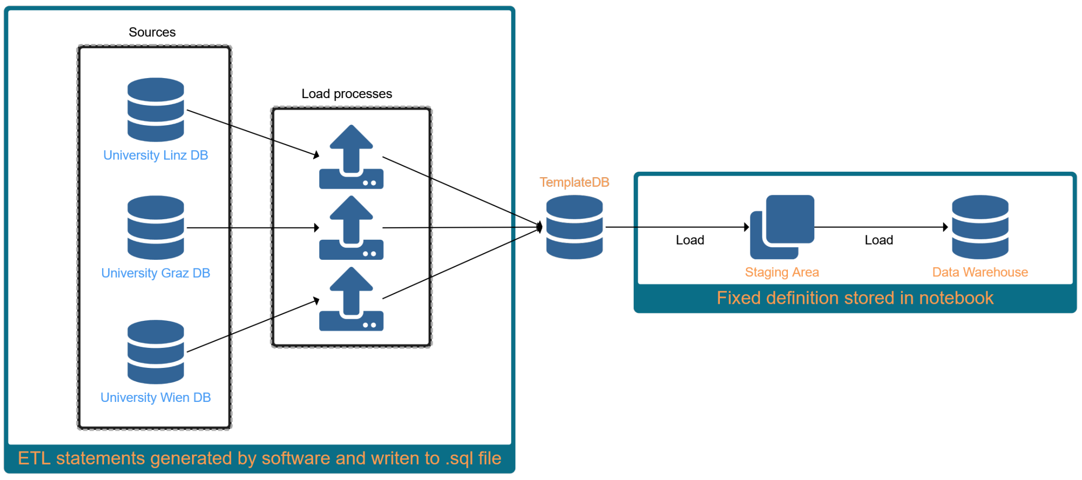


Figure 6: Data warehouse layout, including the fact and dimensional tables

5.1.2. Data Warehouse

Dimensions and Fact tables

To demonstrate the load process, a basic data warehouse is sufficient. In this case, it consists of a singular database with a dimensional fact model, as it is shown in *Figure 7*. The fact encodes some measure-able information, in this case the grade a student got, if the grade was sufficient to pass and the ECTS accredited for the exam. The dimensions a four-fold. First, information about different courses is gathered. These include the university, institute at the institution and the course's name itself. Second, the study program is recorded, this covers the student's faculty and the name of their enrolled program. Third of, some time information when the exam for a course was taken is stored, in this case in the format of year, semester, and month. Fourth and last, basic knowledge about the student themselves, including their gender and country and city of residence. The connections between the tables are executed as foreign keys, where the fact links to the primary key of any given dimensional table.

Most columns are designed as string data types and use `varchar`, except for the primary keys and the elements in the fact table. ECTS are given as floating-point numbers (`FLOAT`), the grade as an integer (`INT`) and the success value is implemented as a binary value (`BIT`). Other values, such as a year, could be implemented by either a date type or a number. However, the design does not aim to operate on the values directly, for example by calculating differences, but just store and group by them.

Staging area

Generally, data is not directly loaded into the tables straight away. It is often the case that a staging area has the job to first get a hold of the data, typically already with the correct data types, and only then transfer the information to the dimensions and facts. The reasons to do as such are manifold, but broken down to a common denominator

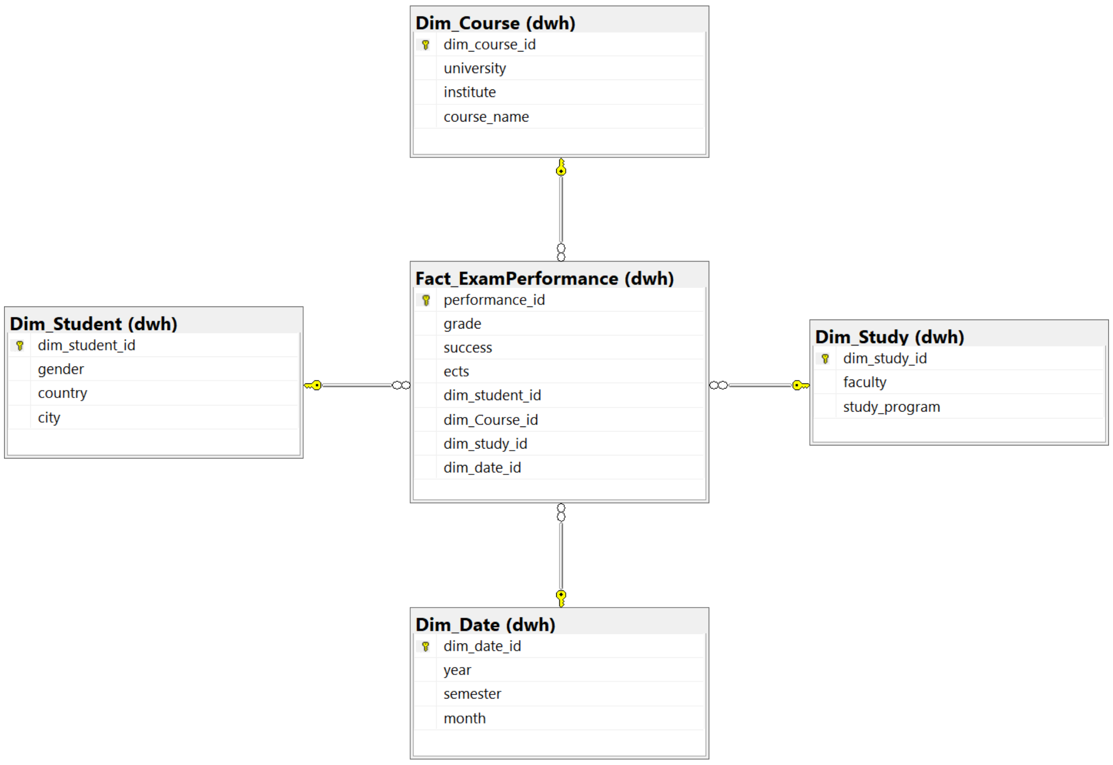


Figure 7: Data warehouse layout, including the fact and dimensional tables

it typically comes to essential qualities like delta loading, decoupling from the source system and acting as a buffer, or failure recovery [33].

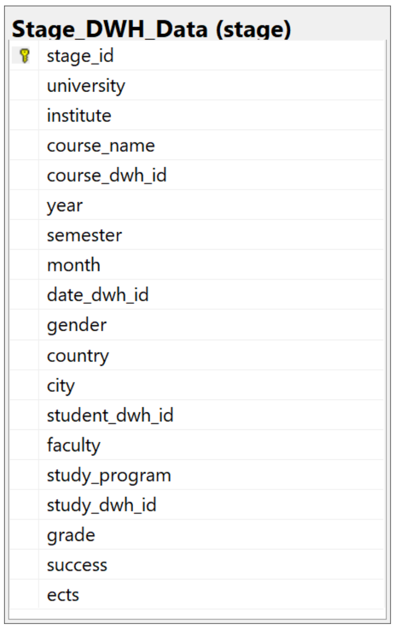


Figure 8: Staging area table of the data warehouse, incorporating all fields and primary key values from the dimension tables.

This stage, as shown in image ***Figure 8***, does have all the columns present in each dimension, including their primary keys, and all non-foreign-key columns from the fact table. Once all data is inserted from the source, a staged loading process to the data warehouse itself can start.

First off, the stage made sure to be empty. Next, each dimension is filled with its respective items, which are the unique combination of all its elements. For example, the date dimension is ingested with the elements from year, semester, and month. The others work along the same line. Further on, the probably most important step, is to get all generated primary keys and write them into the respective `_dwh_id` columns. This enables execution of the last step, which is to load the fact table, and properly link all dimensional tables with the fact record. As only unique combinations of data are loaded into the warehouse tables themselves, no duplicate items are loaded.

Considerations

The way ETL is handled is limited due to the demonstrative nature of this implementation. It is not the aim to support advanced, even though significant, features such as delta loading, update of values via 'slowly changing dimensions' [9], multiple data marts and many more. It is not without regret to limit oneself to the basic elements of ETL, and SQL, yet at the same time, adding such advanced features would not benefit the work itself. Not only that, but it will add extra layers of complexity on top of an already hard-to-grasp topic and not be directly relevant for the script generation, which is the core idea of this thesis. However, this again is heavily dependent on the individual implementation and execution of time-variant data. These must be considered in basically any production-ready grade data warehouse system, obviously, but we considered it not to be that foundational for this thesis.

While this layout looks rather simple, it includes a lot of the basic ideas of a data warehouse. The data structures follow the popular dimensional-fact model, and it contains a staging area. It is intentionally kept simple to not distract from the task at hand, but at the same time is not obfuscating its intended use-case by a DWH-unrelated database layout.

The main data warehouse has its schema inside the database, fittingly labelled `dwh`, whereas the stage elements are located in `stage`. It is considered a good practice, and is intended here for a logical separation of elements.

5.1.3. TemplateDB

As discussed in *Section 4.4.3*, TemplateDB does not only help the ETL processing itself, but also encodes the domain knowledge. We have already seen in the previous section, about the data warehouse, that information about a university and the performance of students is manifold. Therefore, TemplateDB encodes this information in a structure rather similar to the warehouse and its dimensional fact model, which further on supports more streamlined processing.

Layout

The layout, as shown in *Figure 9*, already looks more or less familiar with the data warehouse in mind. The dimensions of student, course, and study are also present, but the values from the time dimension are stored in a table called `Exam`. It does also incorporate the value of `grade`. The number of ECTS for a course, which is also stored alongside the grade in the DWH's fact table, is placed inside the `Course` table. In terms of domain knowledge, this means that an exam will earn a grade, and a course has a set number of ECTS` associated with it. As many students will confirm, this is a rather natural placement of these two elements.

The more eagle-eyed have for sure noticed that the tables `Exam`, `Student`, `Course` and `Study` all have a column labelled `source_id` among them. These are there for technical reasons throughout the load process to the data warehouse, such that the elements can be linked to the correct university in the table of the same name. This is beneficial during the TemplateDB load process, when data is moved from staging to the actual tables and their primary keys are ingested in the stage. Later on, the source identifier becomes irrelevant and is therefore not carried on into the data warehouse. There, as shown in the previous section, the university name is a part of the course dimension.

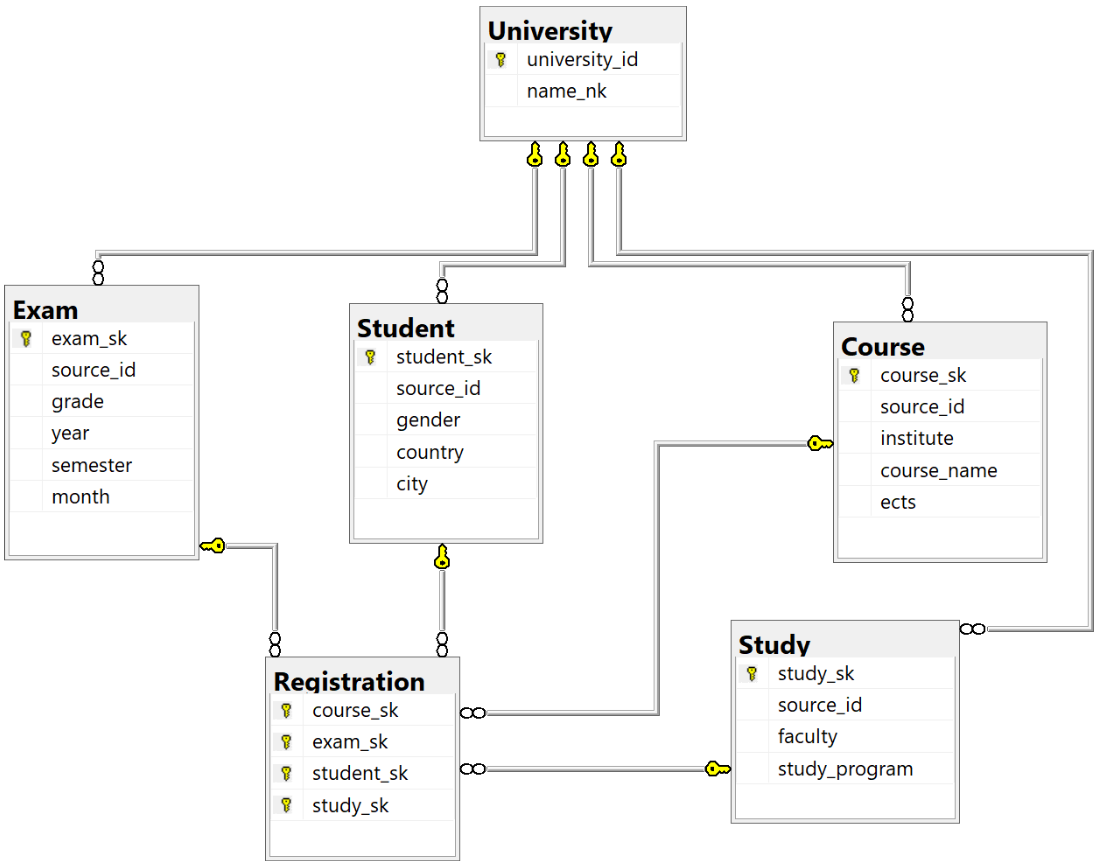


Figure 9: TemplateDB implementation layout

Domain knowledge

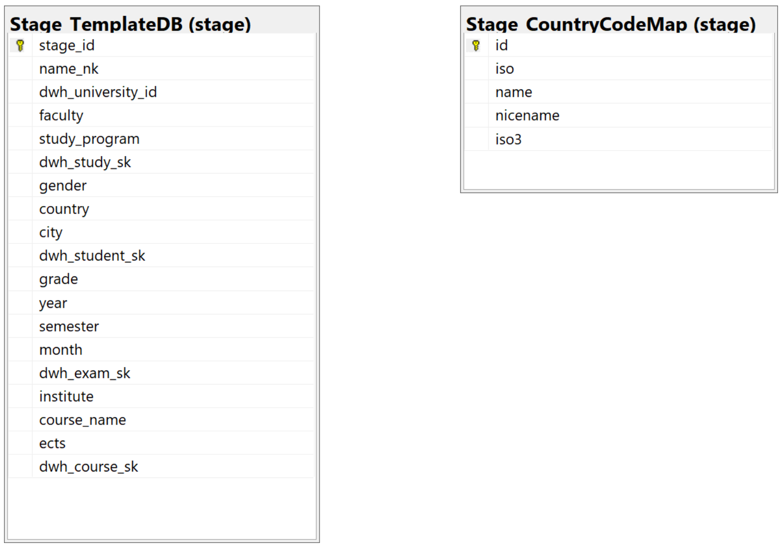
Finally, all items are linked via the `Registration` table, which holds foreign key references to `Exam`, `Student`, `Course` and `Study`. This associates a single student with an exam written for a given course for the student's field of study. This gives as all the elements to piece the domain logic together. It is rather easy to understand, especially with an academic background. A student identifies with a gender and has a location of residence, represented by the country and city they live in. To study, this person needs to be inscribed at the university for at least one study program, which is associated with a university faculty. The very same person writes an exam on a given date, which is in a semester of university, and gets a grade. The exam is taken for a specific course, held by one specific institute (what in the German-speaking regions is understood as an institute).

Staging area

Like the data warehouse, TemplateDB does feature a stage. It contains all fields from the different tables and adds nullable values for the generated primary keys, which are updated in stage after they are generated by data insertion in the main tables.

To load the stage area, the procedure just about aligns with what we do in the data warehouse. Therefore, the stage is cleaned and data is ingested. Next, the key update takes place and, last, all foreign keys are pieced together in `Registration`.

One convenient feature of the stage in TemplateDB is a little pre-filled table going by the name of `Stage_CountryCodeMap`. It contains five columns, including the autogenerated primary key, that allows the translation of the ISO two or three character country codes into the names of the world's countries. It features two name columns, where one is in all caps (`name`) and one with more pleasantly readable characters (`nicename`). ISO3,



Stage_TemplateDB (stage)
stage_id
name_nk
dwh_university_id
faculty
study_program
dwh_study_sk
gender
country
city
dwh_student_sk
grade
year
semester
month
dwh_exam_sk
institute
course_name
ects
dwh_course_sk

Stage_CountryCodeMap (stage)
id
iso
name
nicename
iso3

Figure 10: Staging area table of TemplateDB, both for all values and an additional country code mapping.

therefore the shorted code with three letters, is not guaranteed to be populated. In *Table 1* are some examples of the content.

iso	name	nicename	iso3
AT	AUSTRIA	Austria	AUT
TF	ANTARCTICA	Antarctica	NULL
AQ	GERMANY	Germany	DEU

Table 1: Example data from the mapping table for ISO country codes to names.

It comes handy when a source works with abbreviated, ISO-conforming short codes for countries. This is rather typical for postal addresses, where the country is not written in full, but only in its shorter, often also more convenient form. Working only with these codes may be harder, especially for understanding in the data analysis part, where the data from the warehouse finds its use. Therefore, the expectation arises to only use the nicer formatted name of a country. This stage table supports the effort of name conversion.

```sql
INSERT INTO [Stage_TemplateDB]
(.., [country], ..)
  SELECT ..,
    (SELECT nicename FROM [TemplateDB].[stage].[Stage_CountryCodeMap] WHERE ...),
    ..
  FROM ..
```

Listing 2: Example usage of the country code mapping table

During insertion into the main stage table, a select statement may be formulated that queries the mappings for a given ISO / ISO3 code and uses only `nicename`. To use such a select statement, please look at the code snipped in *Listing 2*, where it assumes that the select from `Stage_CountryCodeMap` is done as a sub-select in an insert statement.

As with the data warehouse, TemplateDB is split into two schemas. The main parts reside in the default `dbo` schema, whereas the stage is within `stage`. This is intended once again for a logical separation of elements and remains good practice.

5.1.4. Data source 'JKU'

Without a source of data, ETL processing becomes rather meaningless. To avoid such a pickle of a situation, several data sources were discussed, designed and implemented into SQL. Every database bears the name of an Austrian university, or at least a name to which the respective university is commonly referred as. Please note that none of the demo systems actually look like, or are close, to the layout of the actual systems used in production. It was simply found way less of a hassle to handle known names when referencing the data sources instead of a more generic name, like "Source 1" or "University A". The first one designed claims, rather fittingly, the name of Johannes Kepler Universität, or JKU, for short.

We will not go into detail about everything there is, not only due to a limited amount of relevance, but also because not every single table, or columns, is of high importance. One of the main characteristics this test source has to fulfil is providing data items for the ETL process. As this is, in a real-world scenario, most likely not a straight-forward path to stroll along, neither should both the test and evaluation be. Ideally, data in this source is not stored in the same fashion as in both the data warehouse and TemplateDB, to show the necessity of transformations and mappings. Its implications are that items stored in a single dimensional table are spread over multiple in the source, or have a different data type, among other options as simple as a different name.

Layout and construction

Very central to this database, as displayed in *Figure 11*, is the Student table, where each student finds themselves as an entry. Things like their full name, gender, matriculation date are recorded. Their place of residence is, however, stored in a separate table, fittingly labelled `StudentAddress`, with a foreign key reference from `Student` to its primary key. A student inscribes for one or more fields of study, where one is marked as the

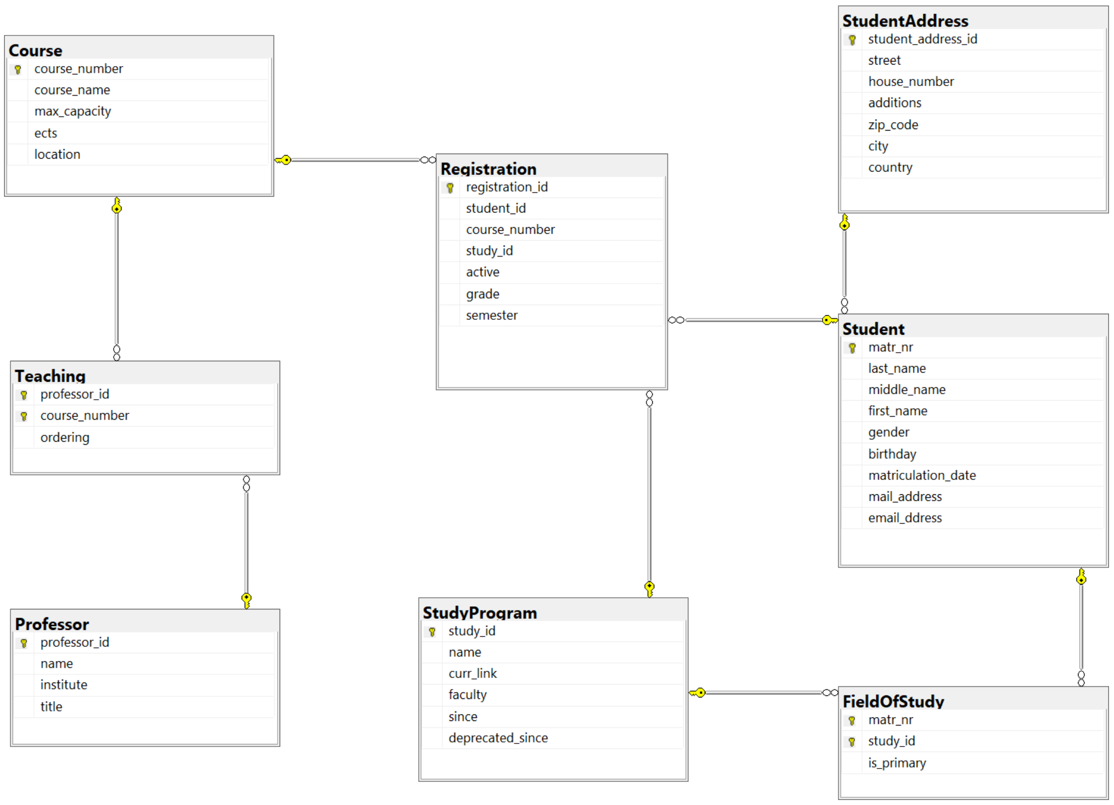


Figure 11: Source table layout, including columns, primary keys and foreign key relations, for the source labelled 'JKU'

main. The table called `StudyProgram` saves the different available studies. This contains some additional data, for example which faculty is associated with it or a link to its curriculum.

The various courses JKU offers have their data loaded into the `Course` table, where the important characteristics like ECTS and available spots reside. A student registers for one or more courses, each linked to one of the student's registered studies. Once a grade for a course is issued, it is stored alongside the other information in the `Registration` table. Additionally, it records in which semester the student takes this class. The relation is realised via foreign key relations to the course, study program and student.

Professors, or to not discredit every person giving a lecture, teaching personnel, have their items stored in `Professor` and linked, in an N:M relation, with the different courses. Therefore, one teacher can teach more than one class, and the other way around. The `ordering` item determines which professor elements is the primary, secondary, etc. teacher, if more than one person is giving the lecture.

Data types

As mentioned in the introduction to the JKU database, the sources will most likely have different data types from the warehouse and the upstream TemplateDB. In the latter two, basically everything is a string, except for values calculated with, which are ECTS and grade. Our JKU database handles some aspects different. For example, the grade is saved as a string as well, but the student attribute gender is encoded by an integer number. This means there is some conversion needed throughout the ETL processing.

5.1.5. KnowledgeBase

Perhaps the most important, and interesting, database design concerns KnowledgeBase. It must consider, as laid out in *Section 4.4.4*, the three main functionalities. First, to store the database layout of sources and destinations. Second, include mappings and transformations between the systems. Third and last, preserve the relationships between data items to properly extract data. Obviously, to properly achieve these goals, there are some additions to store for support of the finer details in ETL processing.

Kicking off this section is the overview in *Figure 12*, which, given, does look a bit frightening at first. There are many elements, even without every table displaying the associated columns. Not to worry, the upcoming paragraphs give a gentle introduction to the different aspects of the KnowledgeBase, starting with the core batch of tables, relevant for storing layout information up to the parts where sub-selects are clauses find their place.

On a technical side, all tables in KnowledgeBase lay within the `kb` schema, which currently has the same access rights as the default `dbo`. While this would not be strictly

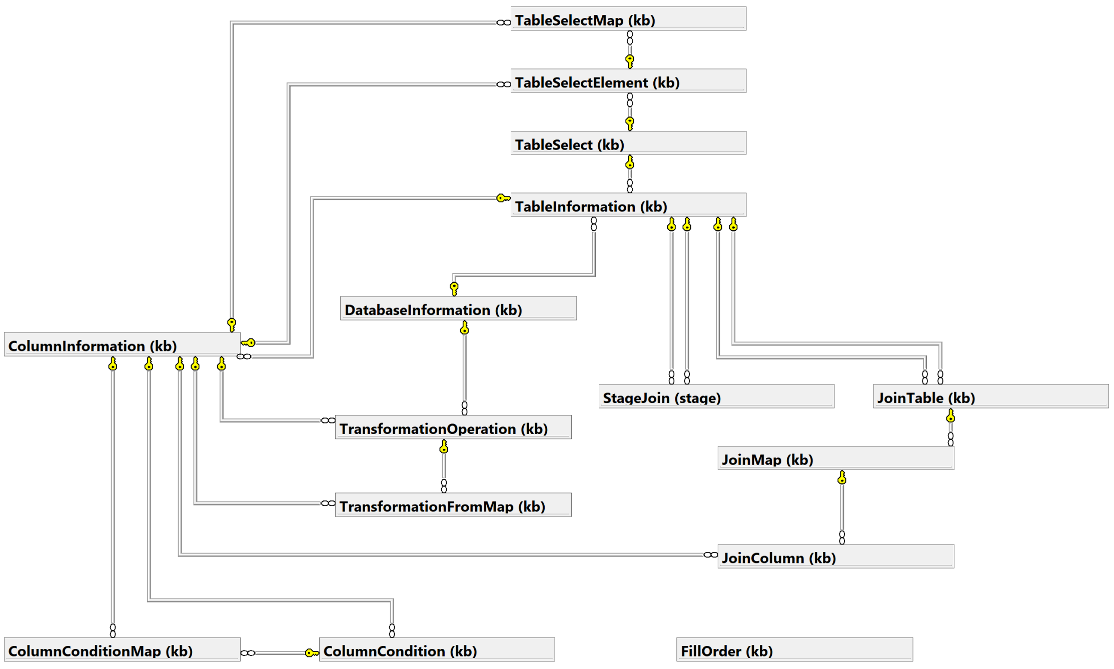


Figure 12: Overview of all tables inside KnowledgeBase implementation with foreign keys, not including the column names.

necessary, as everything resides within one schema only, it can still be considered good practice.

Layout information

KnowledgeBase needs a way to know in which way an engineer constructed the ETL sources and targets, as otherwise SQL statements would become rather vague on where an item is located exactly. The tables provided for this task are `DatabaseInformation`, `TableInformation` and `ColumnInformation`. The items are shown in *Figure 13*. While the names are rather self-explanatory, we find it still worthwhile to discuss design choices in more detail.

The support for multiple source databases is rather essential, therefore the database information stores the database name, and offers, by its generated ID, a way for other tables to refer to a specific database. This is, rather obviously, necessary to correctly link a column with its associated table. The table entry does not only store its name,

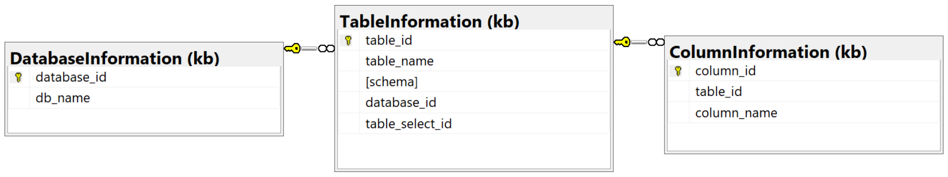


Figure 13: Table layout of information storage about the different database, tables and columns stored inside Knowledgebase.

but also the schema and, when applicable, a reference to a sub-select statement. The latter means a select inside a select (or join) statement. More on this further down the line. Last, each column belongs to a table and stores the name of the column. It does not record the column's specific data type. Matching these between the source and destination is so far the job of an engineer, designing mappings and transformations.

These three tables will be rather central to every other item, and that is why they will be displayed in the upcoming figures of other system parts as well. Most of these parts store some reference to a table or a column, sometimes also the database entry.

Mappings and transformations

When information is moved from A to B, the way it is stored can change. This may not only be limited to the device or the name, but also to the actual type. For example, an image-sending service could convert all photos into a JPG with awful compression, before it delivers it to the recipient. In the same manner, the example also holds true for ETL processing. However, in this case, the concern mostly revolves around available column data types of different SQL servers and how information from one format can be passed into the other. Second, to function properly, the generation system calls for a list of required transfers. These take the form of mappings from one or more columns to a destination column, including possible transformations. The proposed implementation is shown in *Figure 14*.

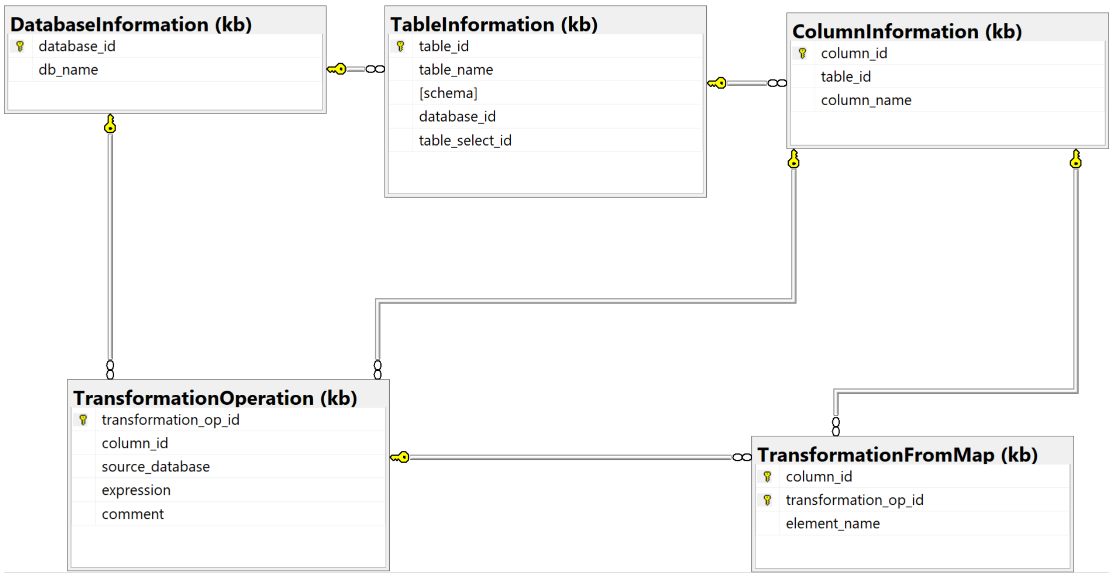


Figure 14: Transformations and mappings between source and target find their place in the transformation section of KnowledgeBase.

To support both of the requested features, two additional tables are added. The main table with the stored transformation expressions is labelled `TransformationOperation`. Within these, one column (`column_id`) references the data destination of a transformation and mapping operation. The table additionally saves a link to the column's source

database for a faster, or at least a less-complicated, lookup of said element. This comes handy when the entries are filtered to return only those of a single source. In some special cases, it poses as the only way to correctly associate a statement with the correct origin. One possible scenario might be, when the loaded value is a constant value and no source column is accessed.
The last important columns labelled `expression` holds the transformation operation. It takes the form of an SQL expression, hence the name, and placeholder names, which start with the dollar symbol (`$`), characters in-between and a whitespace at the end. Such a transformation could look like in *Listing 3*.

```
(Case When $col1 = 0 THEN ''Female'' when $col1 = 1 THEN ''Male'' Else ''Non-Binary'' END)
```

Listing 3: Casting a numerical value for gender to the string representation

This represents an If-ElseIf-Else construct, where the value of the to-be-mapped `$col1` is tested on being 0, 1 or something else, and the output `Female`, `Male` or `Non-Binary` is gathered. As it can be deducted, this transforms the source representation for gender into the data warehouse representation. The doubled single quotes stem from the character escaping required by the database syntax. However, how does one give the appropriate instructions to map the mysterious column to a real column from the respective source database?

The second added table takes the name of `TransformationFromMap` and basically boils down to a simple 1:1 translation between a given placeholder and an ID from the columns. To go with the example above, `element_name` contains the placeholder name `$col1`, `column_id` references, via a foreign key constraint, the ID from the gender column of said database and the `transformation_op_id` references the expression entry stated above. The placeholder can be freely selected, as long as it matches between the mapping and operation, and used multiple times in the same expression. If the operation needs access to more than one source item, simply create a second map with matching naming following the convention. An insert statement into the table could look like *Listing 4*:

```
INSERT INTO kb.TransformationFromMap
(transformation_op_id, column_id, element_name)
VALUES (1, 2, '$col1')
```

Listing 4: Example insert statement into the 'TransformationFromMap' table

Relations between items

Relational database systems have the name 'relation' already in their name. Such connections are often presented in the form of foreign keys, to associate items in one table with entries in another. Furthermore, these are essential to a knowledge-based system, and therefore it makes much sense to store the relation of elements with one another. To do as wished, the additional tables are threefold. *Figure 15* lets us view the proposed solution.

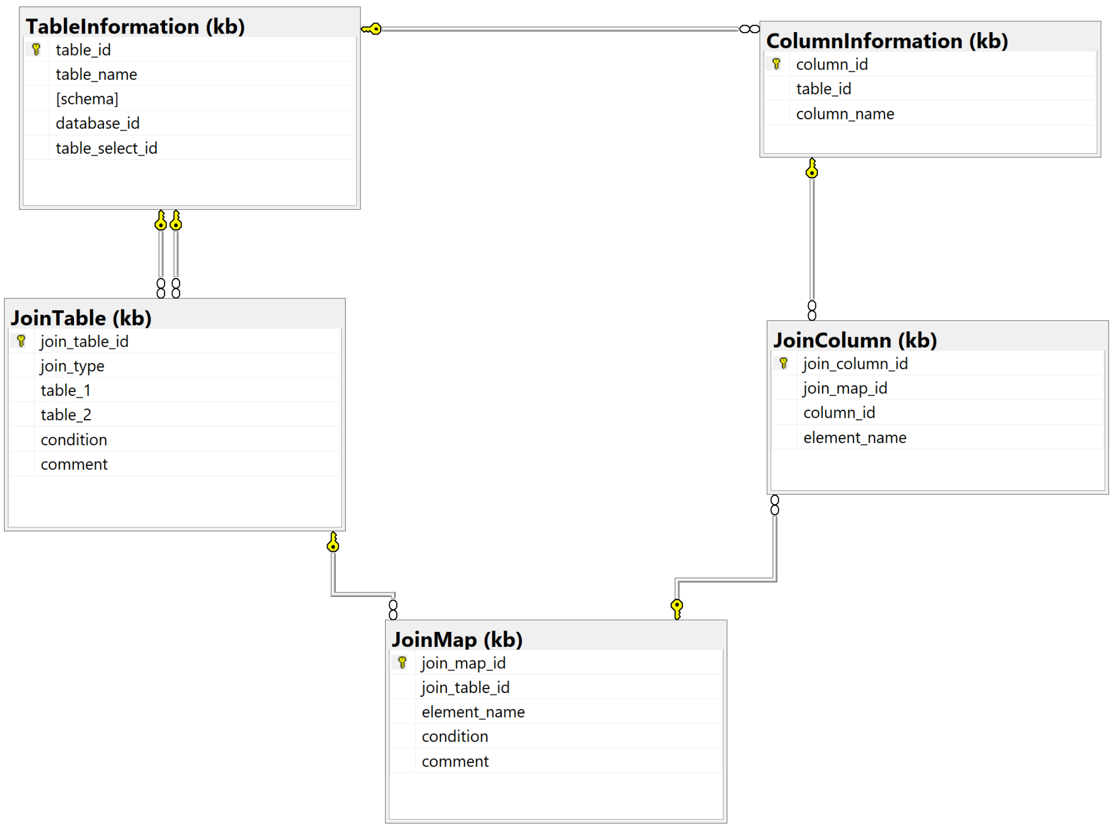


Figure 15: Information about joins, i.e. the relation between data, is stored as join information in KnowledgeBase.

First off, `JoinTable` stores the two tables, `table_1` and `table_2`, related by some `condition` and the exact type of join (`join_type`). Examples of the type are the well-known options such as inner, outer, and so on. The type is stored without the keyword 'join' itself, the generation software automatically adds it to the script. When `join_type` stores an empty string, RDBMs use their default join operator. Each of the two table columns is a reference to the primary key in `TableInformation`, and the developer has the ability to include a comment on why this particular table is needed inside the `comment` string column.

As seen in the image, there are three tables used to depict relations, not the two used for example to do transformations and conditions. While it can be done with only two tables, and there are reasons for it such as less database objects, we preferred to tackle the problem in three parts. It makes each row less cluttered and enables a stricter separation of concerns. Speaking of, the `condition` column of `JoinTable` just depicts the different mappings and how said items are connected logically. Typically, the on-clauses of joins conjunct the conditions with logical 'AND's, but there might be scenarios where an 'OR', or some other operator, is required. Having this in mind, one column entry may look like `$map1 AND $map2`.

Each `JoinMap` item takes care of one mapping. For the correct association, the string in `element_name` must match with its correspondent place in the often mentioned condition. It therefore also references the associated `JoinTable` row via their primary key as the foreign key reference. The engineer can leave a comment in this table as well for each row, having a place in the `comment` field. Each mapping has a condition itself, which is the actual measure used for joining elements. Therefore, this may look like `$col1 = $col2`, with the placeholders having their values assigned in the next table.

`JoinColumn` handles the mapping from named placeholders to the correct column in the source database. In doing so, the foreign key references the associated map's primary key. Another foreign key targets the concrete column by its ID. Lastly, `element_name` saves the placeholder name of the column, which could be `$col1`, in continuation with the example started above.

Each `JoinTable` stores all relevant information to properly join two tables together, therefore to relate them with each other. For each relation between two tables, an individual item in the table, and the associates for mappings and column linkage, is required. The generation software takes care of combining different relations at runtime. Only those relations strictly needed to perform ETL processing are allowed in KnowledgeBase. Otherwise, it is close to impossible for the generator software to check which connections are necessary and which can be excluded without any incorrect results. Let us imagine a scenario, where there are three tables `t1` to `t3` and each table has a reference to all the others. Now, ETL accesses data from `t1` and `t2`, and therefore these two are related to each other, but is `t3` also needed? This might be the case, as only the relation `t1<->t3<->t2` may produce the correct result, but it could also be the other way around or just not be necessary. Therefore, only storing the relations necessary is a strict requirement.

To sum up, the three-tiered approach breaks each join into equally many parts. The first level stores which two tables have an RDB relation and how many interconnected conditions they bond over. The second layer defines the properties of each element, i.e. the columns involved and how they are conjuncted. The third and final layer references the actual columns and maps the elements to them.

Conditions

Not all columns are perfectly fit for data extraction just as they are, as there can be additional conditions which have to be met. For example, when a course has no grade assigned, how much sense does it make to load this incomplete dataset, when the graded performance is so essential to the fact table? It can be argued that this might call for a special flag, indicating a missing entry. For this to work, however, we argue along the line that it is better to keep such items out of the data warehouse. This arises the need to allow for filters. This is met by allowing where clauses based on the value of one or multiple column(s).

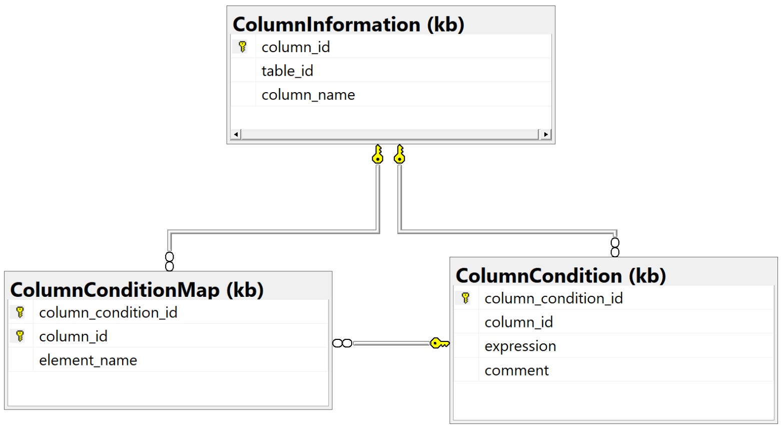


Figure 16: Several elements have additional conditions attached to them to avoid wrongful selection, and therefore those are stored as 'column condition' data.

The layout idea closely follows others which have the same expression and column mapping problem to solve. The "main" table, `ColumnCondition`, holds the reference to the conditional column, i.e. which column has to be included into a select statement such that the where clause must contain the defined expression. Said `expression` may have zero to many columns tested to match certain conditions. Such a statement could look like this:

```
$col IS NOT NULL
```

This simply means that the column tested must not be null to be allowed in the selection. The placeholder `$col` is strictly reserved for the column referenced with `column_id`, which in this case saves us from the additional insertion into `ColumnConditionMap`. Whenever the statement requires more columns, the placeholder naming can be chosen freely, as long as it isn't the reserved `$col`, and the according map entry must be done. Let's give an example with two insert statements, where the first one defines a condition and the second one the mappings.

```
INSERT INTO kb.ColumnCondition (column_id, expression)
VALUES (1, '$col <> $col_other') --Assume this gets column_condition_id 10

INSERT INTO kb.ColumnConditionMap(column_condition_id, column_id, element_name)
VALUES (10, 3, '$col_other')
```

Listing 5: Filling information into column condition tables for where clauses

Subselects

The last feature of KnowledgeBase, labelled subselects and stored in the database tables with the prefix `TableSelect`, has on task in mind, even though such a construct in theory supports more powerful operations. Sometimes, data is both stored in a specific way and used in another format in multiple places, each requiring the same transformation before being used. Implementing this transformation in multiple places creates its own

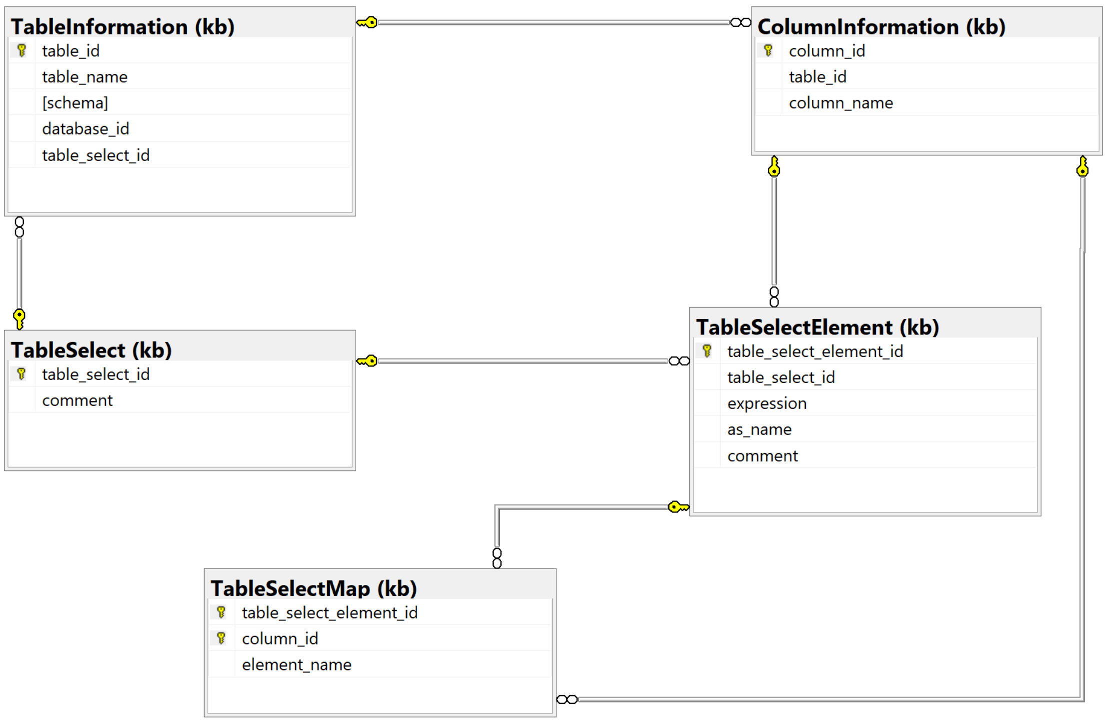


Figure 17: Information about customized sub-select statements in FROM / JOIN clauses is saved as ‘table select’ elements.

problems, especially introducing unnecessary bloat and complicated updates for the transformation. Therefore, subselects are introduced.

As seen in *Figure 17*, three tables take on the job. The first, simply labelled `TabbleSelect`, just provides an ID to indicate that such a subselect exists. The ID is referenced by `TableInformation`, where a null value indicates the absence of the customized select. `TableSelectElement` holds the actual `expression` which will perform the data transformation inside the select. As the output of such a process has no fixed name, the `as_name` column maps it to a column already existing in the source table. Typically, this conveniently maps to the column, or of the columns, being transformed. The placeholder `$as_col$` is also reserved for this purpose, where the generation software recognizes the column behind `as_name` as the replacement for this placeholder.

The `expression` must also include all other columns ever used by any other transformation referencing the table, as otherwise they are not available for selection during the ‘Select’ operation. The engineer may leave a note on why things are done this way in `comment`, the same is possible in the entry in `TableSelect`, referenced by `table_select_id`. Mapping takes place in `TableSelectMap`, where the placeholders in the expression are filled with actual columns, the latter referenced by `column_id` and their names stored in `element_name`. The following code example gives an idea of how elements integrate with one another.

```sql
-- Assume the @as_col has ID 3
DECLARE @as_col INT = 3;

-- Assume this gets the ID 11
INSERT INTO kb.TableSelect (comment) VALUES ('Convert a unix timestamp to datetime');

-- Assume this gets ID 21
INSERT INTO kb.TableSelectElement (table_select_id, expression, as_name)
VALUES (11, 'dateadd(S, [$col1], ''1970-01-01'') AS ''$as_col'', [$col2],', @as_col)

INSERT INTO kb.TableSelectMap (column_id, element_name, table_select_element_id)
VALUES (@as_col, '$col1', 21), (@col2, '$col2', 21),
```

Listing 6: Defining a sub-select for a table

This functionality is intentionally limited, to not open up a very delicate and complicated way of supporting arbitrarily complex subselects and subqueries. That is why the range of options has been deliberately reduced and tailored towards its purpose only. We are aware that `as_col` name cannot be selected for its actual value when it is also transformed at the same time. However, in such cases, we strongly suspect that the transformation is used only seldomly. Therefore, just using the transformation where it is needed and otherwise simply referring to the value directly, without writing the subselect and the need to always include all used columns, should be the more user-friendly way.

Fill order

The last to be explained table in KnowledgeBase is also the exception to the rule, as it does not hold any foreign key references to table or column information. `FillOrder` specifies the order in which tables in the ETL target are filled with data. For example, TemplateDB requires its `University` table to be loaded first and `Registration` only at the end. `FillOrder` stores the table name and schema and the order priority, as seen in *Figure 18*. The last column `order_priority` features a numerical value, where a lower number means a higher priority.

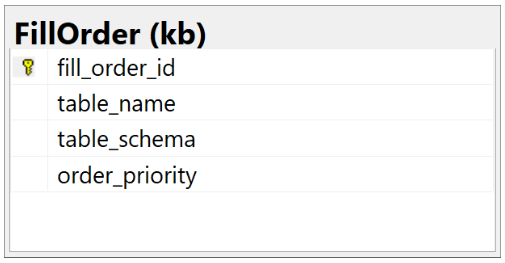


Figure 18: The order in which tables from TemplateDB are filled.

Another option to store the tables would be a reference to `TableInformation`. As TemplateDB is relatively static in its design, creating a copy of the data in `FillOrder` and at the same time avoiding another join was deemed to be an acceptable trade-off.

5.2. Statement generator

The second large software part consists of the statement generator, responsible for transforming the information stored in KnowledgeBase into executable SQL code. The introduction to this chapter already mentioned Java as the programming language of choice, but it obviously did not go into detail on the specifics, such as the IDE, frameworks, build tools, architectural design choices and similar topics.

5.2.1. Java

The software generator was set up with Java [29] in version 22 and a feature-level compatible runtime. For development, OpenJDK libraries were the tool of choice. In theory, there were no used features newer than JDK 17, however, not all components are set up to run with anything older than the preferred JDK out of the box.

5.2.2. Spring

This project uses Spring [34], one of if not the most popular Java frameworks. It describes itself as making "programming Java quicker, easier, and safer for everybody". While it does not impose a specific programming paradigm upon the developers, it supports various architectures and designs, such as batch processing, microservices, cloud, or web. While Spring is well-known for its capabilities with web applications, full-stack development, and therefore also cloud services, these are not needed here, and left aside.

Most useful to this thesis and the implementation is the possibility to seamlessly integrate a variety of powerful and supporting libraries, such as Spring Boot, Spring Data JDBC, Spring Shell, JUnit testing, and bundling with the build tool Gradle*. The latter in this use case takes care of building and running the program and the tests, as well as resolving and linking dependencies. The documentation also meets expectations for nearly all of these elements, as well as tutorials and available online resources, making it a solid choice for a project in progress.

One really useful feature heavily used is the automated dependency injection offered by the framework [35]. What this means is that fields have not to be filled 'manually' by writing the constructor in some class and wiring all the instances together for class instantiation in the main method. This means that, for example, a class 'COne' requires an instance of 'CTwo'. Therefore, one would need to first create an instance of 'CTwo' (`CTwo c2 = new CTwo()`) and then call the constructor of 'COne' and pass the object reference (`COne c1 = new COne(c2)`). Spring instead allows its so-called 'beans' to auto-inject the classes as needed and takes care of all the wiring. This is all done in the class itself and does away with the need to manually call constructors and pass objects, the framework takes care of these steps. The class just requires a `@Autowired` annotation on its primary constructor to let Spring know about the requirement of dependency

*https://gradle.org

injection. Within the framework, the IoC (Inversion-of-Control) container takes care of all object management and injections [36].

```
@Service
public class cOne {
  private final CTwo c2;

  @Autowired
  public cOne(CTwo c2) {
    this.c2 = c2;
  }
}
```

Listing 7: Example for dependency injection and Inversion of Control

Starting a new Spring application is relatively easy with Spring Boot, which facilitates working with Spring by using a "convention over configuration" approach. It is not essential to know all the details about it, the takeaway message of simpler and faster to use should be sufficient for this project. There exists a rather useful, official tool named `Spring Initializr`[*] that creates a project with the chosen Java, Spring version and project tool as well as selected dependencies, alongside other core elements such as the project name and core package layout. The downloadable archive can be imported into the IDE of choice, which for this project is JetBrains' IntelliJ IDEA[†].

Spring Data JDBC

Spring Data, as a framework in the world of Spring, offers a unified way to access data sources of all kinds, including relation and non-relational databases. It retains many of the Spring paradigms, and incorporates the individual aspects of the underlying storage type [37]. With 'Spring Data JDBC', this refers to relational databases and the 'Java Database Connectivity' component used.

A rather useful feature of Spring Data JDBC is the built-in mapping to classes, with support for customized SQL queries. It was deemed to be both sufficient and easy to implement for the task at hand. We found it much easier to use in comparison to plain JDBC, thanks to the automated mappings and default CRUD operations, provided by Spring Data. At the same time, it was more straightforward to use than Hibernate or Spring Data JPA. However, we understand that this view differs between developers, especially with ongoing discussions on Spring Data JPA versus Spring Data JDBC. We find the more lightweight approach and potentially better performance of the chosen system to be more suitable.

Below in *Listing 8* is an example of how an annotated data class may look like, to allow Spring Data JDBC automated loading mechanism to work. The specific example uses

[*]https://start.spring.io
[†]https://www.jetbrains.com/idea

a `record` class type, introduced with Java 17, which avoids boilerplate code like ‘Getter’ and ‘Setter’ methods, although it comes with some limitations on its own.

```
@Table(value = "ColumnInformation", schema = "kb")
public record KBColumn(@Id @Column("column_id") int columnId,
                       @Column("column_name") String columnName) { }
```

Listing 8: Java class annotations for automated data mapping via generated SQL

One nifty feature of Spring Data JDBC is `MappedCollections`, which automatically resolves One-To-Many relationships into lists of elements, where the ‘One’ part automatically holds a list of the ‘Many’. A suitable example is `TableInformation`, i.e. the table of a database, which obviously contains one or more columns associated within.

```
@Table(value = "TableInformation", schema = "kb")
public record KBTable(
        ...
        @MappedCollection(idColumn = "table_id", keyColumn = "table_id")
          List<KBColumn> columns
) { }
```

Listing 9: Example for ‘MappedCollections’, resolving a one-to-many relationship as a simple list of items.

The basic CRUD operations implement effortless, where a simple interface extension of `CrudRepository` provides operations on all items and by the primary ID. The extended `ListCrudRepository` maps iterable objects returned from ‘find’ and ‘save’ methods into lists of elements.

```
public interface KBColumnRepo extends ListCrudRepository<KBColumn, Integer> { }
```

Listing 10: Using a CRUD-Repository for easy and fast access to data

Spring Data JDBC provides a syntax for implementing additional queries without writing the actual SQL code for it. As an example, one may implement a search for a `KBColumn` by its name, as done in *Listing 11*. Spring will automatically translate this into SQL and execute when called.

```
public interface KBColumnRepo extends ListCrudRepository<KBColumn, Integer> {
    KBColumn findByColumnName(String columnName);
}
```

Listing 11: Extension of the plain CRUD repository with an additional method, which can automatically infer the necessary SQL by adhering to a naming convention

When the SQL statement in question is not obvious to Spring, the query itself can be passed with the `@Query` annotation. It ingests the values of passed parameters syntax-wise by an added colon in front of the parameter name.

```
public interface KBJoinMapRepo extends ListCrudRepository<KBJoinMap, Long> {
    @Modifying
    @Query("UPDATE [kb].[JoinMap] SET join_table_id = :table_id WHERE join_map_id IN (:pks)")
        void updateTableIdByPrimaryKey(@Param("table_id") int table_id, @Param("pks")
List<Integer> primaryKeys);
}
```

Listing 12: Extension of a CRUD repository with a custom SQL statement

Spring Shell

Most programs need a way to receive commands and input by the user, or at least another program. For humans, GUIs and CLIs are the most typical, as things such as (REST-)APIs may not be considered to be as user-friendly as the other options. For this implementation, we opted for a Command Line Interface and chose the Spring component 'Spring Shell' for an interactive shell experience. It already provides some functionality by default, such as a `help` option listing all available commands, `version`, `clear`, `history`, and `exit`, to politely kill the application.

Customized commands are best grouped into classes. Each class requires the annotation `@ShellComponent`, marking it as a class that contains commands. Methods in such a class are marked with `@ShellMethod(key = "cmd name", value="Description")`, where the key represents their callable name and the value a description of what it does. The parameter list of the method may contains a `@ShellOption(defaultValue = "")`, i.e. passed parameters after the command's key, which in turn can have a default value when the user input is absent. As an example, the code in *Listing 13* shows most of the relevant code pieces which form the CLI command to generate an SQL script. We omitted the business logic actually performing the generation, or at least calls to methods performing it.

```
@ShellComponent
public class GenerationCli {
    ...
    @ShellMethod(key = "gen etl", value="Generates ETL statements for a selectable source
to the default datatarget")
    public void generateETL(@ShellOption(defaultValue = "") String srcName) {
        ...
    }
}
```

Listing 13: Spring Shell method, declaring the key and text alongside a parameter

Naming of the commands is oriented in natural language, without requiring to write out longer words. As seen above, 'generate' is shorted to 'gen', database is abbreviated by 'db' and KnowledgeBase as 'kb'.

5.2.3. Junit

Testing the application is a good way to ensure correctness and reliability. Due to the complex nature of the script generation, and the potential necessity to test against a test or validation database, the automated testing is limited in scope. It consists of architectural test, covered later on, and ensuring that mappers work correctly. For the latter, a popular testing framework named JUnit[*] in the latest version 5 is up to the task.

The `@Test` annotation marks a method, fittingly, as an executable test. Variables and environments with a need to be set up or reset before all or before each test do not have to be done in each test individually. This can be outsourced to separate methods, marked with a `@BeforeAll` or `@BeforeEach`, respectively. The same holds true for clean-up tasks, labelled with `@AfterAll` and `@AfterEach`. There are many more options available, however, for these mapper tests, there were no others required than the 'Test' and 'Before' annotations.

Testing itself uses one or more 'Assertions', each of which tests some method result against some other value. When it holds true, the test pass, and otherwise an error message shows up in the output panel. Popular assertions include `assertEquals`, `assertNull`, `assertThrows` and their negations. Each method typically features variants with overloaded parameters, such that each default and primitive data type has coverage. The code in *Listing 14* features a simple test case, with a setup call before the test execution itself. It features two assertions calls and some code that maps objects from one type to another. The fields are then tested for correct mapping.

```java
public class LayoutInfoMapperTest {
    private static KBDatabase refDatabase;

    @BeforeAll
    public static void setUp() {
        refDatabase = new KBDatabase(1, "name2", List.of(refTable));
    }

    @Test
    public void testMapperDatabase() {
        final KBDatabaseDTO dto = LayoutInfoMapper.INSTANCE.databaseToDTO(refDatabase);
        Assertions.assertEquals(refDatabase.databaseId(), dto.databaseId());

        final KBDatabase fromDTO = LayoutInfoMapper.INSTANCE.databaseFromDTO(dto);
        Assertions.assertEquals(dto.databaseId(), fromDTO.databaseId());
    }
}
```

Listing 14: Test class for mappings between an element and its DTO.

[*]https://junit.org/junit5

5.2.4. Architecture

Every piece of software has some architecture style it follows. Most likely, the style chosen is specifically beneficial for a program, and followed as well as the people in charge know. Every so often, an architecture evolves from a small program, undergoing transformations as the code base grows. The quality and result may differ from deliberate attempts to properly enforce a specific architectural style. Popular approaches include the layered monolith, microkernel, microservices or even-driven systems [38].

In this project, as it is a rather small in scale, the architecture must not be too complicated and add bloat. Therefore, a more adequate and developer-easy style named the monolithic layered architecture is the inspiration here, while not strictly following and adhering to every single one of its teachings. A layered approach boils down to separating technical concerns into different layers. The 'lowest' takes care of data persistence, followed by a service area containing workflows (the 'business logic') with presentation 'on top'. Everything is packed into one deployment package, and therefore one can call it monolithic [26]. Spring, or better say Spring Boot, supports such a layout by default, as it already suggests a layered separation of concerns.

The program decomposes into four different main packages, where each may contain additional program components to logically encapsulate different concerns.

- `Cli`, containing classes for the user interaction via a shell.
- `Configuration`, handles local Spring configuration files and provides access to their values during runtime.
- `Generation`, takes care of generating SQL scripts and contains the business logic.
- `KnowledgeBase`, representation of the information stored in KnowledgeBase as Java classes for reading and writing via services.

We will go into more detail about the actual content of the separate parts in the upcoming *Section 5.2.5*. For now, it is only important to realize that these components aim towards the style of a layered architecture. `Cli` is the presentation layer, whatever basic it may be. `Generation` contains the workflow, therefore concerned with business logic. The persistence is split into two, but for good reason. Where `KnowledgeBase` handles the connection with the database and provides a service component for the script generator, `Configuration` manages the program-internal representation of spring application properties. The latter is not strictly persistence, at it only provides read access and does not behave like a typical information store. Moreover, for anything not related to script generation, there is no service layer as a separate module. The CLI does provide its own service layer, combining the different persistence packages and, where required, calling the generation service layer.

Testing the architecture

Deciding on an architectural style is one thing, but following it through or at least staying in line with some of its principles often turns out to be a tough challenge.

Putting trust in the developer and yourself is a good thing, definitely, however, some way of testing has its merits as well. This is where ArchUnit[*] comes into play, a dedicated testing framework for architectures in a Java environment. Adding it to a project is easy via Gradle as a test dependency, and it integrates nicely with JUnit.

ArchUnit facilitates a natural language-like syntax, based on the English vocabulary, to test architectural bounds. While some programming languages can limit the visibility of classes and methods rather finely grained, Java is not really one of them. There is the option to limit visibility to the class, a package, or everything. However, already the package visibility is too little when sharing a class in a sub-package (like `internal`) with its parent (like `Cli`), even though they are logically in the same layer architecture level ('Presentation'). Marking it as 'public' is therefore required, but makes it available to other layers (i.e. packages) as well, and that is not wanted. Another thing viable for checking is the separation of concerns between the layers, such that the presentation has no references to persistence (and the other way around as well).

The example in *Listing 15* shows how ArchUnit checks that an `internal` package has no reference outside its module named `Generation`. The base package for the program is `eu.sternbauer.EtlGenerator`. The listing excludes any import statements.

```java
public class GenInternalTest {
    private static JavaClasses importedClass;

    @BeforeAll
    static void setUp() {
        importedClass = new ClassFileImporter()
                .withImportOption(ImportOption.Predefined.DO_NOT_INCLUDE_TESTS)
                .importPackages("eu.sternbauer.EtlGenerator");
    }

    @Test
    void ensureGenerationInternalStaysInternal() {
        ArchRule rule = ArchRuleDefinition.classes()
                .that().resideInAnyPackage("..Generation.internal..")
                .should().onlyBeAccessed().byClassesThat().resideInAnyPackage("..Generation..");
        rule.check(importedClass);
    }
}
```

Listing 15: Architecture test for checking access to internal classes

5.2.5. In detail

This section goes into detail about the four main components of the program implementation, each separated into each own Java package. Each one roughly represents one layer of the architecture and will also be referred to as a system's module. This

[*] https://www.archunit.org

does, however, not strictly coincide with the Java module system, and should not be mistaken for it.

Package KnowledgeBase

KnowledgeBase consists of six base packages, where each represents one of the main aspects of the information storage, as seen in *Section 5.1.5*.

- Condition
- FillOrder
- Join
- LayoutInfo
- TableSelect
- Transformation

Within, each of these six features the same principal layout. The service package usually inhabits one class, providing the service interface for further usage in other classes or modules. The internal packages were made to stay internal for each element in KnowledgeBase, and therefore its insides must not be referenced from outside their base package, and best not even exposed when possible. However, as already mentioned at *Section 5.2.4*, Java visibility modifiers are rather limited in their ability to finely tune accessibility and checks with ArchUnit are utilized to check these constraints. Each `repository` class manages access to the database elements, and model classes inside the `internal` package reflect the database layout, as seen in *Section 5.2.2*.

```
|- <KBElement>
   |- dto
   |  |- <DtoElements.java>
   |- internal
   |  |- mapper
   |  |  |- <ModelMapper.java>
   |  |- models
   |  |  |- <Models.java>
   |  |- repository
   |     |- Repository.java
   |- service
      |- ServiceImplementation.java
```

A common design pattern when dealing with a layered architecture names itself “Data Transfer Object” (DTO), and helps with encapsulating data, and only data, from persistence to other parts of a program. There exist multiple reasons to use this pattern specifically, among them the idea to only pass on a specific subset of elements from a database access object without including any sensitive data [39]. In KnowledgeBase, this is true for any comments left by a developer for mappings, transformations, and other complex operations. These are not relevant outside the database and are therefore not mapped. It also means that they cannot be set directly via a DTO, which is not relevant in this scenario, though.

Spring Data JDBC maps the database tables into the classes provided in the `internal.models` package. The class placement should also indicate the requirement of avoiding any outside references by all costs. The DTOs, on the other hand, are stored in `dto` and are, by design, to be sent around as return values of the element's service implementation.

While mapping could be done by hand, an automated solution saves time during coding and can avoid unnecessary errors with manually written 'boilerplate' code. A fancy little library named 'MapStruct'[*] does the trick. It integrates nicely with Spring and Spring Boot's 'Convention over Configuration' approach.

Implementation-wise, it is also easy to execute for the developer. First of, an interface holds all methods. Each method accepts one argument and has as a return the converted data class. When the field names between the base class and its DTO do not match, MapStruct requires a rather simple `@Mapping(source='', target='')` annotation. This is not needed for any conversion in this project. The last step requires the setup of a static field inside the interface, which other classes can use to access the converter methods.

```java
@Mapper
public interface LayoutInfoMapper {
    LayoutInfoMapper INSTANCE = Mappers.getMapper(LayoutInfoMapper.class);

    KBDatabaseDTO databaseToDTO(KBDatabase database);
    KBDatabase databaseFromDTO(KBDatabaseDTO databaseDTO);
}
```

Listing 16: Mapping data between its class and the respective DTO

MapStruct automatically creates the required code during on compile time [40]. The defined methods can be used like any other, therefore called via `INSTANCE` or passed on as a function reference where applicable, for example to a mapping method in a stream.

```java
// Usage can be like this (as a method reference)
list.stream().map(LayoutInfoMapper.INSTANCE::databaseToDTO).toList()
//or like this
KBDatabaseDTO dto = LayoutInfoMapper.INSTANCE.databaseToDTO(KBDatabase);
```

Listing 17: Example usage of MapStruct generated mapping methods

The last subpackage, `service`, typically contains one class with multiple service methods. These are closely tied to the repository and bring together all components, including the mapping of data from the database access object to the data transfer objects. Most methods include read access in various forms to KnowledgeBase, whereas some allow writing to it. Below is a simplified version of the service implementation for layout elements. The services are designed to be injected into other classes, most likely inside the 'Generation' module.

[*]https://mapstruct.org

```java
@Service
public class LayoutInfoService {
    private final KBDatabaseRepo kbDatabaseRepo;

    @Autowired
    LayoutInfoService(KBDatabaseRepo kbDatabaseRepo) {
        this.kbDatabaseRepo = kbDatabaseRepo;
    }

    public List<KBDatabaseDTO> findAllDatabases() {
        return kbDatabaseRepo.findAll().stream()
        .map(LayoutInfoMapper.INSTANCE::databaseToDTO).toList();
    }

    public List<KBDatabaseDTO> findAllDatabasesByName(String name) {
        return kbDatabaseRepo.findKBDatabaseByDbName(name).stream()
          .map(LayoutInfoMapper.INSTANCE::databaseToDTO).toList();
    }
}
```

Listing 18: LayoutInfo service class with methods and database repository

Package Configuration

The configuration package is by far the slimmest one, yet fulfils an equally important job. Authentication credentials to SQL servers have to be stored somewhere, yet the database may not be the ideal place. The password would need to be stored in plaintext, or in a reversible encrypted format, which is somewhere between borderline insecure and rather complicated. Therefore, we opted to store the credentials in the application properties and add suitable Java beans for these settings.

```
|- GenerationCli.java
|- JoinCli.java
|- LayoutInfoCli.java
```

In its core, each database has a set of elements the configuration needs to know. These include the name, location, username, password and the usable Java driver. The record class `DatabaseConfig` implements these.

```java
public record DatabaseConfig(
        String url,
        String username,
        String password,
        String driver_class_name,
        String database_name
) {}
```

Listing 19: Custom Spring configuration for databases

All configurations for KnowledgeBase, TemplateDB (labelled as ‘data target’) and the list of sources are loaded into an object of `DatabaseProperties`. The class features an annotation, binding all entries in the property file with the prefix `eu.sternbauer.db-config` to it. The name of elements between the file and fields must match, such that Spring can automatically map the items. Thanks to the framework, these properties can be injected where they are relevant.

```
@ConfigurationProperties(prefix = "eu.sternbauer.db-config")
public record DatabaseProperties(DatabaseConfig knowledgebase,
                                 DatabaseConfig datatarget,
                                 List<DatabaseConfig> datasource) {
}
```

Listing 20: Custom configuration for application properties under the given key

The listing below shows the `application.properties` entries required to run the script generation properly. Notice the prefix and property names matching with the Java classes. The only field not filled is `database_name` for KnowledgeBase, as it already known.

```
eu.sternbauer.db-config.knowledgebase.url=jdbc:<server-
type>://<address>:<port>;database=<db>;trustServerCertificate=true
eu.sternbauer.db-config.knowledgebase.username=<username>
eu.sternbauer.db-config.knowledgebase.password=<password>
eu.sternbauer.db-config.knowledgebase.driver-class-name=<jdbc driver>

eu.sternbauer.db-config.datatarget.url=<address>,<port>
eu.sternbauer.db-config.datatarget.username=<username>
eu.sternbauer.db-config.datatarget.password=<password>
eu.sternbauer.db-config.datatarget.driver-class-name=<jdbc driver>
eu.sternbauer.db-config.datatarget.database_name=<KB name of data target>

eu.sternbauer.db_config.datasource[0].url=<address>,<port>
eu.sternbauer.db_config.datasource[0].username=<username>
eu.sternbauer.db_config.datasource[0].password=<password>
eu.sternbauer.db_config.datasource[0].driver_class_name=<jdbc driver>
eu.sternbauer.db-config.datasource[0].database_name=<KB name of data source>
```

Listing 21: Basic application.properties content, containing all essential keys

With this layout and naming, the default configuration for a JDBC connection in its naming schema is not filled (`spring.datasource`). Without any additional code extension, Spring cannot recognize the KnowledgeBase connection properties. By design, it looks for a method called `dataSource` and takes the returned 'DataSource' object as the configuration. As the default `spring.datasource` is not used, the chosen approach requires an overwritten method with the same name and return value. Registering it with Spring, and at the same time overloading the default, is done rather easily with the `@Bean` annotation. This works as the custom method receives a higher priority and is the go-to function for the method named `dataSource`.

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setUrl(properties.knowledgebase().url());
    dataSource.setUsername(properties.knowledgebase().username());
    dataSource.setPassword(properties.knowledgebase().password());
    dataSource.setDriverClassName(properties.knowledgebase().driver_class_name());
    return dataSource;
}
```

Listing 22: Overwriting the default method implementation to provide the configuration from a different property

Package Generation

The most interesting parts of the program reside within the generation package. They manage the script generation with data from KnowledgeBase and write the SQL into the respective files. The implemented public methods inside the `service` package offer the generation routines to the CLI, and others for file interactions.

```
|- internal
|  |- mapper
   |  |- GenerationMapper.java
   |- model
      |- CondFinder.java
      |- FullLayoutElement.java
      |- JoinFinder.java
      |- SimpleColumn.java
      |- SimpleDb.java
      |- SimpleTable.java
|- service
   |- GenerationFileService.java
   |- GenerationService.java
   |- LoadingTemplateService.java
```

Script generation is one of the most complex parts of the program, and the DTOs returned from the service layer in KnowledgeBase are not always the best tool for a job. Therefore, `Generation` has its range of data classes, stored in the package `model`. Most notable, there are representations for a database, table and column, and all are without a list of their associated child elements, if they had any to begin with. The class `SimpleTable` additionally saves a string `shortname`. This is the name given in the FROM/JOIN clause of a statement (like `FROM dbo.Table1 t1`), to precisely reference the table.

GenerationService

Within `GenerationService` lies the core functionalities of the generator. The methods manage the act of SQL script generation for defined sources and the target database, TemplateDB. It generates the necessary SQL statements tailored towards a specific database configuration and its schema, addresses tasks such as managing linked servers, constructing joins, generating ‘INSERT’ and ‘SELECT’ statements, and applying ‘WHERE’ conditions.

To do as such, it interacts with many other services, most notable `DatabaseProperties` for database credentials, `LayoutInfoService` for schema information and `TransformationService` to get mappings and transformations required. Other dependencies include `JoinService` for handling relations between elements in the form of joins, `ConditionService` for necessary 'WHERE' conditions, and `TableSelectService` for managing custom table sub-select in 'FROM' and 'JOIN' clauses. Finally, the service also uses the two sub-services specified in their own modules, which are discussed below. `GenerationFileService` for handling file creation, content buffering and writing, as well as `LoadingTemplateService` to append loading statements for transferring data from the staging area to TemplateDB.

The main functionality lies within the method `generateEtlLoadingStatements`. It starts by validating the configuration of data sources and destinations, notifying the user on success or error. Next, all schema information necessary are retrieved with the help of `findFullLayoutByDatabaseName`. This is followed by the 'shortcode' generation. On success, the method generates linked server entries, allowing the script to pull and push data from and to remote MSSQL servers.

```
EXEC sp_addlinkedserver
  @server='<ServerName>',
  @srvproduct = '',
  @provider = 'SQLNCLI',
  @datasrc = '<ip,port>';

EXEC sp_addlinkedsrvlogin
  @rmtsrvname='<ServerName>',
  @useself = 'FALSE',
  @locallogin = NULL,
  @rmtuser = '<user>',
  @rmtpassword = '<password>';
```

Listing 23: Linked server SQL statements for location and authentification

Next comes the setup of necessary, and helpful, mappings between IDs and the elements. This is for faster and less complicated lookup operations and includes the relevant transformation and mapping information. The following step takes care of 'INSERT' and 'SELECT' generation, which occur at the same time, as they are closely tied to each other. What is meant to be inserted must be selected from somewhere. The program iterates over all transformations, takes the affected columns at the destination, looks up their counterparts with potential transformations at the source-side and concatenates them into strings. Lastly, these are added to the content buffer in `FileGenerationService` with the 'INSERT' statement first, followed by its 'SELECT' counterpart.

Next, the generator creates SQL join statements by leveraging the `findJoins` helper method to identify the necessary relationships between tables. This is probably the most complex task, as the order of joins is relevant. Each table joined can only reference those entries already showing up before it, not after, in its 'ON' condition, and therefore the correct order must be determined. To illustrate, assume joining two tables `t1` and `t2` with `t1.id = t2.id`. If the method generated `JOIN table2 t2 ON t2.id = t1.id`, and only inserted `JOIN table1 t1 ON <valid statement>` afterwards, this would result in an error message during execution. The other way around, however, will result in a working SQL script.

Finding the correct sequence is a task where each join gets attributed two things, the offers and the additional requirements. Each join has two so-called 'offerings', namely the tables it joins together, call them `t1` and `t2`. These are labelled as such as they offer these tables to be used later on by other joins. Second, the added requirements are all the tables apart from `t1`/`t2` that are used in the 'ON' statement of the join, therefore which the relation cannot self-satisfy. The algorithm for joining the tables keeps a set of all already joined tables and compares the known entries to the requirement. If, and only if, the known tables can satisfy the added requirements and one of the two tables to be joined is known, the join may be added.
When none of the two sides are known, or the requirements cannot be fulfilled, the algorithm will check the next join in line, and so forth. One special case occurs when both tables (`t1`/`t2`) are already known. Then the algorithm looks for through the processed joins to find the suitable one and concatenates the additional 'ON' condition with the existing one for the table combination.

```java
public record JoinFinder(KBJoinTableDTO join, List<Integer> addedRequirements) {
  public int getT1Offer() { return join.table1(); }
  public int getT2Offer() { return join.table2(); }
}
```

Listing 24: Helper class 'JoinFinder' with the join object and a list of additional requirements apart from the two joined tables

We fully understand that this is a rather abstract description, and therefore we would like to provide an example of a hands-on situation. Take three tables `Table1`, `Table2` and `Table3`, where `Table2` joins with `Table1` like `ON t1.id = t2.id1` and with `Table3` like `t2.id3 = t3.id`. `Table3` joins with `Table1` on `t3.id1 = t1.id`. This means three join entries in total, as each join with another tables in stored separately. There is none for `Table 1`, two for `Table2` (the program must consider that both have to be fulfilled to work) and one for `Table3`. The finished SQL script should look something like this:

```sql
FROM Table1 t1
JOIN Table3 t3 ON t3.id1 = t1.id
JOIN Table2 t2 ON t1.id = t2.id1 AND t2.id3 = t3.id
```

When only `t1` is already known and `Table2` with `Table3` should be joined next, the algorithm will find that not both of the required tables (only `t1` is) for `t2` are available, and the join will be postponed, i.e. re-queued.

```sql
FROM Table1 t1
```

In this scenario, the method will find `Table3` to be suitable to join on `Table1`, and adds it.

```sql
FROM Table1 t1
JOIN Table3 t3 ON t3.id1 = t1.id
```

Afterwards, the same holds true for `Table2` with the now available `Table1` and `Table3`, and the new JOIN is added, let's assume it is between `t1` and `t2`.

```
FROM Table1 t1
JOIN Table3 t3 ON t3.id1 = t1.id
JOIN Table2 t2 ON t1.id = t2.id1
```

Next, the join between `Table2` and `Tabl3` reveals that both of the tables are already used in the script. Therefore, when the last entry with one of the two tables in a 'JOIN' is searched for, it will find `Table2` to be used last. The 'ON' condition of `t2` is therefore appended with an 'AND' to the previous one. The program must find the last table used, as otherwise the clauses would reference a table that is only visible later on in the join, and produce failing code.

```
FROM Table1 t1
JOIN Table3 t3 ON t3.id1 = t1.id
JOIN Table2 t2 ON t1.id = t2.id1 AND t2.id3 = t3.id
```

None of these cases has additional requirements, as all conditions for `Table2` and `Table3` only list columns from the tables that are joined. However, an ON-condition could also contain something like `t2.sum > t1.sum + t3.sum` for the relation between `Table1` and `Table2`. Then `Table3` finds its place as an entry in the list of additional requirements, and that table must already be joined before this specific 'ON' condition can be satisfied. Coincidently, it is in the example and the resulting code would look very much alike, except for the altered condition.

```
FROM Table1 t1
JOIN Table3 t3 ON t3.id1 = t1.id
JOIN Table2 t2 ON t2.sum > t1.sum + t3.sum
```

Join statement generation is split into two parts. First, it determines which tables necessary do not have a specialized ON-clause and can therefore be used for the 'FROM' section. The second part finds the correct order of elements for joining. This section of code does also evaluate eventual sub-select statements. As discussed beforehand, this is done by replacing placeholders with the actual column references and concatenating the different parts together. Last, all parts are written to the content buffer.

In case KnowledgeBase contains faulty data and the generation runs into an endless loop, for example when determining the order of joins, the method has an included failsafe and terminates with an error message.

The last part of dynamic code generation concerns conditions, i.e. the 'WHERE' clauses. As before, the constructions of these work by replacing the placeholders with the actual columns as stored in KnowledgeBase and saving the work into the content buffer. Subsequently, it sets the file name to the source database with the suffix `_load_script` and tells the file service to write the file.

As the last step, the generator writes the SQL for data transfer between the staging table and the actual TemplateDB tables. The data in 'FillOrder' specifies the correct ordering and the predefined queries inside 'LoadingTemplateService', more on that in just a bit, are inserted into the content buffer one after the other.

GenerationFileService

The `GenerationFileService` class is designed to manage the writing and handling of the script files. It buffers the content, allows for a customized file name, when required, and file writing to the user's home directory. Therefore, this class encapsulates the necessary logic for these operations and is separated from the actual generation code, to not only follow good coding practices, but also allow for an eventual reuse.

The class maintains three fields: `userHome`, `fileName`, and `sb`. The first stores the user's home directory, read from the according system property. The second holds the name of the file, writable via the setter method. The last field is a `StringBuilder` which acts as the content buffer.

Various methods are available in this class. `reset()` empties the StringBuilder and is mandatory to be called before each run. This is particularly important because Spring may inject the same instance multiple times within a session, and this preservation of content between calls leads to errors. `addContent(String content)` appends the passed `content` to the builder and automatically adds a newline (`\n`), stripping the generation service of the need to add newlines all the time.

Finally, `writeOut()` takes the StringBuilder's content and writes it to the file with a given name and the ending `.sql` into the user's home directory. When no name is specified, or it equals to an empty string, the name is set to `etl_loading`. The writing utilizes Java NIO using `Files.writeString` with added options, including UTF-8 encoding and overwriting any already existing content inside the file with the StringBuilder's one.

LoadingTemplateService

`LoadingTemplateService` orchestrates the movement of data from TemplateDB staging area to its tables. This service utilizes predefined SQL templates, dynamically modifies them based on the provided remote server name, and organizes the execution order of these operations. The class uses `FillOrderService`, which provides the necessary filing order of tables.

The `LoadingTemplateService` class contains multiple predefined SQL statements as constants, each tailored to a specific target table in the database. These include `universityStatement`, `courseStatement`, `examStatement`, `studentStatement`, `studyStatement`, and `registrationStatement`. Each statement transfers data from a staging table into a corresponding TemplateDB table and includes logic for updating references in the staging table to point to the newly inserted data. Summarized, the operations facilitate the flow of data as it transitions from staging to its final destination.

The class constructor initializes a mapper, `tableStatements`, which associates TemplateDB table names with their respective SQL templates. This map allows dynamic retrieval of SQL statements during execution. By using `FillOrderService` via dependency injection, it is ensured that the loading process follows the logical dependencies between tables, as dictated by the ordering.

The primary functionality is encapsulated in the `getStageToTablesStatement(String templateRemoteServerName)` method, which takes the remote server name of the source as an argument. This method generates the SQL script to move data from the staging table to the TemplateDB tables in the correct sequence. It starts by fetching the list of fill orders from the `FillOrderService` and sorts the tables by their assigned priority. For each table, it retrieves the corresponding SQL statement, replaces placeholders within the statements with the provided remote server name, and appends the modified statement to a `StringBuilder`. The returned string is the resulting script with the correct ordered sequence of SQL commands.

Package Cli

As described in the architectural part, this program provides a small CLI for user interaction. The internal layout remains on the simpler side as well, with a service class, a toolkit class, and a CLI class for each component callable on the shell, labelled with the suffix 'Cli'. Each class features the `@ShellComponent` annotation and the methods are `@ShellMethod`, as seen in *Section 5.2.2*. The command naming is, if not mentioned explicitly in this section, further elaborated during program evaluation in *Section 6.1.2*.

```
|- internal
|  |- CliTools.java
|- service
|  |- CliService.java
|- GenerationCli.java
|- JoinCli.java
|- LayoutInfoCli.java
```

In the `service` package resides a class offering all business logic for the UI. Inside `CliService`, the combination of methods from various KnowledgeBase services and the configuration takes place. All CLIs reference this service implementation, as well as a small tool collection in `CliTools`. The latter includes options for pretty-printing lists to the shell and getting a valid selection, via an entered number, from a range of listed options. The return value can be either the string representation of the selected entry or its index.

`LayoutInfoCli` processes information stored inside the property file and KnowledgeBase. It shows the stored entries from both sources, and an additional shell method combines them, showing which sources had both their semantic information and credentials ingested into the system. Those are then available for generation.

`GenerationCli` handles file generation in its `generateETL` method. It takes one optional argument, namely the data source name. When it is absent, a list of available sources is offered. Otherwise, the provided name is taken and passed to the script generator.

`JoinCli` allows the end-user to view the storage information regarding relations between data. Typically, these are the foreign key relationships over which joins are then, hence the name. The options are to view all joins, delete a join selected from a list, and create a new relation between two elements. The latter proceeds to ask several questions, including the tables, mappings, referenced columns, among others.

5.3. Implementation conclusion

This chapter elaborated on the implementation of the explained design of the previous chapter. It proposes the data warehouse as the final data location, alongside TemplateDB as an intermediary step containing the domain knowledge. TemplateDB also serves as the primary destination for the dynamic part of script generation, as the following steps are rather non-volatile by design. Encoding the knowledge itself is the job of KnowledgeBase, which implements various ways to store the transformation and mappings, relations between items, conditions and sub-selects. Furthermore, a source database named ‘JKU’ is introduced as well.

The generation software itself is based on Java and the Spring framework. It makes use of the framework’s many modules, including Data JDBC for database access, Shell for building a command line interface, and JUnit for testing. The generation algorithms produce MSSQL server conforming T-SQL with remote server support, automated selection of relevant transformations, mappings, conditions and thereon as well as correct ordering of joins.

The next chapter will shift focus toward evaluating the constructed KnowledgeBase and the accompanying code generation. This evaluation will assess the effectiveness and adherence to design objectives, providing insights into their practical applicability and areas for improvement.

6. Evaluation

Having both the solution design and implementation covered, a demonstration on how these parts work out is the next step. At first, the already shown implementation will be demonstrated on how it works with a single database source. Next, additional example source databases are added, and their information inserted into Knowledge-Base. Finally, it is demonstrated that the system can once again produce code for these freshly added data sources.

6.1. Testing the implementation

This section deals with setting up the database and the programme, storing semantic information in KnowledgeBase, running the program and evaluating the generated output by executing the script on a SQL server. The latter is assumed to be in working order and the user is entitled to execute a script. As discussed in *Section 5.1*, all demonstrator database code comes in the form of database projects. These can be deployed in Azure Data Studio with a single dialogue. The same process loads the demo data into the sources and the country code map. All data is filled into the database layout in the section just mentioned.

6.1.1. Filling KnowledgeBase

The first big task comes with filling information into KnowledgeBase, as without any attempt to generating any (meaningful) script is in vain. Currently, there exists no larger framework accompanying the generator which could extract the structure and relations from the sources. Therefore, this task must be done manually. At the same time, the MSSQL server already stores some meta information in the 'system' tables of each database, which makes extraction a little easier. For example, getting the layout information for the 'JKU' labelled database looks like in *Listing 25*.

```sql
DECLARE @dbName VARCHAR(30) = 'JKU-Student-System';
DECLARE @dbID INT;

-- DB information
INSERT INTO kb.DatabaseInformation
(db_name)
VALUES (@dbName);
SET @dbID = SCOPE_IDENTITY();

-- Table information
INSERT INTO kb.TableInformation
(database_id, [schema], table_name)
SELECT @dbID, s.name, t.name
FROM [JKU-Student-System].sys.tables t
JOIN [JKU-Student-System].sys.schemas s ON s.schema_id = t.schema_id;
```

```sql
-- Column information
INSERT INTO kb.ColumnInformation
(table_id, column_name)
SELECT ti.table_id, c.COLUMN_NAME
FROM [JKU-Student-System].information_schema.COLUMNS c
JOIN kb.DatabaseInformation di ON di.db_name = c.TABLE_CATALOG
JOIN kb.TableInformation ti ON
    c.TABLE_NAME = ti.table_name
    AND c.TABLE_SCHEMA = ti.[schema]
    AND ti.database_id = di.database_id
```

Listing 25: Filling the layout information about the JKU system into KnowledgeBase

The only thing hard-coded inside the script is the database name, which further on identifies the items when searched for during generation in the software. This is also the name that must match with the credential record in the application properties, see more about that in *Section 5.2.5*. Integrating join information is more complex, but still doable with a large query for this setting, where some requirements such as support for self-references or multiple references of the same table do not apply. For some sources, some joins must be manually removed to keep KnowledgeBase in a functioning state, which involved some trial and error. Conditional statements and subselects are left to the engineer as of now to properly insert. Below are two code examples in *Listing 26* and *Listing 27*, the first for a 'WHERE' clause, the second for a sub-select.

```sql
-- WHERE clause
SET @col_id = (SELECT column_id FROM kb.ColumnInformation ci JOIN kb.TableInformation ti
ON ci.table_id = ti.table_id WHERE ci.column_name = 'ordering' AND ti.database_id = @dbID
AND ti.table_name = 'Teaching' AND ti.[schema] = 'dbo');
INSERT INTO kb.ColumnCondition
(column_id, expression)
VALUES (@col_id, '$col = 1 OR $col IS NULL')
```

Listing 26: Telling KnowledgeBase about a required WHERE-clause

```sql
-- Subselect
INSERT INTO kb.TableSelect
(comment)
VALUES ('Select to convert a unix timestamp to datetime');
DECLARE @ts_id INT = SCOPE_IDENTITY();

INSERT INTO kb.TableSelectElement
(table_select_id, expression, as_name)
VALUES (@ts_id, 'dateadd(S, [$col1], ''1970-01-01'') AS ''$as_col'', [$col2], [$col3]',
@as_col)
DECLARE @tse_id INT = SCOPE_IDENTITY();

INSERT INTO kb.TableSelectMap
(column_id, element_name, table_select_element_id)
VALUES (@as_col, '$col1', @tse_id),
(@col2, '$col2', @tse_id),
(@col3, '$col3', @tse_id)
```

```sql
-- Update table with subselect ID
UPDATE ti
SET ti.table_select_id = @ts_id
FROM kb.TableInformation ti
    WHERE ti.table_id = @tbl_id
```

Listing 27: Generating a sub-select query for a table

As it can be seen, these queries are rather lengthy and currently done one-by-one. An integration into a larger ETL framework with automated feature extractions from all sides involved could significantly accelerate the process and help to avoid mistakes. The same holds true for the actual mappings. For the purpose of evaluation and validation, all items were done by hand. The following code in *Listing 28* highlights one such mapping from the 'JKU' source to the gender field in TemplateDB. First, it selects both required columns and then inserts the mapping and transformation information into the designated tables.

```sql
SET @src_col = (SELECT column_id FROM kb.ColumnInformation ci JOIN kb.TableInformation ti
ON ci.table_id = ti.table_id WHERE ci.column_name = 'gender' AND ti.database_id = @source
AND ti.table_name = 'Student' AND ti.[schema] = 'dbo');
SET @target_col = (SELECT column_id FROM kb.ColumnInformation ci JOIN kb.TableInformation
ti ON ci.table_id = ti.table_id WHERE ci.column_name = 'gender' AND ti.database_id = @target
AND ti.table_name = 'Student' AND ti.[schema] = 'dbo');

INSERT INTO kb.TransformationOperation
(column_id, source_database, expression)
VALUES (@target_col, @source, '(Case When $col1 = 0 THEN ''Female'' when $col1 = 1 THEN
''Male''Else ''Non-Binary'' END)');
INSERT INTO kb.TransformationFromMap
(transformation_op_id, column_id, element_name)
VALUES (SCOPE_IDENTITY(), @src_col, '$col1');
```

Listing 28: An example data transformation and mapping for a student's gender

6.1.2. Running the query generation

Before one can start the generator software, the application properties require some entries for access to KnowledgeBase, TemplateDB (labelled the 'data destination') and all available sources. For more on this topic, see *Section 5.2.5* at "Package Configuration". For example, the source 'JKU' could look like the following listing, without the line break at `driver_class_name`:

```
eu.sternbauer.db_config.datasource[0].url=localhost,1433
eu.sternbauer.db_config.datasource[0].username=sa
eu.sternbauer.db_config.datasource[0].password=This_i5_A_Password
eu.sternbauer.db_config.datasource[0].driver_class_name
  =com.microsoft.sqlserver.jdbc.SQLServerDriver
eu.sternbauer.db-config.datasource[0].database_name=JKU-Student-System
```

Listing 29: Setting application properties for the first data source

Once completed, and all databases setup and running, the program is ready to launch from the development environment after building. The whole set of commands can be looked up in *Section 5.2.2*. First and foremost, a sanity check for the available sources with the CLI command `show available db`, where it checks the databases in KnowledgeBase and system properties. It should yield a result like:

```
Available matching databases in the KnowledgeBase and application properties
(0) - JKU-Student-System
```

If this fails, checking the databases in properties via `show property db` and `show kb db` are the best options for error debugging, but this is not the point of this section and therefore not elaborated in any length. Further verification of KnowledgeBase, related to all joins, is available with the `show joins` command. It prompts the user to select a stored database and then displays all relationships known for the selected options. If one is missing, it may be created with `create join`, which itself prompts some additional questions to fill in the items, whereas `delete join` offers a removal service for individual entries.

Generation itself starts with the `gen etl` command. It does take an optional parameter with the data source name, or present a dialogue with all available options. Once selected, the program starts with the generation as described in *Section 5.2.5* under 'Package Generation'. The code does produce a couple of informational outputs to the shell, hopefully informing about the successful completion, with an entry showing the location of the just generated script file. Error messages may stem from and be about various topics, such as missing elements in KnowledgeBase, properties, file access errors, incorrect join information, and so on. Just below is an exemplary output from a successful run with the JKU source, stripped of the timestamp and some other prefixes coming from the Spring framework for more clarity.

```
shell:>gen etl
(0) - JKU-Student-System

Enter a number (0-0): 0
: Generating ETL loading statements for JKU-Student-System
: Found suitable configurations for data source and data target.
: Found suitable database information for JKU-Student-System
: Found suitable database information for TemplateDB
: Found information in knowledge base about source and destination.
: Generated linked server
: Generated insert and select statement(s)
: Generated FROM and join statements
: Generated WHERE clauses
: Generated stage to table loading statements
: Finished generating ETL statements
: ETL script located at: <UserHome>/JKU-Student-System_load_script.sql
```

6.1.3. Executing the query

Opening the generated file in any environment able to execute a MSSQL server compatible script is sufficient for a simple testing of the capabilities. For this purpose, we will utilize Azure Data Studio and start the script. It will create the connections, empty the staging area of TemplateDB, fill it with content from the source and then move the entries to the actual tables inside TemplateDB. Please note that multiple runs can produce primary key violations, as the program currently assumes the tables to be empty. After the execution, checks with `SELECT * FROM stage.Stage_TemplateDB;` and on the TemplateDB tables such as `Course`, `Student`, `Registration` show the inserted data. As an example, the code in *Listing 30* joins all tables together and therefore displays each complete set of data as a single row. Please note that the stage will be cleared before each loading process and does not present a suitable persistent storage area.

```
SELECT *
FROM dbo.Registration reg
JOIN dbo.Student s  ON reg.student_sk = s.student_sk
JOIN dbo.Course c ON reg.course_sk = c.course_sk
JOIN dbo.Exam e ON reg.exam_sk = e.exam_sk
JOIN dbo.Study st ON reg.study_sk = st.study_sk
JOIN dbo.University u ON u.university_id = s.source_id AND u.university_id = c.source_id
  AND u.university_id = e.source_id AND u.university_id = st.source_id
WHERE u.name_nk = 'JKU-Student-System'
```

Listing 30: Returns the entries of TemplateDB as rows for manual verification

This output lets the user verify that all data is loaded completely and without error through the generated code for a single source.

6.2. Adding new sources

The implementation has proven to work for and with one data source. To evaluate whether the system also works for another origin of data with a different layout, and therefore properly supports multiple sources, more example databases are needed. For this reason, two additional sources, named after two more well-known universities in Austria, were designed. Again, as with the data source for JKU, we will not go into every detail, but only the most relevant parts.

For both additional sources, the basic knowledge insertion process into KnowledgeBase stays the same as with the first item. More on this, and in greater detail, may be looked up in *Section 6.1.1*.

6.2.1. Additional data source 'MedUniWien'

Students of a medical topic in Austria have a more rigid study plan than other university students. In human medicine, subjects are often coupled with a given semester and progress is often strongly tightly coupled to previous successful course completion. The database layout therefore also adapts to these prerequisites. As a speciality, all naming

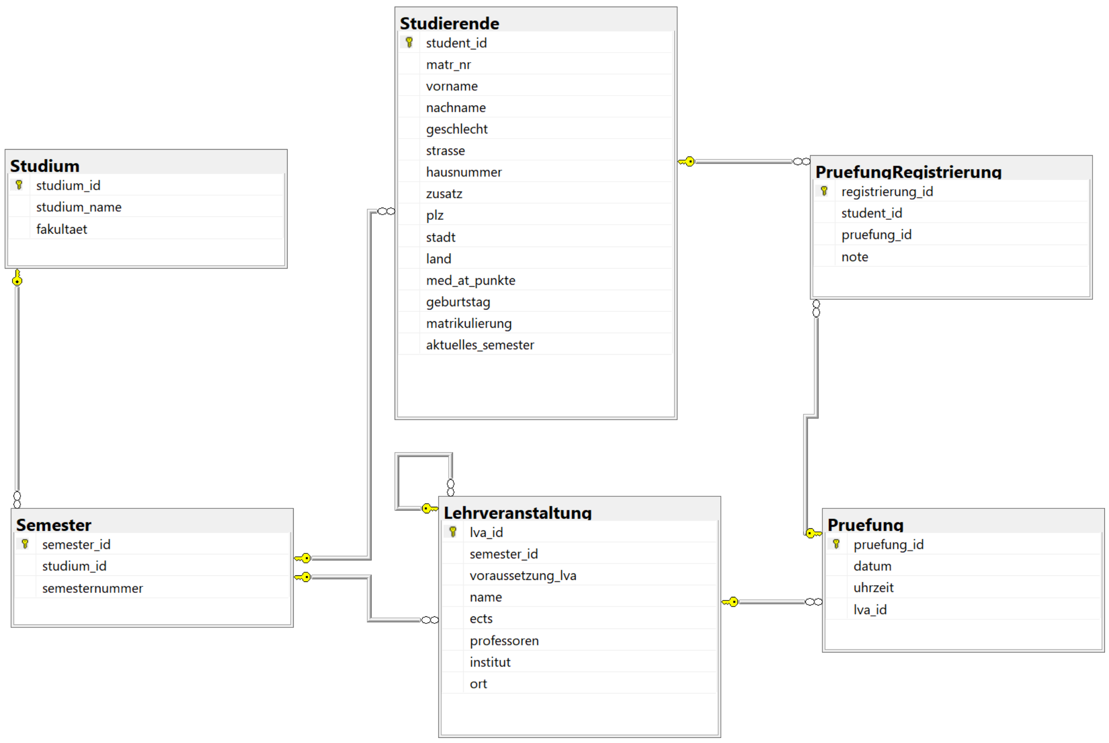


Figure 19: Overview of all tables inside the source implementation called MedUniWien

is done in German, to show that different namings, even in the another language, do not hinder the system in functioning correctly.

Layout and construction

The layout can be seen in *Figure 19*. Central to the database is the `Studierende` table, storing all information about a student. A string encodes the gender in the column `geschlecht` and the location of residence resides in `stadt` and `land`. Each student takes the courses from one semester, which is why the student holds a reference to the `Semester` table. The different courses are then saved into the `Lehrveranstaltung` table. The latter also records the number of ECTS, whereas time-related information finds its place in the exam table `Pruefung`. The month, semester, year may be extracted from the date stored with each exam. Students may register for an exam, and the grade is saved with the registration.

Each course is associated with one semester, and each semester is in turn associated with a study, located at a university faculty. The professors are stored as one string directly at the course level, without any additional tables and links. As courses may have prerequisites, the table is self-referencing itself when applicable. The exam registration table `PruefungRegistrierung` records the achieved grade.

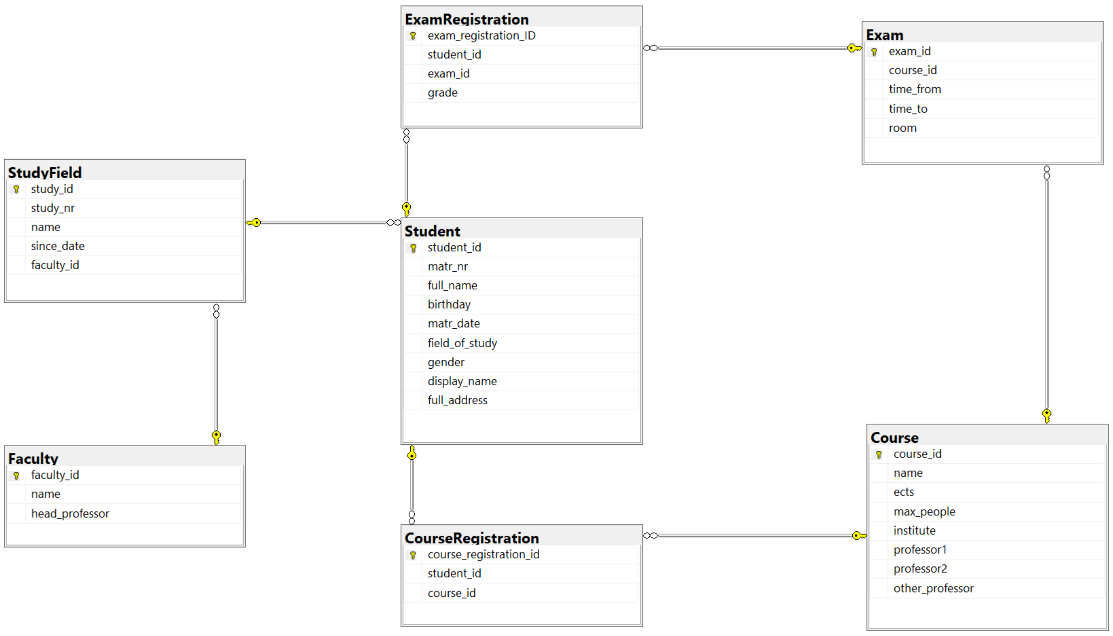


Figure 20: Overview of all tables inside the source implementation called TUGraz

6.2.2. Additional data source ‘TUGraz’

The second additional source lends its name from the renowned technical university in Graz. This time, the database contains English names only again, but comes with a set of its own design choices. More on them in the upcoming paragraphs.

Layout and construction

The layout is shown in *Figure 20*. All student data finds its place in the fittingly named `Student` table. Each student inscribes for one field of study, which in turn has an association with one `Faculty`. A student may register for one or more courses, as seen in `CourseRegistration`, even multiple times for the same course when they have to retake it after a failed attempt, or because they simply liked it a lot. An exam (table `Exam`) takes place for one course, and obviously, there may be more than a single one of a course. The student registers for an exam via `ExamRegistration`, so course and exam registration are decoupled in this system.

There are some caveats when it comes to the used data types and combination into single fields. For example, all data entries store their information as a Unix timestamp, therefore counting the seconds since the 1st January 1970. To not stop in the near future of 2038 with counting, the dates are all stored as 64 bit values, adding another 292 billion years. This should prevent most problems. Students also record their address as a singular entry, not split it into different database columns. The relevant pieces have to be extracted from the string by splitting. A simple boolean value represents a student’s gender. To allow non-binary options, a null value indicates something apart from the binary system. The achieved grade lays with the exam registration, whereas the number of ECTS points are associated with the courses.

6.3. Testing new sources

The basic procedure stays the same as before in *Section 6.1.2* for generating the scripts and *Section 6.1.3* for evaluation. Therefore, the new database credentials and connection options are inserted into the application properties file. After starting the program and executing `get etl`, select the respective database and verify with the output that the generation completed successfully. Next comes loading the file content into a suitable environment, in our case Azure Data Studio, and running it. For both cases, the execution does not produce any errors. Again, checking the content of TemplateDB does provide an additional layer of verification, which for both MedUniWien and TUGraz shows that all items are processed correctly.

At the same time, the SQL script generation for the source labelled JKU ran again to verify that sources do not influence each other. A check for differences in between the old and new file revealed none. This shows the capability of the system to handle multiple sources without affecting knowledge in the system. As a last step, executing the Jupyter notebook with the ETL steps for TemplateDB to the DWH brings the data into the warehouse itself. This is, as of now, not implemented yet for the generated code, as the sources do not influence it.

6.4. Evaluation conclusion

To summarize, this chapter shows the capabilities of KnowledgeBase and the generator. Combined, these components produce ETL code in the form of SQL script files, ready for execution with support for remote access to MSSQL servers, and staging. The previous sections demonstrated the capabilities for a single implemented database first and two others afterwards, showing that the system can very much handle different sources without trouble.

In the future, extensions to the program will be required for a more seamless integration within an upscaled environment. This includes continuous data integration into TemplateDB and the data warehouse with techniques such as slowly changing dimensions. Furthermore, checks on data integrity and quality are done manually in this thesis, but cannot be done by hand on a larger scale. This requires an automated system in coordination with the script execution, possibly bringing up the need for separate integration testing or function calls to test routines from the generated code.

7. Conclusion and Outlook

This thesis has discussed many aspects of adaptive ETL processing and proposes further automation of core tasks. Starting off with general aspects and the problem statement, related works were discussed. This is followed by the proposed solution design, its implementation and evaluation with adding new source systems. The thesis discussed important topics such as the need for an ontology design separated into TemplateDB, for the domain knowledge, and KnowledgeBase, to store the concrete instances. Furthermore, it suggested a way to automate code generation, which opens the door to further implementations as a core component into a larger ETL framework or system.

7.1. Conclusion

The goals of this thesis, discussing, designing and implementing a way for dynamic ETL processing, supported by ontological data, for data warehouses, are completed successfully. The different chapters evaluate on the many steps required, from defining the basics down to implementation and evaluation, and break the larger parts down into the finer details.

After the introduction to the problem statement in *Chapter 1*, the thesis starts defining the core concepts in *Chapter 2*, which include knowledge, ontologies, and ETL processing. Subsequently, it goes into detail on current and past research of the topics, and emphasizes works integrating and implementing different theoretical concepts into a functioning system. This defines the baseline this work builds upon. The next chapter, *Chapter 3*, takes a swing at work related to the basic concepts. They paint a broader picture of the topic, without directly linking to the thesis, by discussing other attempts and aspects of ETL and ontologies.

Solution design starts in *Chapter 4*, with outlining the purpose and requirements. It proceeds to list the system's components, such as the data warehouse, TemplateDB, KnowledgeBase and sources as well as the software generator. The chapter highlights how information is encoded in these components. This is followed by an explanation of how these different components interact with each other and where human interaction should be required.

The implementation details find their place in *Chapter 5*, starting with the languages, software and tools, proceeding with the databases and finishing with the generator program. Using Microsoft's SQL server and Java, two established tools, the proposed solution, along with the test setup for data sources and data warehouses, is developed. Due to the modular software design, an extension with more capabilities or a different type of semantic information storage could be proceeded with, when necessary.

In *Chapter 6*, the system shows its capabilities and flexibility by performing the script generation successfully. The chapter further emphasizes how data for the ETL processing gets processed, adapted and ingested into KnowledgeBase. Later on, two more sources are introduced and the system's functionality is tested once again successfully. The script files themselves are tested by running them on a SQL server and checking the filled tables in TemplateDB and the data warehouse after script execution (source to TemplateDB). This provides a satisfying outcome and a solid foundation to build upon.

7.2. Outlook

The possibilities for further work on the system are neither rare nor miniscule. There are many chances to enhance upon the baseline laid out by the thesis. First and foremost, integrating the generator component into a larger ETL framework deems to be the next bigger step in development, where the system can further prove its value and functionality. At the same time, this generates valuable insights into areas that require further enhancement, possibly some new information required in KnowledgeBase or adding an API to the generator. Support for continuous data integration and techniques such as slowly changing dimension must be added as well in order to work successfully within a production-grade system.

As of now, the system only works with Microsoft's SQL server, whereas other platforms, and especially the different syntax of the respective SQL dialects, are ignored. ETL systems are often operated in heterogenous environments, where different types of database servers must interact. These may include Oracle, PostgreSQL or MySQL, not considering NoSQL options. Therefore, adding an abstraction layer to the software which allows the SQL scripting in different SQL dialects is another open topic and leaves room for improvement.

Storing semantics and an ontology in the form of a relational database is untypical, even though the possible forms of preserving such information are manyfold. It further prohibits the utilisation of automated reasoning on the ontology and upgrade it to a fully fledged knowledge-based system. The implementation of the domain ontology with an ontology language, for example OWL, and adding a reasoning engine could create a knowledge graph. Speaking of, the data is very interconnected and the connections, i.e. the relation between data items, are of great interest. Deploying a graph database instead of an RDB could be a worthwhile choice, depending on the other parts of the deployed software stack.

To summarize, this thesis lays a strong foundation for an automated ETL system while highlighting several avenues for future work. Integrating the generator into a larger ETL framework, supporting diverse SQL dialects, and exploring approaches with ontology languages or graph databases present opportunities to enhance the functionality and adaptability. These developments could further solidify the system's value and address complex, heterogeneous environments for data processing.

Bibliography

[1] A. L. Antunes, E. Cardoso, and J. Barateiro, "Incorporation of Ontologies in Data Warehouse/Business Intelligence Systems - A Systematic Literature Review," *International Journal of Information Management Data Insights*, vol. 2, no. 2, p. 100131, 2022, doi: https://doi.org/10.1016/j.jjimei.2022.100131.

[2] A. Dhaouadi, K. Bousselmi, M. M. Gammoudi, S. Monnet, and S. Hammoudi, "Data Warehousing Process Modeling from Classical Approaches to New Trends: Main Features and Comparisons," *Data*, vol. 7, no. 8, p. 113, 2022, doi: 10.3390/data7080113.

[3] P. S. Diouf, A. Boly, and S. Ndiaye, "Variety of data in the ETL processes in the cloud: State of the art," in *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*, 2018, pp. 1–5. doi: 10.1109/ICIRD.2018.8376308.

[4] D. Skoutas and A. Simitsis, "Designing ETL processes using semantic web technologies," in DOLAP '06. Arlington, Virginia, USA: Association for Computing Machinery, 2006, pp. 67–74. doi: 10.1145/1183512.1183526.

[5] D. Skoutas and A. Simitsis, "Ontology-based conceptual design of ETL processes for both structured and semi-structured data," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 3, no. 4, pp. 1–24, 2007.

[6] R. Gacitúa, J.-N. Mazón, and A. Cravero, "Using Semantic Web technologies in the development of data warehouses: A systematic mapping," *WIREs Data Mining Knowl. Discov.*, vol. 9, no. 3, 2019, doi: 10.1002/widm.1293.

[7] D. Skoutas, A. Simitsis, and T. K. Sellis, "Ontology-Driven Conceptual Design of ETL Processes Using Graph Transformations," *J. Data Semant.*, vol. 13, pp. 120–146, 2009, doi: 10.1007/978-3-642-03098-7\ 5.

[8] W. H. Inmon, *Building the Data Warehouse*. John Wiley & Sons, Inc., 2005. [Online]. Available: https://books.google.at/books?id=QFKTmh5IFS4C

[9] R. Kimball and M. Ross, *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[10] L. Jiang, H. Cai, and B. Xu, "A Domain Ontology Approach in the ETL Process of Data Warehousing," in *2010 IEEE 7th International Conference on E-Business Engineering*, 2010, pp. 30–35. doi: 10.1109/ICEBE.2010.36.

[11] J. Chen *et al.*, "Big data challenge: a data management perspective," *Frontiers Comput. Sci.*, vol. 7, no. 2, pp. 157–164, 2013, doi: 10.1007/s11704-013-3903-7.

[12] R. Kimball and M. Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd ed. Wiley Publishing, 2013.

[13] R. Kimball, M. Ross, W. Thornthwaite, J. Mundy, and B. Becker, *The Data Warehouse Lifecycle Toolkit*, 3rd ed. Wiley Publishing, 2013.

[14] R. Kimball and J. Caserta, *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming and Delivering Data*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2004.

[15] J. Chakraborty, A. Padki, and S. K. Bansal, "Semantic ETL — State-of-the-Art and Open Research Challenges," in *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*, 2017, pp. 413–418. doi: 10.1109/ICSC.2017.94.

[16] S. Bergamaschi, F. Guerra, M. Orsini, C. Sartori, and M. Vincini, "A semantic approach to ETL technologies," *Data Knowl. Eng.*, vol. 70, no. 8, pp. 717–731, 2011, doi: 10.1016/j.datak.2011.03.003.

[17] J. Villanueva Chávez and X. Li, "Ontology based ETL process for creation of ontological data warehouse," in *2011 8th International Conference on Electrical Engineering, Computing Science and Automatic Control*, 2011, pp. 1–6. doi: 10.1109/ICEEE.2011.6106642.

[18] Computer Hope, "Semantics." Accessed: Dec. 17, 2024. [Online]. Available: https://www.computerhope.com/jargon/s/semantics.htm

[19] A.-C. Boury-Brisset, "Managing Semantic Big Data for Intelligence," in *Proceedings of the Eighth Conference on Semantic Technologies for Intelligence, Defense, and Security, Fairfax VA, USA, November 12-15, 2013*, K. B.

Laskey, I. Emmons, and P. C. G. da Costa, Eds., in CEUR Workshop Proceedings, vol. 1097. CEUR-WS.org, 2013, pp. 41–47. [Online]. Available: https://ceur-ws.org/Vol-1097/STIDS2013_T06_Boury-Brisset.pdf

[20] L. Ehrlinger and W. Wöß, "Towards a Definition of Knowledge Graphs," in *Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems - SEMANTiCS2016 and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS'16) co-located with the 12th International Conference on Semantic Systems (SEMANTiCS 2016), Leipzig, Germany, September 12-15, 2016*, M. Martin, M. Cuquet, and E. Folmer, Eds., in CEUR Workshop Proceedings, vol. 1695. CEUR-WS.org, 2016. [Online]. Available: https://ceur-ws.org/Vol-1695/paper4.pdf

[21] V. Mulwad, T. Finin, and A. Joshi, "A Domain Independent Framework for Extracting Linked Semantic Data from Tables," *Search Computing - Broadening Web Search*, vol. 7538. in Lecture Notes in Computer Science, vol. 7538. Springer, pp. 16–33, 2012. doi: 10.1007/978-3-642-34213-4_2.

[22] G. A. Miller, "WordNet: A lexical database for English," *Communications of the ACM*, pp. 39–41, 1995, doi: 10.1145/219717.219748.

[23] D. Skoutas and A. Simitsis, "Designing ETL processes using semantic web technologies," in *DOLAP 2006, ACM 9th International Workshop on Data Warehousing and OLAP, Arlington, Virginia, USA, November 10, 2006, Proceedings*, I.-Y. Song and P. Vassiliadis, Eds., ACM, 2006, pp. 67–74. doi: 10.1145/1183512.1183526.

[24] M. Sir, Z. Bradac, and P. Fiedler, "Ontology versus Database," *IFAC-PapersOnLine*, vol. 48, no. 4, pp. 220–225, 2015, doi: https://doi.org/10.1016/j.ifacol.2015.07.036.

[25] T. C. Ong *et al.*, "Dynamic-ETL: a hybrid approach for health data extraction, transformation and loading," *BMC Medical Informatics Decis. Mak.*, vol. 17, no. 1, pp. 1–12, 2017, doi: 10.1186/s12911-017-0532-3.

[26] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly Media, 2020. [Online]. Available: https://books.google.at/books?id=wa7MDwAAQBAJ

[27] I. Eifrem, *Graph Databases, 2nd Edition.* O'Reilly Media, Incorporated, 2015. [Online]. Available: https://books.google.at/books?id=pUOhAQAACAAJ

[28] A.-M. Bonteanu, C. Tudose, and A. M. Anghel, "Performance Analysis for CRUD Operations in Relational Databases from Java Programs Using Hibernate," in *2023 24th International Conference on Control Systems and Computer Science (CSCS)*, 2023, pp. 655–662. doi: 10.1109/CSCS59211.2023.00109.

[29] Oracle Corporation, "Java Software | Oracle." Accessed: Nov. 02, 2024. [Online]. Available: https://www.oracle.com/java/

[30] Microsoft Corporation, "Microsoft Data Platform | Microsoft." Accessed: Nov. 02, 2024. [Online]. Available: https://www.microsoft.com/en-us/sql-server/

[31] Oracle Corporation, "Database | Oracle." Accessed: Nov. 02, 2024. [Online]. Available: https://www.oracle.com/database/

[32] Docker Inc, "microsoft/mssql-server - Docker Image | Docker Hub." Accessed: Nov. 22, 2024. [Online]. Available: https://hub.docker.com/r/microsoft/mssql-server/

[33] DWBIConcepts Team, "Why do we need Staging Area during ETL Load." Accessed: Nov. 22, 2024. [Online]. Available: https://web.archive.org/web/20160513032020/http://dwbi.org/etl/etl/52-why-do-we-need-staging-area-during-etl-load

[34] Broadcom, "Spring | Home." Accessed: Dec. 02, 2024. [Online]. Available: https://spring.io/

[35] Broadcom, "Dependency Injection :: Spring Framework." Accessed: Dec. 07, 2024. [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html

[36] Broadcom, "The IoC Container :: Spring Framework." Accessed: Dec. 07, 2024. [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans.html

[37] Broadcom, "Spring Data." Accessed: Dec. 06, 2024. [Online]. Available: https://spring.io/projects/spring-data

[38] R. Gandhi, M. Richards, and N. Ford, *Head First Software Architecture.* in Head first series. O'Reilly Media, 2024. [Online]. Available: https://books.google.at/books?id=Whj9EAAAQBAJ

[39] V. Seniuk, “Data Transfer Object Pattern in Java - Implementation and Mapping.” Accessed: Dec. 07, 2024. [Online]. Available: https://stackabuse.com/data-transfer-object-pattern-in-java-implementation-and-mapping/

[40] G. Morling, A. Gudian, S. Derksen, F. Hrisafov, and the MapStruct community, “Data Transfer Object Pattern in Java - Implementation and Mapping.” Accessed: Dec. 08, 2024. [Online]. Available: https://mapstruct.org/documentation/stable/reference/html/