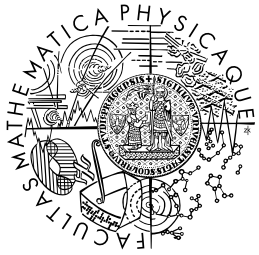

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Denis Drobný

Extracting Information from Database Modeling Tools

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	RNDr. Pavel Parízek, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2019

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the “Metodický pokyn o etické přípravě vysokoškolských závěrečných prací”.

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the ”Copyright Act”), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs (”software”), I hereby grant the so-called MIT License. The MIT License represents a license to use the software free of charge. I grant this license to every person interested in using the software. Each person is entitled to obtain a copy of the software (including the related documentation) without any limitation, and may, without limitation, use, copy, modify, merge, publish, distribute, sublicense and / or sell copies of the software, and allow any person to whom the software is further provided to exercise the aforementioned rights. Ways of using the software or the extent of this use are not limited in any way.

The person interested in using the software is obliged to attach the text of the license terms as follows:

Copyright (c) <year> <copyright holders>
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sub-license, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

In Prague date ..... signature of the author

Dedication.

Title: Extracting Information from Database Modeling Tools

Author: Denis Drobný

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Data lineage is a way of showing how information flows through complicated software systems. If the given system is a database, tables and columns are visualized along with SQL transformations of the stored data. However, this picture may be difficult to understand for people with weaker technical background, as database objects usually obey naming conventions and do not necessarily represent something tangible. To improve lineage comprehension, we developed a software called Metadata Extractor that on one hand brings the further description of the database objects, as well as introduces a whole new perspective on data in a system through business lineage aimed for non-technical users. The additional metadata enriching data lineage is extracted from data modeling tools, such as ER/Studio and PowerDesigner, that are widely used in the database design process. The solution extends the Manta Flow lineage tool while taking advantage of its features at the same time.

Keywords: Data Lineage Data Modeling Database Architecture Metadata

Název práce: Extrakce informací z modelovacích nástrojů pro databáze

Autor: Denis Drobný

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Pavel Parízek, Ph.D., katedra

Abstrakt: Data lineage je zobrazením toku informácií v komplikovaných systémoch. V prípade, že takýmto systémom je databáza, ukazuje tabuľky a stĺpce, spolu s SQL operáciami, ktoré modifikujú obsah dáta v nej uložené. Takýto pohľad však môže byť ťažko pochopiteľný pre ľudí, ktorí sú menej zbehlí v technickom prostredí, keďže databázové objekty väčšinou podliehajú menným konvenciám a častokrát reprezentujú implementačné detaily. Pre jednoduchšie pochopenie dátových tokov sme vypracovali softvér Metadata Extractor. Ten na jednej strane získava podrobný popis databázových objektov a zároveň prináša novú perspektívu na tok dát pomocou business lineage, ktorá je pochopiteľná pre širší okruh ľudí. Informácie potrebné na vytvorenie takejto funkcionality získavame z modelovacích nástrojov, akými sú napríklad ER/Studio a PowerDesigner, ktoré sa využívajú pri navrhovaní databáz. Naše riešenie využíva a zároveň rozširuje Manta Flow, čo je softérový nástroj na výstavbu dátových tokov. ňň

Klíčová slova: Dátové toky Dátové modelovanie Databázová architektúra Metadáta

Contents

1. Introduction

There is no business today that can live without being backed by a database to store and query its data. No matter what domain an enterprise is focused on, we can enumerate many reasons why database storage helps a company to be more effective and why its deployment is a good idea. Here we provide some examples of how databases are used through various business domains:

- Social Media
  Every piece of information that has ever been published on social media, from photo through a reaction or comment to friendship establishment, was stored somewhere and that place is a database. Usually, the database that a social platform uses does its job in the background. Nevertheless, there may occur events when the data storage reminds of its presence as it did on the most recent outage of Facebook [1].

- Healthcare
  Easy accessibility of a large amount of patient's data is the main reason to deploy a database in the doctor's office or within a healthcare organization [2]. High discretion is a requirement when managing data of such sensitiveness.

- Finances
  Databases take care of our money and transactions as well. The standards for coping with such a huge amount of critically important data are set high, thus the processes related to, say ATM withdrawal, must be complex to guarantee reliability [3].

- E-commerce
  Every company that sells products online should use a database. The bare minimum is to store offered products themselves and keeping track of purchases that were done by users.

Once the decision to use a database is made and its usefulness for the business is proved, there may be still a long way until everything runs as expected and the company can make use of all the advantages that data storage brings.

The database design phase comes in place then. By the nature of the problem, a top-down approach to the process is usually followed. At the start there is an enterprise that knows what real life aspects need to be captured in a database. To convert this idea into a working solution, the company typically hires a database designer.

Then follows a discussion between an expert in the business domain where the enterprise operates and a database professional, in order to identify and collect requirements for the future system. At that moment, data modeling comes into play. Instead of a transcript of the conversation, a better solution is to translate the debate into a more intuitive and standardized piece of documentation, which means into a conceptual data model. Once the initial model is created, the next steps are going more and more toward the final implementation of the database.

After that, a database designer works on the development of a logical model and the most high-level concepts are transformed into the ones that are combining high-level perspective with more technical aspects, but the description of them remains independent of a concrete database.

Lastly, the organization of the database is pointed out and captured in a physical model of the analyzed system. From this point, solid documentation is available and it is straightforward to finally deploy a database that is described in the low-level model as it has a one-to-one mapping with the implementation itself.

The process of development and deployment of a database consists of multiple stages, as we have seen. In the beginning, there is a high-level view of why the database is needed and what purpose it will serve. Hopefully, in some time the result is that the data described in the initial step are stored physically at some server. This way the data can be accessed and processed.

But that is just the beginning. The importance of a database for an enterprise is not in how it is designed. What does really matter is that big companies have plenty of business processes managing contents of storage via scripts in an automated way.

For example, travel companies offering airplane tickets tend to increase price when there are not many spaces left for a trip. Thus, when a customer buys a ticket, there is a business logic that computes how the price of the remaining tickets should be raised and updates the records in the database representing the not-yet-sold tickets accordingly, so that valid information is shown to customers. The logic takes place thanks to SQL queries applied to a database. As the amount of similar business processes grows, the ability to justify the correctness of data decreases. Also once an error in data is found in such a big ecosystem, it may be very unpleasant to trace it as data are affected by a possibly huge number of sources and transformations hidden in scripts.

Data lineage is the answer to the struggles with being overwhelmed by the complexity of a big data solution. It brings ease to seeing what and how is affecting data stored in databases.

The lineage of data shows database tables and transformations used for either writing or reading data from tables. It is really helpful, however not for everyone. We outlined that there are multiple perspectives on a database through data models, and every perspective has a different audience. For example, the conceptual is for business people while the physical one is read by database engineers. The problem with the existing data lineage solutions is that they only display the level of abstraction understandable by database professionals. At the same time, people with a less profound technical background that would want to make decisions based on how data flow in their systems are not having an easy time trying to figure out what is going on in such data lineage. That is why we want to bring the business data lineage, which is speaking the language of more enterprise people coping with data and making decisions related to them on a daily basis. We assume big companies approach database development responsibly, thus there exists documentation of their systems in the form of

data models. We reuse the models for creating a business perspective on the movement of data in a system. Data models also store valuable metadata that can make data lineage, even the technical one, more readable and transparent. Even though the business lineage provides a summarized and simplified view of data flow, it has to be well aligned with physical flow of data, so the high-level view does not drift away from the low-level situation.

Let us demonstrate the importance of data lineage on the regulation that every company storing personal information about citizens of European Union faces - General Data Protection Regulation (known as GDPR). In order to comply with the regulation, a company must have a precise knowledge of what it does with data of its customers. For example, GDPR enforces the Right of access [4], meaning any customer can access all data related to him the company stores upon request. With the help of data lineage, it is only needed to identify what are the entry points for information about users. Then the map of data lineage does the rest and highlights where the data end as a consequence. To serve the user's request, the enterprise would just collect data from the identified sources without having to do an exhaustive and error-prone analysis of internal processes. Surely, GDPR is a complex set of rules like this, but data lineage can help greatly with many aspects of the regulation. Although data lineage does not make a company automatically GDPR-compliant, it is a shortcut to get there.

To improve decision making related to data in enterprise-wide systems, we developed the software which we are going to discuss throughout this work.

1.1 Goals

The overall goal was to create a piece of software, called Metadata Extractor, for augmenting data lineage by information obtained from data models. The software must support the extraction of metadata from two modeling tools - PowerDesigner and ER/Studio. Another requirement was to be able to construct business lineage based on the extracted information. We integrate Metadata Extractor into the ecosystem of Manta Flow, a software tool for creating and visualizing data lineage.

- Develop a component that extracts metadata from data models that were created using SAP PowerDesigner.
- Develop a component that extracts metadata from data models that were created using ER/Studio.
- Build business data lineage based on cooperation with the existing solution that is able to create technical lineage.
- Provide a description for the general scenario of metadata extraction from a data modeling tool. Describe how the acquired information should be passed to Manta Flow, making Metadata Extractor easy to extend towards new modeling tools in the future.

1.2 Glossary

Let us introduce some crucial terms used throughout the whole text.

A *database* is a collection of related data. By data, we mean known facts that can be computerized and that have the implicit meaning [5]. We will consider that a database stores data relevant to an enterprise at a host that can be accessed via network.

A *data model* is a description of data, data relationships, data semantics, and consistency constraints.

A *database schema* defines how is the database described in a data model actually constructed, specifying types of fields from the data model. It represents an instance of a data model.

A *diagram* is a graphical visualization of a data model.

A *data modeling tool* is a software that allows a database designer to create data models. End users may use the tools for interactive previewing of the models' diagrams.

Data lineage provides a picture of a data movement in some system across its components. It is a description of how data go from an origin through their transformations until they reach a destination. The ability to see graphically how data are used, what for, and what are the consequences of the usage in a system is a powerful tool for error tracing.

1.3 Outline

In Chapter 2, On Database Background, we go through the concepts fundamental for understanding the domain to which Metadata Extractor contributes. The prerequisites are to understand databases, data modeling, and data lineage.

Chapter 3, Modeling Tools, describes more specifically what are the programs used for creating data models capable of doing. Also, we discuss the specific tools, with which Metadata Extractor works, in more detail.

Chapter 4, Analysis & Design of the Solution, is concerned about analyzing the aspects of the two selected modeling tools - PowerDesigner and ER/Studio. Based on the analysis, we identify requirements for our software and propose a high-level architecture of Metadata Extractor. In the chapter, we also preview programs solving similar problems as we do.

Lastly, Chapter 5, Implementation, is discussing in detail the classes of which is Metadata Extractor made.

2. On Database Background

In this chapter, we will introduce terms and concepts that help us understand the domain to which Metadata Extractor contributes. To understand requirements, design and the need for our software, a user must have some level of knowledge about databases and data lineage beforehand.

2.1 Databases

A standalone database is not very useful, as it is only some physical storage that never changes. To take the full advantage of it, users need some means to define, create, maintain and control access to the database. That is the purpose of software called *database management system (DBMS)*.

We already described why someone may want to use a database and roughly mentioned what are the pieces of data that may be saved there. Now let's take a look at what are the differences between concrete database implementations.

The basic classification of database types is simple and binary - they are either relational or non-relational. There are database management systems built around both kinds providing the necessary functionality.

Here we define important database-related terminology.

Relational Databases

A *relational database* is a set of tables. A table consists of rows (also called records) and columns. We can see such a table as a class whose characteristics are represented by columns and instances by rows. The important aspect is that relational tables carry both data saved by a user and relationships between the stored data as well. When inserting an atomic piece of data about a record/instance, the corresponding column is filled with a value. Whereas to capture a relationship between objects the concept of keys is used.

A *key* is a subset of table's columns used for identifying a record.
A *primary key* is a key that non-ambiguously identifies a record in a table and is used when referring to the specific record.
A *foreign key* is a key uniquely identifying a record from a table (the referenced record may be originating in the same table or coming from a different one).

Relational databases are known also as SQL databases by the language - *structured query language (SQL)* which is used in RDBMS for managing data.

The most widely used relational database management systems are Oracle, MySQL, Microsoft SQL Server, PostgreSQL, IBM Db2 in this specific order. [1]

In this work, we focus only on databases that are of the relational kind, even though there also exist NoSQL databases that do not follow the relational paradigm.

[1] The database technologies usage statistics are based on data from the most up to date version of the website db-engines.com [6].

The main reason behind this is the fact that NoSQL databases became popular just recently and modeling of these databases is a quite new discipline. At the same time, there is a variety of approaches to NoSQL modeling and none of them has established as a standard yet. Also, data models bringing a greater level of abstraction are not commonly used in this field.

Another fact to consider is that once a database is of the relational kind, it is more or less known what to expect from it. The hierarchy of objects in a RDBMS has a similar skeleton every time. That means a tool that would extract metadata from relational data models is potentially more powerful, as it can be applied to more database technology types. An equivalent tool aimed for some non-relational data model would be effective for a narrower set of technologies as No-SQL databases are classified further by the way they store data and the data models must reflect it. For example, once Metadata Extractor can load PowerDesigner models, it directly supports Oracle, IBM DB2, etc. While if it was working with NoSQL data models it would not be that straightforward. In this situation, further questions must be asked, such as if a given data model reflects, let's say, a key-value store or a graph database and approach them individually. It is because the way data are organized in the two types of NoSQL databases is substantially different, therefore data models corresponding to them must vary [7].

Lastly, despite that non-relational databases are be growing in numbers and becoming a serious alternative to RDMBS, the relational ones still are, and in the near future will be, far more widely used.

Means of Database Access

Databases can be managed directly through database management systems by a user which is using a query language for accessing data. However, third party (or application) programs also need to access a DBMS. The way applications use databases is through an application programming interface (API) of a DBMS which provides a set of methods to an application. When such API is called, it usually translates the request so that a specific target DBMS driver understands it and performs the desired action.

A *Connection String* is textual information used to identify a data source and establish a connection with it. It is made of pairs of keys and values separated by a semicolon, where the keywords are parameters of a specific database connection. The most widely used APIs to DBMS are ODBC, JDBC and ADO.NET.

2.2 Database Modeling

Modeling is a crucial phase of the database design process. Developing a database is just like building a house. Everyone will agree that no construction work can go without a solid design and documentation. It would sound a bit strange to hire construction workers straight ahead and tell them we need a house that has 5 rooms and some toilets and expect a good result. Most probably, some kind of building would be produced at the end, but it is clear that the expectations and requirements of the later inhabitant cannot be met properly this way. Surely

there are good reasons why the usual steps are followed strictly. Let us move on from this analogy back to the database domain.
When deploying a database from scratch one may think of two short term advantages. Firstly, the time interval between the initial thought until having data stored somewhere would be much shorter, and secondly, the initial cost of the system may be lower.
But over time both of these advantages will, most likely, if the database is not ridiculously small, get outnumbered by problems, that will begin to appear. Maintenance of a poorly designed system (or not designed at all) is difficult and leads to numerous outages.

There are good reasons why modeling has its place in the database development process:

- Higher quality.
  Modeling enables stakeholders to get a thorough definition of the modeled problem. Once is known what to solve and what is the scope, it is much easier to come with different solutions and to justify which of the proposed approaches is the most suitable one.

- Costs reduction.
  Errors are identified in the early stages when they are easy to fix.

- Better documentation.
  Data models form a nice piece of documentation and they are understandable by each of the involved stakeholders. When someone wants to comprehend a system, he can choose a data model on an appropriate level of abstraction that will introduce him to the important aspects of the problem that suits his knowledge and qualification.

- Correctness.
  Tracking whether high-level concepts were implemented and represented correctly in the end is made straightforward.

- Deeper understanding.
  During the design process, developers may learn a lot about the characteristics of the data that will be stored. This information is crucial for choosing an appropriate type of database - whether to stick with a relational database, and if so, which DBMS is the right one, or to better look for a non-relational database option.

2.2.1 Data Model Perspectives

In this section, we discuss data models from two basic perspectives. The vertical classification talks about what levels of abstraction can data models be designed on. While when looking on data model types horizontally, we deal with specific types of data organization that are used on each of the vertical levels.

Vertical Classification

We will go through three important hierarchies, where each of them consists of multiple views on a database, to see how variously databases can be perceived.

American National Standards Institute came with a database structure called three-schema architecture [8]. It is formed by:

- External Level
  Database as a user sees it.
- Conceptual Level
  Point of view of the enterprise that the database belongs to.
- Physical Level
  The actual implementation.

The idea behind the structure was to create three different views that are independent of each other. For example, a change of the implementation, which is tied with the physical level, would not affect any of the remaining levels if the structures remained the same. The important aspect is that this structure is used to describe a finished product, it does not say anything about the design process leading to the product and should not be mistaken with the differentiation of data models that will be introduced later.

On the other hand, to standardize the process of designing a database, Peter Chen identified four levels of view of data [9], where each of the levels has its important place:

1. Information concerning entities and relationships which exist in human minds.
2. Information structure - organization of information in which entities and relationships are represented by data.
3. Access-path-independent[2] data structure - the data structures which are not involved with search schemes, indexing schemes, etc.
4. Access-path-dependent data structure.

The categorization of data models has undergone some modifications. For example, the first level is omitted today. The classification takes into account who is the audience that will work with a data model, whether it is someone who knows everything about databases or it is a business person without a technical background. The levels of abstraction used today [11] are the following:

- Conceptual Data Models (high-level)
  Reproduces real-world objects along with their relationships. Should be close to how business end-users perceive them.

[2] An *access path* is a description how records stored in a database are retrieved by database management system[10]. The important part for us is that the path is DBMS technology specific.

- Logical Data Models (implementation, representational)
  In the middle, between the two other model types, there are representational data models, which are comprehensible by end-users and at the same time are not too abstract. Therefore, that they can be used as a starting point for an actual database implementation of the modeled data.

- Physical Level Data Models(low-level)
  In contrast to the conceptual models, the physical ones are tied with how data are stored physically at a storage medium, showing all the specific internal details which may be overwhelming unless the reader is a computer specialist.

Horizontal Classification

We introduce three specific types of data organization that are widely used nowadays - relational, entity-relationship and enhanced-entity-relationship data models.

Relational Data Model

A relational database is a direct implementation of a relational data model. We already went through terminology used when referring to SQL databases. All the terms such as a table, column, entity, record, and keys originate in the definition of relational data model [12].

Entity-Relationship Data Model

The *entity-relationship (ER) data model* was the direct answer to the four-level architecture[9] that covers the highest two levels and may be a basis for a unified view of data.
It was an opposition to the three major data models that were used - relational, network and entity set model. Its aim was to bring a data model that would reflect real-world objects and relations between them naturally while having advantages of all the three already existing models. The mission seems to be successful as years have proven the ER data model to be the most suitable one for conceptual data modeling. Moreover, ER data models are used most commonly in logical data modeling as well.

Enhanced-Entity-Relationship Data Model

An extended version of ER data model was introduced later - *enhanced-entity-relationship (EER) data model*. The main change is that the concept of sub-classes and super-classes, known as inheritance or is-a relationship, between entities was added.

Summary

To put the two perspectives together - conceptual and logical data models are usually represented by ER data models. The low-level model type is tied

directly with how a database is organized, therefore physical models must obey the structure of a database. Given that we work with relational databases, we assume physical data models to be of the relational kind.

Now, we describe the standard three levels of data models in more detail, starting from the most abstract one.

2.2.2 Conceptual Data Model

The purpose of a conceptual data model is to project to the model real-world and business concepts or objects.

The main characteristics include:

- It is aimed to be readable and understandable by everyone.
- It is completely independent of technicalities like software used to manage the data, DBMS, data types, etc.
- It is not normalized.

Object Types

- A real-world object is captured by a *entity*.
- *Attributes* are used for further description of entities. Each attribute should represent an important aspect of an entity. [3]
- *Relationships* between entities are necessary to provide a complete view of the section of the world that a data model resembles.

We illustrate the concepts used in conceptual models on an example supported by Figure 2.1. If our modeling domain is education, then an entity may be a teacher or a lesson. A salary would be information to be stored when describing the teacher, making it an attribute. Having lectures captured in our data model, it is really fundamental to see what lesson is taught by which teacher - that would be done using relationships.

[3] Definitions varies and in some literature, it can be even found that conceptual entities lack attributes. We assume, that these entities may contain important attributes as it is more common interpretation and modeling tools support attributes on the conceptual layer as well.

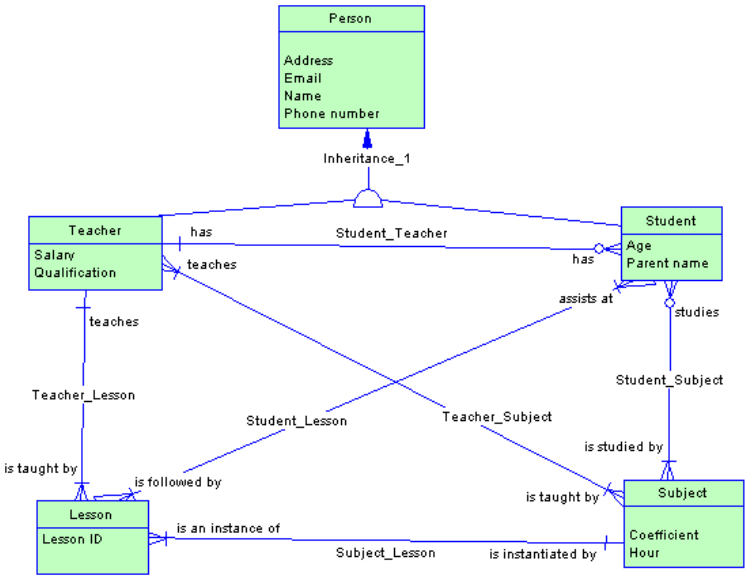

Figure 2.1: The figure shows a diagram of a conceptual data model realized by the EER data model. There are five entities, each of them has at least one attributes. Teacher and student entities inherit from the person entity. Blue lines represent has-a relationships between entities.

2.2.3 Logical Data Model

Keeping its structure generic, a logical data model extends a conceptual data model as they capture more details, making it not that easy to read, but becoming good base documentation for implementation. At this level, data requirements are described from the business point of view.

The main characteristics include:

- It is independent of the software used to manage the data or DBMS.
- Each entity has a primary key.
- Foreign keys are expressed in the model.
- Data types description is introduced (but in a way that is not tied with any specific technology).
- It is normalized up to the third normal form.

Object Types

- *Entities*, in contrast to conceptual entities, do not represent solely real-world objects, in Figure 2.2 we can see entities were created by a deconstruction of many to many relationships.
- *Attributes* do not necessarily capture only the important features of objects anymore. They are used also for storing keys of an entity.

- *Relationships* are not that abstract as on the upper layer. Keys that actually define relationships must be added to entities as attributes.

An example of a logical data model can be seen in Figure 2.2. It describes the same system as Figure 2.1.

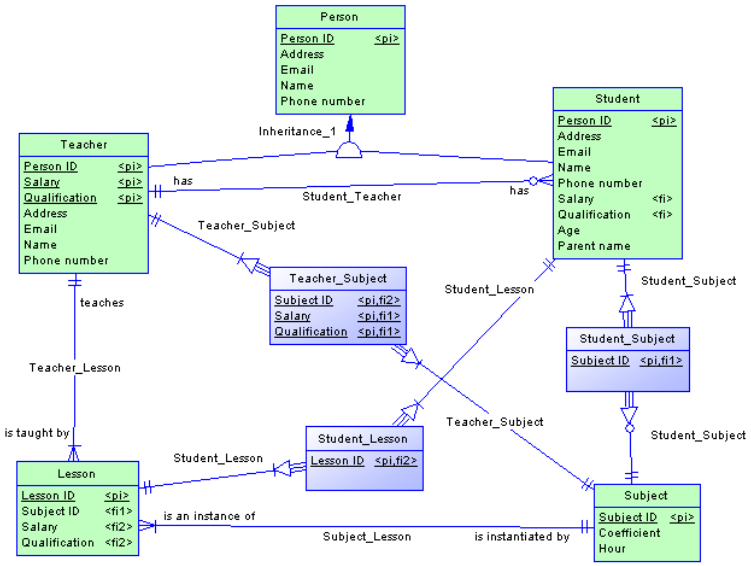

Figure 2.2: The figure shows a diagram of a logical data model. The main difference in contrast to Figure 2.1 is that many-to-many relationships are deconstructed to entities, and primary with foreign keys are defined for entities.

2.2.4 Physical Data Model

A physical data model is a description of a database implementation, so it is tied with one specific database technology. It necessary because the models needs to have one-to-one mapping with their actual implementation. Main message of physical models is to communicate how data are stored in detail.

Objects in physical models should reflect the database organization, and at the same moment related higher-level concepts should be transformable to the physical level.

The main characteristics include:

- Exact data types (DBMS specific) and default values of columns are outlined in the model.
- Naming conventions are applied on objects.
- Constraints are defined (eg. not null, keys, or unique for columns).
- The model contains validation rules, database triggers, indexes, stored procedures, domains, and access constraints.
- Normalization in order to avoid data redundancy (or de-normalization for performance improvement) is reflected in the model.

Object Types

- *Tables* should store records that correspond to logical entities.
- A *schema* is basically a container for tables that logically groups them. Database users have usually schemas assigned to them and can access only the tables contained in that schemas[4]. Not every DBMS supports this concept, though.
- *Columns* should represent logical attributes in memory.

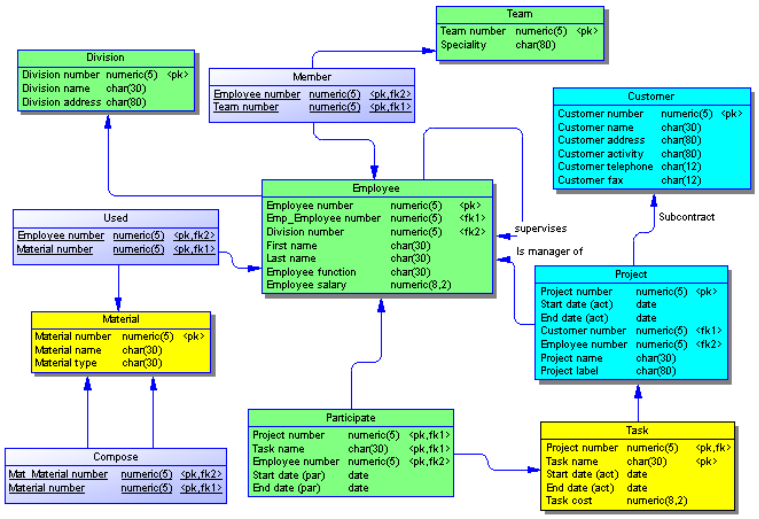

Figure 2.3: The figure contains a diagram of a physical data model with eleven entities. We can identify keys and specific data types of columns. Relations are pictured based on key columns.

2.2.5 Relations Between the Models

We described what the role of each of the layers in a database design process is. Here we show that the data models are somehow connected vertically - across different abstraction levels. Let's see why are such connections extremely useful.

When talking about vertical connections, it is good to think about how database design may proceed. The basic approach is the *top-down approach* to database modeling. It is quite natural to start with a general idea of what information should a database store, and what are relations between the stored objects. An end-user defines this high-level logic. As time goes, the importance of a database designer grows, until he is at full charge and develops a complete database. It is the most common scenario of database development - a client identifies high-level needs for a database and hires an expert in this domain to make it happen.

The other way to create a full view of a database is the *bottom-up approach*. It can be more challenging to imagine what would be possible use-cases for this approach, but there are some problems that are bottom-up in nature. A nice

[4]Plural of the word schema is schemata but in the literature about database design, the word schemas is used

real-world example of the bottom-up strategy is how physicians work. They start with "low-level" details such as symptoms and they are trying to build the whole image of patient's condition. In the field of databases, data elements are firstly identified first and after that, they are logically grouped to form bigger units - entities. And so on until the full hierarchy is known.

Maps-to Relation

In order to see how concepts captured in data models are getting transformed across levels of abstraction, a relation that we will call *maps-to* is used. The relation connects vertically objects that are semantically equivalent in different data model types. Sometimes, even mapping between objects on the same abstraction layer is allowed, but we will not consider this, as it mixes two different concepts together - database design with data lineage. To be more precise, what we mean by the semantically equivalent objects. We assume maps-to edges leading solely between these types of modeled objects:

- A data model can be mapped to another data model.
- An entity can be mapped to another entity or a table.
- An column can be mapped to another column or an attribute.

Following mapping links is extremely useful when a person wants to gain an overview of the system and tries to comprehend it fully. For example, when a user sees a data table in a physical model that (i) has a technical name that obeys some naming convention and (ii) due to normalization does not represent any object straightforwardly, he can follow mapping links that lead to higher layers. We know a higher level provides greater abstraction over the implementation and uncovers motivation why the table was created. After doing this, the purpose of the inspected table in a database should be much clearer. It is worth mentioning that usually, mappings between objects are not one-to-one relationships, but the cardinalities may vary greatly. One logical attribute may be commonly realized via multiple database columns. Normally, more technical models are composed of a bigger count of objects so one conceptual entity may be realized by multiple database tables in the end. Generally, it is assumed that the number of conceptual objects is smaller than the number of logical objects, which is at the same time smaller than the number of physical objects. It is natural - when capturing important high-level aims fewer entities are needed to express the intention, but as we are getting closer to the implementation, more necessary details come into play. Lastly, we consider the mapping relation to be symmetrical.

2.3 Data Lineage

Data lineage brings a way of tracking data from its origin throughout its whole life cycle, taking into account every process that manipulates the data until it reaches its final destination. It is like telling the story of a piece of data, including where does it come from, what transformation it undergoes, and how it interacts with other data. It should provide answers to questions such as where the data

in a given solution come from, whether it can be trusted or not, how it gets from point A to point B, and how the data changes over time in the analyzed system. Basically, data lineage helps enterprises to gain deeper knowledge and understanding of what happens to data as it travels through various interconnected data pipelines[5] that their systems consist of. Despite the fact we are focused specifically on databases, data lineage is a general concept, where data origins and targets are not necessarily databases. The data may come from, let's say, a user interface, where it is inserted directly by a customer, and end in an output of a reporting software. This overview of a system that data lineage provides is crucial when making decisions about infrastructure. With help of data lineage understanding the consequences of the actions should be more clear. Also, the lineage makes much easier to find errors in systems since the bugs can be tracked down from where the undesired behavior came to the surface to where the suspicious data originate. Certainly, somewhere between these two points the malfunctioning part is, and, thanks to data lineage, the set of suspicious operations is reduced and visible. Therefore, much time spent on solving issues should be saved.

A visual representation is most commonly used to present data lineage. Generally, we can think of visualization as a graph. An example of a data lineage graph is demonstrated in Figure 2.4.

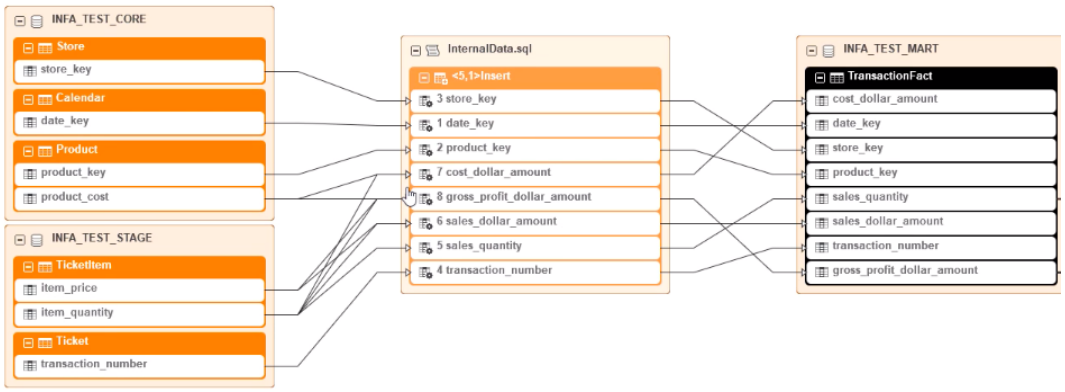

Figure 2.4: Data lineage as it is visualized in Manta Flow user interface. The simple example illustrates how the TransactionFact database table is filled with data originating in five different tables via InternalData.sql transformation that inserts the data from sources to the target. The directed edges are leading from a data origin to destination.[13]

Having a reference point of interest, we can divide data lineage into three types by what information it captures. *Forward data lineage* inspects the movement of data towards the destination, *backward data lineage* creates a picture of what happened to data when traveling to the point from the source. The last type, *end-to-end data lineage*, combines both approaches and shows the full flow of data from its source until the very end destination.

Another classification of data lineage is between business and technical lineage. *Business data lineage* highlights only transformations and aggregation of data in a

[5]A pipeline is a set of elements manipulating and processing data where the output of one element is the input of another.

simplified way to the target business user, whereas *technical data lineage* displays precisely the flow of physical data as it is happening in underlying components (e.g., applications) the system is made of. A comparison of the two data lineage perspectives can be seen in Figure 2.5.

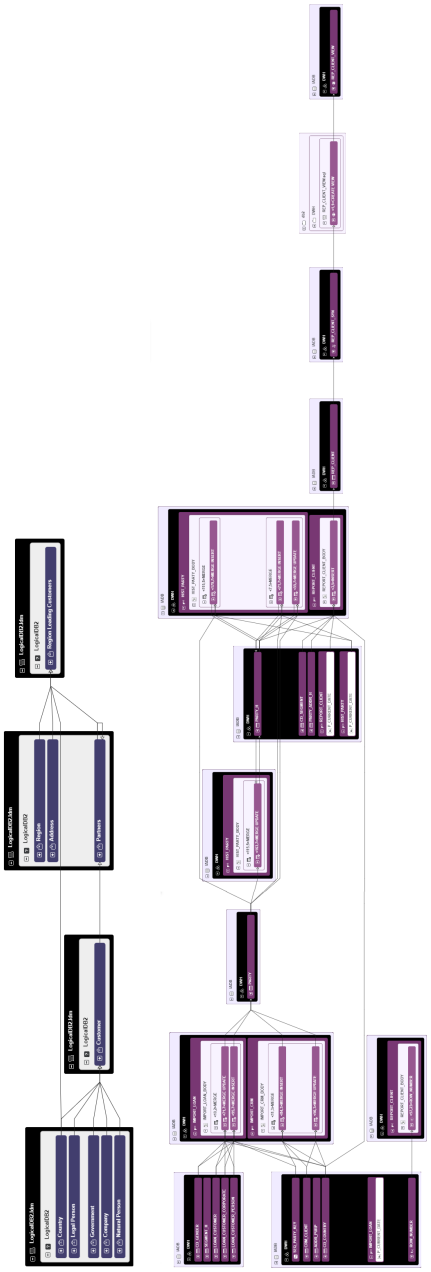

Figure 2.5: The two visualization of data lineage show the very same database system from two different perspectives. The business lineage (blue) hides transformations and speaks natural language, using concepts that appear in real world providing cleat motivation of the business process. Whereas technical data lineage (purple) precisely illustrates what happens with data in a database management system. Each of the business objects is mapped to its physical counterpart and the high-level flow fully corresponds to the low-level one. For example, table PARTY is equivalent to entity Customer. The lineage was created by Metadata Extractor in cooperation with Manta Flow.

Now we will focus on how data lineage can be computed to describe the lifespan of data that are coming from or being saved to an SQL database. To analyze the flow of actual data, having access to quality metadata is fundamentally needed. *Metadata* is the data describing other data. The metadata we use when analyzing a database includes database name, names of tables, columns in tables, names of columns, procedures, data types, etc. When we have this information describing all the records that can be stored in a database, together with all SQL scripts (or transformations generally) which are used for management of the database, we can reliably determine how the data flow as the database is being used.

The process of data lineage construction is as follows. The first precondition is to have access to all metadata related to the database under analysis, in order to have a clear picture of objects stored there. Then transformations that modify data are examined. Either SQL or ETL (extract, transform load) procedures may be applied on data. A node representation is added for each of the transformations. Then data lineage creation tools, like Manta Flow, identify what tables and columns are the sources of input data for procedures and where outputs of given operations are stored. Each input and output is represented by a graph node as well. Based on an analysis like this directed edges between the nodes we described are added to show dependencies. Inputs are connected procedures in such a manner that every edge originates in of the input nodes and ends in the transformation node. Correspondingly, edges from query node to output nodes are made.

2.3.1 Manta Flow

Manta Flow is a product of Czech startup company MANTA (see `www.getmanta.com`). It is a tool that automatizes data lineage creation by analysis of programming code. It is able to cope with SQL, altogether with various of its sub-dialects, ETL and reporting tools, and other programming languages like Java. The uniqueness of the software product is in its capability of handling code that is hardly readable by a human. Thanks to this feature, Manta Flow can automatically process databases consisting of millions of records and create a map of data flow across the business intelligence environment - data lineage. Alternatively, the data flow is not visualized directly by Manta but cooperates with third-party data governance solutions like Informatica, TopQuadrant, Collibra, and IBM IGC.

Our aim to interconnect Metadata Extractor with Manta Flow to enrich the data lineage it produces by metadata that can be obtained from relevant data models and thus bring a better understanding of the system under analysis. Secondly, Metadata Extractor should automate business lineage creation using the information extracted from modeling tools.

Supported Database Technologies

Among other technologies, Manta Flow is able to scan, these are the currently supported relational database types it can handle. That means when physical models are aimed at one of the following database systems, we can create corresponding business lineage. Metadata Extractor is, naturally, effective on the

same DBMS as Manta Flow. Specifically:

- Oracle Database
- Microsoft SQL Server
- SAP ASE (Sybase)
- Hive
- IBM Netezza
- IBM DB2
- PostgreSQL
- Amazon Redshift
- Greenplum
- Teradata

2.3.2 Data Lineage in Modeling Tools

It is quite common that modeling tools provide some kind of view of how data flow in the modeled diagrams or they have data movement models where objects from data models take part. However, this is not the way we will determine the desired business data lineage. The reason why not to take into account this feature is that it may be completely away from how a system really works and how data move in it. This is because none of the modeling tools inspects live databases and scripts working with them, so the only way how a data lineage can be created in these tools is that a user draws a lineage graph by hand. It may be useful at the time when a database is not yet implemented and there is this type of dependency relationship that cannot be captured other way. But once a database is running, this lineage may be misleading, as there is no way to enforce the correctness of the data flows specified manually. That is why Manta Flow brings data lineage that corresponds to how a database is deployed and used in reality. Then, thanks to mapping relations, Metadata Extractor can propagate the lineage to more abstract concepts on the conceptual and logical level.

3. Modeling Tools

The main feature of modeling tools is to capture metadata about data models that can be created using them and previewed. The tools use diagrams to present data models to their users.

3.1 Construction of a Data Model

Now we will take a look at how someone developing a database can actually create those models. In fact, a data model could be created by hand using only paper and pen. It would definitely bring some of the benefits described above, but to take the full advantage of modeling, we assume using *computer-aided software engineering (CASE) tools*. The tools are here to help with the development of quality software [14]. Here is an overview of different ways how a data model can be created using them.

3.1.1 Modeling

By modeling, we mean creating a data model via a user interface of CASE tools from scratch dragging and dropping data model objects around. This way of creation is the most similar to the pen and paper method. A user builds a model manually by selecting what object should be created and bringing it to the particular model, then he provides details about the object, creates sub-objects or specifies relationships with different objects. Some tools do not allow creating an arbitrary model, but only the conceptual or logical models may be drawn like this. The reason behind not allowing user to create a physical data model out of scratch is that a physical model should either (i) be the result of a design process and be based on a model with higher level of abstraction (see Section 3.1.3) and then adjusted, or (ii) should resemble a live database that can be transformed into the corresponding data model by reverse-engineering (see Section 3.1.2).

3.1.2 Reverse Engineering

Reverse engineering, or alternatively back engineering, is the process which aims to find out principles of how things are done or work in a system that is already running and try to gain a deeper understanding of the system. Applied to our domain, the reverse engineering approach to the creation of a data model means that a CASE tool connects to a database and brings every object found to the physical model that is created. A database management system usually can provide additional metadata on objects, for example about primary/foreign keys, thanks to what relationships between tables. If the modeling tool is smart enough to make use of it, can be brought by the reverse-engineering as well. Every model acquired by this process is an exact image of a database and a one-to-one mapping between the model and database should be secured.

3.1.3 Generating

From a data model, new models on different levels of abstraction can be derived from it. Modeling tools usually support translating objects to semantically equivalent ones towards either greater or smaller abstraction. Of course, models created like this are not full-featured but they are definitely a better starting point for database designers to take over. For example, when a conceptual data model is arranged and a logical model should be created based on it, it is really helpful not to start from scratch. Instead, an outline of the logical model can be created by generation from a corresponding conceptual data model. Then the result can be reshaped into the desired condition more quickly. Generation sources and targets are in maps-to relation implicitly.

3.1.4 Importing

Finally, a CASE modeling tool may be able to import data models that were created using different modeling software and recreate the data models.

3.2 ER/Studio Data Architect

ER/Studio Data Architect is a data modeling and database architecture tool by IDERA, Inc. The latest version is 18.0 [15].

The tool is focused on building a business-driven data architecture providing an understandable interface for business users. It also improves data architecture standards by reducing redundancies, enforcing data consistency and quality. The tool also tries to provide a framework for visualizing data flows by data lineage diagrams.

ER/Studio supports creating logical and physical data models. The feature of forward and reverse-engineering may be applied on them. Tens of database platforms can be targeted by a physical model.

Logical models implemented by entity-relationship data models, whereas physical models are of the relational type.

The extension of the product, ER/Studio Data Architect Professional, comes with a model repository that makes collaborative development of data models easier.

3.3 PowerDesigner

PowerDesigner is a software for data modeling owned by company SAP SE. The tool is well established and has been used by enterprises for 30 years, the current version is 16.6 [16].

PowerDesigner provides a range of various modeling techniques such as application UML modeling, business process description, enterprise architecture, data movements models. But what is most important for us, data modeling where conceptual, logical and physical data models are supported and the low-level models are compliant with more than 60 database management systems. Forward and reverse-engineering of the models can be applied. The first two

levels are of extended-entity-relationship data models and the physical one is of relational data model [17].

The CASE tool allows sharing metadata across all the supported model types and disposes of enterprise repository solution, which makes cooperative modeling by multiple users easy, and has a version controlling ability.

4. Analysis & Design of the Solution

The purpose of Metadata Extractor is to obtain metadata from data models that were created using modeling tools ER/Studio and PowerDesigner. The solution needs to be able to interact with Manta Flow and bring business lineage on objects from physical and logical data models. Currently, Manta Flow supports only automated creation of the technical lineage.

Here we describe how we proceeded when analyzing the solution and discuss crucial features of ER/Studio and PowerDesigner in detail. Therefore, we can identify the important aspects of the developed software that we needed to focus on when finally implementing Metadata Extractor.

In this chapter we provide answers to these questions:

1. Identify what data models the modeling tools work with, what objects are contained in the supported data models, how they are organized and what metadata can be obtained that are relevant to be brought into data lineage.
2. Find out how the data models are saved and how the interesting information from 1.) can be reconstructed.
3. Determine how the file format in what data models are stored can be parsed.

4.1 Analysis of the Problem

We already presented that modeling tools are capable of creating data models. These models are saved into files. The output files of the selected modeling tools are going to be the input of Metadata Extractor that uses them for recreation of objects and information contained in the data models. In order to come with the logic of the reconstruction, we needed to identify what objects are saved in the files produced by modeling tools, and how they are represented. In Section 2.2 about database modeling, we introduced the standard layout of every data model type. We will quickly review their main objects.

Conceptual and logical data models consist of entities which may have attributes. Physical data models are made of tables. A table is composed of columns that may belong to a schema.

These are the fundamental metadata objects which Metadata Extractor must load form the files. Consequently, it must correctly load hierarchy of these objects. For example, an attribute must be assigned to a specific entities correctly, etc.

Next, we look at how it is possible to find pairs of objects that are in maps-to relation, and how do they refer to each other across levels of abstraction. In an example, how a logical attribute and a physical column, are tied together even if defined in different data models.

4.1.1 File Format

In this section, we firstly discuss what is the output of the analyzed modeling tools. Then we look at the main principles of how and what information is stored in the given formats. Knowing this, Metadata Extractor can find the required objects (which specifically mentioned in Section 4.1) serialized in a file.

ER/Studio

The ER/Studio modeling tool uses its custom file format. These files have the .DM1 extension. In a single .DM1 file, related data models are stored. Let us call these models a solution. An *ER/Studio solution* is a set of data models, describing a problem on both logical and physical levels (the two layers are only that ER/Studio supports). In such a solution one logical model must be present and 0 to n physical ones supporting the logical model. We can imagine why ER/Studio behaves like this. The motivation may be that once there is a problem (if there is no challenge, no data modeling is needed), it must be described by a logical model. Possibly user has worked out the way to solve it, and that is when physical models are present as well. Note that the actual storage may be distributed and the corresponding databases can be of different technologies, that is why more than one physical model are allowed in a single solution.

.DM1 is a textual file format that is organized into many tables. Such a table is a customized CSV (comma-separated values) structure. Since the file format consists of many of these tables, (i) each table has a unique name that identifies it. Next, (ii) a table contains a CSV row, which defines each column of the table. Lastly, (iii) CSV rows are standing for records that are stored in a table. To put it together, by a single table in .DM1 file we understand its name, column definitions, and records (rows representing data stored in the table). At first sight, it is not clear how can complex objects be stored in the files that we just described. To get the idea behind it, we did some work that we are going to describe in this section.

```
StringUsage
StringUsage_Id,UniqueObject_Id,StringUsageType_Id,String_Id,Row_Time_Stamp
1,1,51,1,0
2,1,52,265,0
3,1,53,339,0
4,1,54,340,0
6,1,58,176,0
7,2,37,8,0
8,3,5,10,0
9,3,3,11,0
10,3,1,188,0
11,3,2,188,0
12,4,5,10,0
13,4,3,13,0
```

Figure 4.1: Example of a simple CSV table from a .DM1 file. It is named StringUsage, is made of five columns and stores thirteen records.

An attentive reader may find the terms, we used for when describing a .DM1 table, familiar. That is because we already saw the terminology when introducing relational databases (see Section 2.1). When going through the tables, we can notice columns that have common names. That resembles primary and foreign

key concepts used in relational databases, where a column is used to reference a record stored a different table. The keys indicate relations between tables.

It looks like this could be the answer to the question we proposed earlier - how complex objects can be stored in the simple tables. Each table stores a simple aspect of an object but thanks to the relations, interconnected tables can be linked together and compose from their data a more complicated object.

To identify crucial relations that Metadata Extractor needs to know, in order to be able to reconstruct composite objects such as entities, attributes, etc., from .DM1 files, we developed a little reverse-engineering utility, which helps us to get an overview of the links between the tables. Further details on how we proceeded when designing the utility can be found in Section 5.1.1.

In order to work with a .DM1 file programmatically, it is needed to load it into suitable data structures that can be processed further. That is what a parser does. Metadata Extractor, as well as the reverse-engineering utility, need such parser. We mentioned the file format is basically a sequence of CSV tables. The question was whether to reach out for an existing CSV parsing solution or to develop a tailor-made one. We took the second option, why and how we did so is described further in Section 5.1.2.

To sum up, the reverse-engineering utility provides an insight into the logic of .DM1 files, so we know what data model objects are stored in the files and how the complex objects are deserialized there. Accordingly, we can adjust Metadata Extractor so it can process the files automatically. The parser prepares an input file, thus our software can reconstruct the objects and their metadata stored in the input.

PowerDesigner

PowerDesigner produces files storing data models with three types of extensions - .pdm, .ldm and .cdm. They stand for physical, logical and conceptual data models respectively.

While ER/Studio groups data models into solutions, every model created in PowerDesigner is saved independently. A set of files that are at the same time opened in PowerDesigner forms its state. Such a state is called a workspace and can be saved into a .sws file. However, the workspace files do not uncover us anything interesting. They only capture what data models were at some time opened in the user's interface and does not tell anything about logical links between the files.

The data model files are XML (Extensible Markup Language) based file formats. Alternatively, PowerDesigner is also able to save its data models into a binary file. The advantage is that the binary files can be processed more quickly and are smaller in size than their XML counterparts. However, we need a Metadata Extractor user to save his models as XMLs that is the only this way currently supported.

Once a user provides a correct input file the software needs to load it. When parsing XML files there are two major ways how to do it.

The first approach is SAX (Simple API for XML), which is an event-driven parser that processes an XML document sequentially by a single pass. By default

its processing is stateless and handlers are triggered when an event occurs. It is a simple and lightweight XML parser.

On the other hand, we have a family of DOM (Document Object Model) parsers. They load an XML file into a full AST (Abstract Syntactic Tree) structure. This way of file processing is more memory and time consuming but translates everything stored in the input file into data structure straightforwardly. Then, it can be worked conveniently with the tree-like resulting structure, where nodes represent parts of the processed document.

The great aspect of XML files is that they are human-readable. To figure out how the objects, Metadata Extractor is seeking, are serialized in these files is not that difficult. In the next section, we describe what objects and information Metadata Extractor has to retrieve from the PowerDesigner data model files. We will see that the objects and their properties are quite complex. Having a stateful parser would bring advantage as the context of the XML elements matter. For example, an element "name" that Metadata Extractor has to obtain can be attached to an entity, an attribute and many different kinds of objects. Therefore, using a DOM parser is much more suitable and having the ability to do XPath queries over a DOM document is nicer than having to store a context manually, what would be needed to do with SAX.

4.1.2 Metadata to Collect from Data Models

The main goal of Metadata Extractor is to bring business data lineage and to extract metadata for objects in both business data lineage and the technical one. To meet the goal, we have to identify what objects and which of their metadata to collect from data models on every level of abstraction.

Metadata Extractor is aimed to bring information that is relevant to for data lineage and that may be visualized when presenting a flow of data. Modeling tools are exhaustive pieces of software with many features, thus they are able to capture plenty of aspects that a modeled system has. To filter only the metadata relevant for our problem, we made an analysis where we determine the interesting information that can appear in data models of PowerDesigner and ER/Studio. Before presenting results of the analysis, let us mention two important categories of metadata that we assume irrelevant to retrieve by Metadata Extractor.

Firstly, our tool does not pay attention to relationships in the entity-relationship model. This is given by the nature of Manta Flow. It is not a modeling tool, thus it does not work with relations. Edges between objects are used solely to represent a flow of data in a given system. The only relationship type Metadata Extractor needs to cope with is the inheritance relation which is present in the enhanced-entity-relationship model. The reason is that without capturing the is-a links, an entity may not be described completely and the attributes that were inherited may be missing.

The second category of information that Metadata Extractor will ignore is the one that describe constraints on the actual data records saved in a database. Metadata Extractor works, just like Manta Flow, exclusively with database's metadata and does not access the data really saved in a storage. Thus, our

solution neither can monitor nor enforce any constraint on database entries. That is why the metadata like keys or data types, which are also defined in data models, will not be in our domain of interest.

In this section, we list the specific types of objects we considered important for data lineage, and what are the means to describe these objects even further by properties of theirs. The full list of metadata, including the ones we do not find useful to extract, is listed in Appendix A.4.

Conceptual & Logical Data Model

Here we go through objects that appear in both conceptual and logical data models, together with what additional metadata can be attached to them.

ER/Studio

CDM

ER/Studio Data Architect does not support conceptual data models.

LDM

- *Owner*: An owner is a concept equivalent to a schema - it is a container for logically related entities. Every entity belongs to an owner.
- *Entity*
  - Name: The name of an entity.
  - Attributes: Attributes assigned to an entity.
  - Definition: Further description of an entity. Plain text or RTF (rich text format).
  - Note: Notes are used when documentation about an entity is generated. Plain text or RTF.
  - Where Used: This property shows objects that are in maps-to relation with an entity. Those which were created by generating.
  - User-Defined Mappings: Shows objects that are in maps-to relation with an entity. These mappings are user-defined. They can contain a description of a relation, but we do not fetch the text as Manta Flow does not support attributes on mapping edges.
  - Owner: The owner an entity belongs to.
- *Attribute*
  - Name
  - Definition
  - Notes
  - Where Used
  - User-Defined Mappings

PowerDesigner

Conceptual and logical data models in PowerDesigner have so much in common that we propose a unified view on what may be stored in them. The properties/objects that are specific for either of them are marked with information in brackets saying "CDM/LDM only".

CDM & LDM

- *Data Item*(CDM only): A data item holds an elementary piece of information, which is given by some fact, or a definition, in a modeled system. It may or may not be present as a modeled object. Data items can be attached to entities to form their attributes. It is a datum that may seem relevant and is possible to capture at first but later may not be used as no entity needs it in the end.
  - Name
  - Code
  - Comment: Plain text short description.
  - Definition: An RTF description of an object.
  - Annotation: A further RTF description.
  - Keywords: Set of significant words specifying object's domain.
- *Entity*
  - Name
  - Attributes
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Dependencies: Mapped entities.
- *Attribute*
  - Name
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Parent Entity
  - Dependencies: Mapped attributes.

- *Inheritance*
  - Parent Entity: Predecessor.
  - Child Entity: Inheriting entity that takes over attributes of the parent.

Physical Data Model

In this section, we list objects from the physical level and their possible metadata.

ER/Studio

- Type of Data Model: Database management system that a model is aimed for.
- *Schema*
  - Name
  - Tables
- *Table*
  - Name
  - Columns
  - Schema
  - Definition
  - Note
  - Where Used
  - User-Defined Mappings
- *Column*
  - Name
  - Definition
  - Notes
  - Where Used
  - User-Defined Mappings

PowerDesigner

- *Table*
  - Name
  - Columns
  - Code
  - Comment
  - Definition

  - Annotation
  - Keywords
  - Schema
  - Dependencies

- *Column*
  - Name
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Table
  - Dependencies

4.1.3 Reconstruction of Data Model

Earlier in Section 4.1.2 we defined what are the objects and their properties that Metadata Extractor must obtain from data models. The objects living in common environment of a data model, are related to each other. The relations form some kind of hierarchy. The listing of the objects with their characteristics, which Metadata Extractor will retrieve from data models, makes us realize that the very basic layout of objects captured by a model resembles a tree-like structure. The reason is that the analysis implies a general skeleton of a data model that goes like shown in Figure 4.2. However, it is just a simplified view. For example, data models do not need to support the concept of owners/schemas.

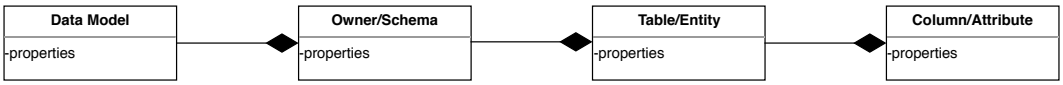

Figure 4.2: General hierarchy of objects in a data model. The UML diagram pictures a data model that contains an arbitrary number of owners (schemas respectively), they own 0 to N tables (entities resp.), while each entity consists of zero or more columns (attributes resp.). These objects can be seen as nodes of a tree, whereas their properties are attributes of the corresponding nodes.

Surely further relations between the objects will come to play, like inheritances or mappings. They are going to be discussed later in the Section 4.1.4, making the diagram of objects in a data model more complex.

Metadata Extractor builds the tree from the top to the bottom. In the next two subsections, we describe how the main objects from Figure 4.2 can be reconstructed from the output files of ER/Studio, PowerDesigner respectively.

ER/Studio

In the case of ER/Studio, each type of object is defined in a single table. Such a table stores all instances of the given type. An instance is identified in a table uniquely by its ID among other realizations of the same type. Therefore, this ID is a primary key of a table. The relations from Figure 4.2 are done by these keys. An object has a reference to its, if we stick with tree terminology, parent by storing parent's ID as a foreign key property of itself. Object's textual properties like name or definition are saved in a table containing string entries. Tables storing records of the main objects, therefore, have also a column with foreign keys pointing to the string table. So if Metadata Extractor proceeds by loading data models firstly, then descending down the tree, every time it is loading an object, its parent has already been constructed and the child can be simply plugged to the parent which its child references by a foreign key.

PowerDesigner

XML files form a tree structure by definition. This fact makes storing a hierarchy of objects with the same nature suitable and straightforward. This way a parent object of a child is simply its predecessor in the tree formed by XML elements of a file. Properties of an object are stored as child XML elements. When Metadata Extractor creates the result, the top-down traversal of an XML tree is followed. At first, on its way down from the XML root, it builds objects higher in the hierarchy and only if a parent is built, Extractor examines its children. This way the context is clear and once, let us say an attribute, is created the program knows what is its parent entity trivially - it must be the one that was created most recently.

4.1.4 Maps-to Relation

Once we identified possible objects across the data models, it is really useful to know which objects are related, even though they were not defined at the same level of abstraction. It is important to have the data models readable by keeping track of what tables are implementing which high-level concepts.

Our tool deals only with mappings of objects which are not at the same level of abstraction. Some modeling tools allow mapping, for example, one logical entity to another. However, it is unclear what is the meaning of such construct. One explanation would be that it expresses a relationship, but data modeling tools have different means available for defining this kind of relation. Possibly, it could indicate that two modeled entities are used identically as they are implemented by a single database table. But that is what data lineage describes precisely and Metadata Extractor solves better.

To be specific, Metadata Extractor copes only with the following types mappings:

- An entity to a table or another entity.
- An attribute to a column or another attribute.

Now we take a look at how the mappings are realized in the chosen data modeling tools.

ER/Studio

In Section 4.1.2 we mentioned two types of mapping relation in ER/Studio for entities, tables, attributes, and columns - where-used and user-defined. In fact, the meaning of both types is the same. The only difference is that the where-used mappings are generated automatically, while the user-defined are drawn manually by a user. We assumed that all the objects that can appear in a maps-to relation are in the very same solution, but there is also an option to create a mapping to objects which are defined in different .DM1 files. It can be done using the Compare and Merge utility in ER/Studio, its functionality is to synchronize a model with another model/live database/SQL file. Among other operations that keep pairs in sync, there is the mapping creation option. We are interested in the first scenario, where the two compared data models may originate even in two different solutions. This brings us third kind of maps-to relation. It is referred to as a universal mapping.

Knowing all the possible types of mapping relation, we will further focus on how they are saved in an ER/Studio .DM1 file and how can they be extracted.

Let us start with the seemingly easiest cases, with maps-to relation between objects inside the same ER/Studio solution, namely where-used and user-defined mappings. After some analysis using the reverse-engineering tool (see Section 3.1.2), we found a table that stores these mappings. It is named *Where_Used_PD*. In the table, there are four crucial attributes, IDs of the mapped objects and their Meta table types. The first two attributes are foreign keys to tables where the mapped objects are defined. The second pair of columns defines a type of the object so that Metadata Extractor knows to which tables it should look for the instances which are referenced by the foreign keys. The Meta table types also allow us to check, if the objects are actually compatible with each other in the sense of how we constrained the allowed mappings in Section 4.1.4.

At first sight, solving the universal mappings may appear more difficult. It looks like Metadata Extractor will need to search for an object in different ER/Studio solutions and reconstruct the external objects. But the way universal mappings are realized in ER/Studio is much simpler. Metadata Extractor does not need to open other files, as the external objects referenced by a universal mapping are stored in the analyzed .DM1 file. They are described briefly in a table called *External_Mapped_Objects* by XML structures. Metadata Extractor uses SAX parser to load these objects, as the XML object description is simple. Finally, the table called *Universal_Mappings*, where all the mappings to external objects are defined is using the same concepts as *Where_Used_PD* allowing Metadata Extractor to reconstruct them easily. The external objects must get reconstructed in such a way that once the solution in which they appear as internal objects gets processed, the two equivalent representations are merged.

PowerDesigner

PowerDesigner saves every data model into a separate file. Therefore, mapping resolution is not as straightforward as in the case of ER/Studio. The problem is that every mapping of an object to another one, that is on a different abstraction layer, leads across PowerDesigner files. Firstly, let us go through the possible types of mappings, then we will discuss how to resolve the relation efficiently.

The mappings are divided into two categories. Similarly as in the first modeling tool, the mappings may be either generated or user-defined.

Before going through how the mappings are represented, we must mention the way objects taking part in the relation are identified. Every important object in PowerDesigner has a unique identifier named *ObjectID*. This sequence of characters is used when referring to the object.

When an object is created out of an existing one by generating, the resulting object has an XML element named *History*. The element contains all the IDs of objects that the final object was generated from.

User-defined mappings are represented by a composite XML structure. The element representing a single mapping consists of a pair of mapped entities/tables IDs. If sub-objects of the pair are tied together by the maps-to relation as well, the XML element is a parent of further XML elements, which are specifying IDs of the underlying attributes/columns that are mapped to each other.

However, knowing the types of mappings and how they are saved in the PowerDesigner files is not enough to reconstruct the relation. Let us imagine the following scenario. Metadata Extractor gets a file to process. It reconstructs objects stored in the file, then the program tries to resolve mappings, but one object of a mapped pair is not accessible, as it is defined in a different file than is the one Metadata Extractor is currently reading. Only the ID of the mapped counterpart is known. In order to gather all the required metadata about the mapped object, Metadata Extractor needs to find it by its ID in the file where it is defined. In a situation like this, a data model file is dependent on others. In PowerDesigner's file format, Metadata Extractor can find an XML element describing targets of the given model, to learn what are the files the model depends on.

When a model is generated from a file, such dependency is saved to both of the data model files, in the generation target as well as in the origin. In other words, if we imagine an oriented graph, where a file is a node and an edge leads from a file a to file $b \iff b$ is listed as a dependency of a, then a bidirectional edge is created when models are generated.

Whereas, when a user-defined mapping is created from an object in a source model c to an object in a target model d, only the $c \to d$ edge is created, and d has no knowledge about the mapping. Metadata Extractor must solve how a resolution of the external objects should be done. As it has no further knowledge than the target object's ID, not even information what file does it come from, a naïve approach would be hugely inefficient. The simple solution would search for every demanded ID across all targets of the processed model and once the ID is found, the objects get reconstructed. That potentially leads to a great number of file openings. Also if an object is referenced from n mappings, it would have

to be reconstructed n times while still processing a single data model, leading to a huge time overhead. Surely, this solution can be improved by collecting the external IDs and postponing the resolution to the end, once all the needed external objects are known. So when processing a single file, each of its targets would be opened only once and the reconstruction of the object would take place one time as well. But still, if an object is referenced from n different models, it would be reconstructed $n + 1$ times, which definitely not ideal.

If we went in a different direction and processed each of the models once, constructed all their objects, then stored all of them by their IDs and once all the data models that Metadata Extractor had to process are loaded, resolved mappings. Advancing like this would decrease the count of reconstructions and file openings to the ideal amount but eventually, if the number of inputted data models is big, the size of memory claimed by Metadata Extractor could become unbearable.

To achieve a solution that would have the advantages of both of the approaches, Metadata Extractor needs to split the set of input models into disjoint subsets that represent the smallest group of logically tied files. Metadata Extractor transforms all of the unidirectional edges in the dependency graph described above (and illustrated in Figure 4.3), into bidirectional and finds connected components. These components are the logical subsets we were looking for. Therefore, the program can reconstruct objects from a single component and make the resolution at the end. That keeps the storage acquired by loaded objects low and at the same time objects are constructed only once, while files don't get opened multiple times. We assume that the far most common use-case is having components of size three with three data models - logical, conceptual and physical (or few physical ones).

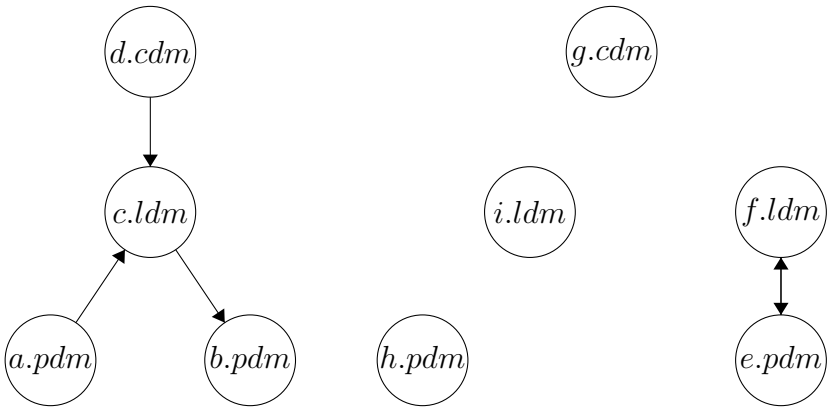

Figure 4.3: On the example we have five connected components made of PowerDesigner data models. If dependencies of data models are like this, a.pdm should get processed and resolved in one batch with b.pdm, c.ldm and d.cdm. Then f.ldm must be handled with e.pdm, while rest of the models are independent and form component of size one.

However, when dealing with the dependencies, there is one more trap remaining that may cause problems. The basic scenario how a user behaves when

using Metadata Extractor is that he works with PowerDesigner data models that are saved somewhere, for example in C:/PowerDesigner/Project/, and once he wants to let them analyze by Metadata Extractor, he drags them to a different directory that is used as input for our tool. This way, paths pointing to target models become incorrect, since the paths don't get updated until the moved files are not opened again. So the models in input still depend on files in C:/PowerDesigner/Project/ which may be not existing anymore. Alternatively, if the paths remain incorrect there may happen a situation like this - we have a file a referring to b in, both of them in the input folder, but a change of an object in C:/PowerDesigner/Project/b affects a that is using the changed object. Anyway, we want Metadata Extractor to work only with the files that a user explicitly marked as to process, those are the ones present in the input directory. So Metadata Extractor has to have a fallback for this situation and should try to find target files in the input folder. The idea is to try the ideal scenario and check whether the target is in the input directory. If yes, the resolution is done. Otherwise, the program assumes the name of the target file has not been changed and that the former and the input directory have similar structure.

4.1.5 Business Lineage Creation

One of the main goals of this work is to make Metadata Extractor capable of creating business[1] data lineage automatically. The way we chose to approach the problem is that Metadata Extractor builds the high-level lineage based on the technical one created by Manta Flow. The physical lineage provides the most precise foundation possible. It is directly aligned with how a live database system is really used and how data travels through it. Despite the fact that Manta Flow primarily focuses on the analysis of physical data flow, there is an already implemented functionality, which can propagate lineage from physical objects via mapping edges to objects on higher levels of abstraction. The process of propagating the data flow acquired on the physical level to the logical or conceptual level is called *interpolation*. The crucial part is to ensure that Metadata Extractor merges correctly physical modeled objects with their database counterparts that are used by Manta Flow.

4.1.6 Database Connections

Even though Metadata Extractor will not connect to databases directly, it needs to gain details of database connections when working with physical models. Only then Metadata Extractor is able to know what specific database the extracted physical objects belong to. Firstly, we have to answer the question why do we actually need the connections and then we will look at how in case of PowerDesigner and ER/Studio the connection can be made and obtained.

What Metadata Extractor does is not accessing metastores [2] of databases. Getting metadata directly is not really straightforward as each database technology has its own specifics - types of metadata and their organization vary greatly.

[1] By business lineage we mean data lineage that is formed on a higher level on abstraction than on the physical level.

[2] Shortly for metadata storage

Instead, we make use of the fact that Manta Flow does has connectors that do the job for Metadata Extractor, the data lineage tool stores the metadata in its own local database and has unified API for getting the metadata independently of a database engine.

Why would we actually want to request another metadata when that is just what Metadata Extractor is getting from physical data models? Because we are interested in data lineage, which is created by Manta Flow based on the real metadata of physical objects that are present in a database. The simple view is that at the moment when we ask for the objects from Manta Flow's metastore, the analysis of data flow has already taken part, thus there are data lineage edges between physical objects. To bring together both features of Manta Flow and Metadata Extractor, the program must merge equivalent physical objects which come from both sources - from Manta's extraction and from our tool. Thus, only if Metadata Extractor is on the same page with Manta Flow - knows what specific database at what specific server the modeled objects belong to, the program can ensure correct pairing of the objects and data lineage.

The details we use for identification of a database instance are the following:

- Database Type (Technology)
- Database Name
- Server Name
- Schema Name
- User Name

All the above can be stored in one property called connection string (see Section 2.1 in which the term was introduced).
To this set of the listed properties and a connection string, we refer as a *connection*.
Since each physical data model describes a single database (or its subset) we need exactly one connection for each processed physical model in order to achieve what we described just above.

However, the identification of a database that a physical data model reflects cannot be found in the file storing the physical model, we have to inspect other possibilities for obtaining the database details.

ER/Studio

Databases whose models are created in ER/Studio can be accessed only by ODBC drivers present in the used machine.

Metadata Extractor is not able to reach these drivers generally so we will leave it up to a user to define connection parameters by hand. A .ini file is used for this. It contains sections that are named by corresponding physical models and connection details are specified inside a section.

Metadata Extractor tries to get as much information as possible from an ER/Studio data model, however, the name of a database and server must be added manually by a user as the tool does not save these data.

PowerDesigner

In PowerDesigner a user has multiple options for connecting to a database to choose from. Either ODBC or JDBC connection may be used. The modeling tool has a nice user environment for creating or using connections to a database where a user is guided through the setup. He can test if he did set up everything correctly. There are two options of how to connect to a database using the interface that may be helpful for Metadata Extractor, using .dsn and .dcp files.

The .dsn files are definitions of ODBC connection containing parameters for an ODBC driver and store all the interesting information we would like to have. The drawback of this file format is that it varies from a database technology to database technology - the properties representing the same concepts may be called differently. That means we would need to have a custom parser for each supported database engine.

On the other hand, there is the possibility of connecting via .dcp files. They can store information about a native DBMS connection or about a JDBC connection. The nice fact about them is that they are not that flexible and once we know whether dealing with a native or JDBC connection respectively, the structure can be parsed easily. A .dcp file consists of a property=value map. There are a couple of properties common for both types, like description and user name. Then the most important property of a JDBC .dcp file is the JDBC connection URL - in other words, a connection string that sufficiently defines a connection. In the case of the native DBMS variant, Server Name along with Database Name are crucial, in order to identify a database we are connecting to by the setup.

But there is a problem that is common for both of the described approaches - there is no link between a connection file and a model that corresponds to the connection. So Metadata Extractor has to stick with the same solution as in the case of ER/Studio - to use auxiliary .ini files (described above in Section 4.1.6).

4.1.7 Output Representation

The output of Metadata Extractor is a graph. Its nodes are the important extracted objects such as entity, attribute, etc. properties of the nodes will be the metadata acquired about the extracted objects. Edges of the output graph are of different types, they can either stand for mapping of the nodes of indicating the flow of data between them. The remaining part is how technically Metadata Extractor represents its graph output. We require data structure that is both convenient to work with programmatically as a data structure and able to be visually presented to a user of our tool. Also, we must take into account that Metadata Extractor is going to be plugged into an already existing software environment. Manta Flow is backed by a database storing graphs of data lineage. There is an already existing browser-based user interface. Using the interface, data flows can be previewed interactively. Given that we would like to comply with the graph database and merge created graphs to the storage, and at the same time have the ability to reuse the visualization for presentation of output. The most natural solution is to stick with the very same representation of a graph as Manta does. In the alternative scenario, when we don't want to let Manta handle

the output, there is a possibility of using a writer which produces an image, or a textual representation of the output at a local machine, in contrary to sending the output to the Manta Server where the graph database is located.

4.2 Desired Features

The analysis forms a set of functional requirements or features we expect Metadata Extractor to have:

- Load objects and their metadata from the output of modeling tools and reconstruct the hierarchy of the objects.
  - ER/Studio: Logical data model & physical data model.
  - PowerDesigner: Conceptual data model, logical data model & physical data model.
- Resolve mappings leading between objects originating in different data models.
- For every physical data model, obtain connection details to the database counterpart.
- Match the loaded physical objects with their equivalents extracted by Manta Flow if possible, in order to bring in the physical data lineage they take part in, so the business lineage can be interpolated.
- Create a graph out of the loaded structure.
  So that it can be further:
  - Displayed in the user interface of Manta.
  - Printed to a file as image.

4.3 Survey of Existing Solutions

We are working on the development of an automated solution that delivers business lineage. In order to justify that we are not reinventing the wheel let us have a look at the software that can provide similar functionality as Metadata Extractor.

The competitors can be divided into multiple categories:

- Data Governance Frameworks
  Data governance is a discipline that helps enterprises to gain control over their data. Commonly data lineage is a part of the functionality that data governance solutions provide.
  Usually, the solutions work with *business glossary* which is a set of terms used in business together with their definitions specifying what they precisely mean in a domain. It unifies a vocabulary between the system's stakeholders to avoid misinterpretations when it comes to high-level terms.

  - Collibra
    Works with business assets that connect business terms from glossary to data assets (eg. database column or table). The connections are established manually [18]. In data lineage diagram business terms can be displayed along with the related data assets to ensure better traceability [19].
  - Informatica Axon & EDC
    The solution by Informatica Corporation works on a very similar base as the previous one. Data assets are connected by hand in a user interface to business glossary entries[20]. That allows, once a technical data lineage is created, to drill down to the data lineage going through the mapped database elements. In the data flow can be also found related business assets next to related tables.
  - IBM IGC
    IBM approaches to data lineage in such a way that it only displays assets that should be relevant for a business user. In fact, it is just a subset of technical lineage and what is shown is picked by a user [21].

- Data Lineage Tools
  A data lineage company ASG Technologies seem to do something with modeling tools, as they apparently dispose of connectors for some modeling software. However, no appropriate documentation can be found and the latest update traceable on ER/Studio connector was made in early 2014. [22] The supported version of ER/Studio is 9.7, while version 18.0 is out today. Similarly with their PowerDesigner connector, it is not easy to find documentation and even if something related is mentioned the information looks to be obsolete nowadays.

- Modeling tools
  Both of the analyzed tools, ER/Studio and PowerDesigner, have means to create something like data lineage models, or lineage can be specified by mappings in a single data model. The problem with this approach is that it is not based on an analysis of SQL code managing the database and the approach is not automated. Creating such models is exhaustive and error-prone as a user has to define the flow all by himself.

To our knowledge none of the solutions disposes of the automated functionality we aim to provide by putting together modeling tools, Manta Flow, and finally Metadata Extractor. That is to create an abstraction over technical details of databases, summarizing the real data flow using business vocabulary.

4.4 Architecture of the System

Metadata Extractor is able to process output files of two modeling tools - ER/Studio and PowerDesigner. For each tool Metadata Extractor has a separate part, both of them consists of the following major components:

- Model [3]
  This component is a read-only description of a data model source. On one hand, it reflects the raw structure of a file so no information is left out when compared to the source. On the other hand, it allows reading access to the modeled objects we are interested in that were reconstructed in a convenient fashion.

- Resolver
  This is the part where the logic of construction of objects from a file is hidden and the loading of the model is done.

- Parser
  Loads a file into data structures that can be further worked with.

- Reader
  Puts together the Model, Parser and Resolver units - creates a model from parsed file using the resolver and hands the result to Data Flow Generator.

- Data Flow Generator
  Creates a graph representation out of the output of the reader component using the modeling common module (introduced right after).

These two modules are shared:

- Modeling Common
  The common part for communicating with Manta Flow via its API in order to pair modeled objects with database objects extracted from live databases. Based on a correct pairing interpolation is done. The common part is responsible for creating node representation of objects that have no backing in the database dictionary of Manta Flow.

- Manta Flow
  The external part capable of extracting objects from databases, analyzing transformations on these objects, and creating data lineage based on the analysis.

The most important parts of which Metadata Extractor consists are shown in Figure 4.4.

[3] There is a naming collision but here we don't refer to any data model but a data structure that reproduces objects stored somewhere, which one of these two possible meanings we use should be clear from the context.

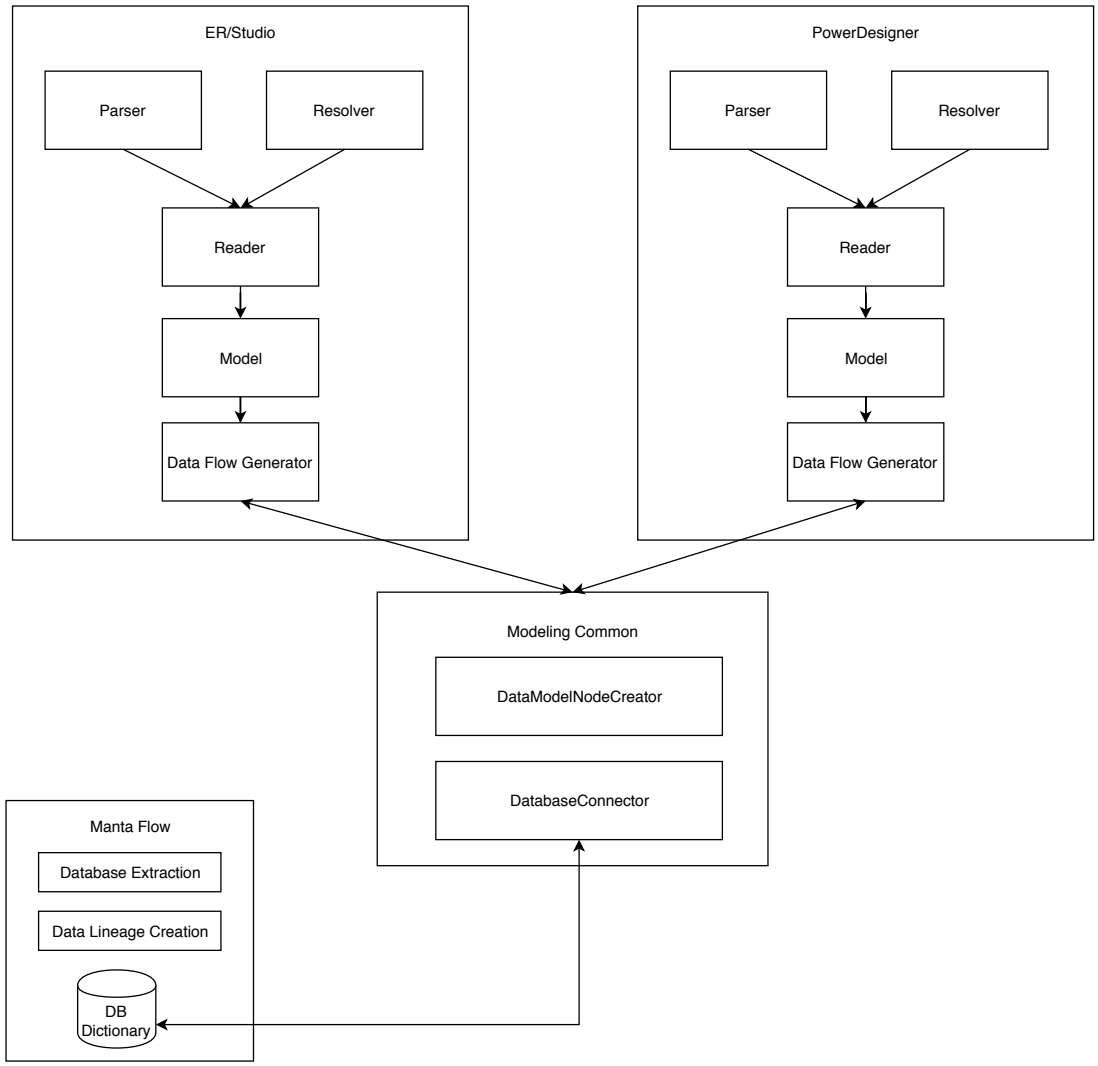

Figure 4.4: A high-level view on software architecture of Metadata Extractor

5. Implementation

In this chapter, we will describe the most important classes of Metadata Extractor and explain their main responsibilities. The software consists of the main three parts which will be discussed - ER/Studio module, PowerDesigner component and Modeling Common, which is the shared part. The common module is a general artifact that is designed to be helpful when designing a metadata extracting tool from an arbitrary modeling tool and the tool is aimed to be integrated into the ecosystem of Manta Flow.

We go through classes and methods from the source code of Metadata Extractor from a higher perspective. Even though not every class, interface, and method are described in the chapter, a detailed programmer's documentation containing this information is attached in the form of JavaDoc.

5.1 ER/Studio

Now let us introduce the parts of Metadata Extractor implementation that are coping with data models product of ER/Studio Data Architect.

5.1.1 File Reverse-Engineering Tool

In Section 4.1.1 we introduced the .DM1 file format, which is used for storing ER/Studio's outputs. The file format consists of CSV tables and its inner organization of data resembles a relational database. We need to decrypt the relations between the tables in .DM1 files, to be able to reconstruct objects stored in the files. That is because it is not easy to see, how structured information is stored in tables of the file format. Relationships in this type of tabular data storage are realized via primary and foreign keys. The task is to find out what tables are linked in the format, since it may uncover how complex objects are reconstructed to many interrelated tables. For this purpose, we developed a little reverse-engineering (RE) tool which should help us to gain the knowledge necessary for reconstruction needed information from the files.

Input for the reverse-engineering tool is an arbitrary .DM1 file, while output must describe the logical layout of the file structure. We observed the analogy of the file format's storage logic with relational databases. These databases can be represented understandably by a diagram. Thus, it would be nice, if the output of our utility would be such a diagram capturing the organization of .DM1 files. However, creating a relational diagram directly from the reverse-engineering tool would be too complicated. We may recall that for creation of relational diagrams modeling tools are used. The best solution would be to reuse this capability of theirs. Modeling tools usually have the ability to transform selected routines of SQL into a visual representation. For example, based on the statement "CREATE TABLE T" a modeling tool can draw a box in a diagram representing table T. By means of SQL, table columns and key constraints can be defined as well. If there is a foreign key column in table B, which is at the same time a primary one

in table A, it forms a relationship between the two tables, A and B. This relation is showed graphically in a diagram produced by modeling tools.

Therefore, if the RE tool is able to generate a SQL create procedure for each of the tables from ER/Studio's file format and to define primary and foreign keys on the tables afterwards, modeling tools can transform such code into a visual representation of the analyzed file format.

To define a SQL create statement for the tables of ER/Studio's file format is straightforward once a .DM1 file is parsed. For parsing, the tool uses the parser described in Section 5.1.2.

On the other hand, deciding which column of a table is primary, which columns are foreign keys and to what primary column every foreign key refers to, is a more complicated task. We needed to take the reverse-engineering process one step further - to be able to infer keys from tables. Modeling tools have the ability to obtain relationships between tables in a reverse-engineered relational database, however, the knowledge is based on metadata defining those keys. Databases can store such metadata, but plain SQL create statements that our RE tool produces, do not provide any additional information like that. What we needed was a utility inferring relations between tables based solely on column names or/and content of database tables. However, we have not found any suitable already existing solution. In this situation, we had no better option than to enrich our reverse-engineering tool of this ability. The output of the RE utility, therefore, needs to be a SQL script containing definitions of tables, columns, and key constraints.

As the development of this utility is not the focal point of this work, the RE tool is not going to be an out-of-the-box general solution for deducing keys of relational databases and it is primarily concerned with the objects and properties picked by the analysis in Section 4.1.2. Even though, it should provide us the needed overview of .DM1 files organization.

We had to decide what programming language to use when creating the reverse-engineering utility. Although there may be scripting languages that would make some operations, like joins on tables, easier, we decided to use Java. Metadata Extractor itself is written in it and there is an important module - Parser that the column deducing utility and Extractor require. By developing both in Java it is possible to reuse the parser component.

Let us now describe the workflow of the RE tool.

To begin with, an analyzed .DM1 file is parsed into a set of instances of the class `CsvTable`. When trying to find out what columns are key constrained, the idea is to look at the loaded tables from two different views.

The first perspective is to take into account only metadata of tables. This approach is represented by the class `DependencyCreator`. It treats columns that look like keys (e.g., those which name ends with the "ID" suffix. The policies are determined by `isOnBlacklist` method) as if they were equivalent across all loaded tables. The most important method in the class is `createDependencies`, which pairs potential key columns and tables containing the candidate columns. Then the class `RelationFinderByTableDefinitions` allows querying over the structure found by `DependecyCreator`, showing what tables are possibly related through a series of joins. Among others, there is a general method `getDependenciesWithPaths` showing the sequence of joins needed to do, in order

to put together records from table A with those from table B.

The second view takes into account data stored in tables as well. When analyzing what are relations between tables, we consider a table being a collection of columns. The `CsvColumn` class represents a column, storing its name and a set of values from all records of a single table at the position of the given column. The `RelationFinderByTableContents` class is used for this approach when further searching for key columns. This way, the RE tool examines if columns with the same names have corresponding contents.

An instance of `RelationFinderByTableContents` filters tables that have no content, thus are irrelevant for this approach. The class has a method determining if a column can be considered the primary key of a table. That is true only if such a column contains solely distinct values, as a primary key must be completely unique in a table. The relation finder also has a method finding out if a column can possibly be storing foreign keys. The idea is that if column A contains foreign keys referring to primary keys specified in column B, each entry of A must be present in B, thus A must be a subset of B. What more, in .DM1 file format is a table *Identity_Value* that explicitly pairs tables with their primary key columns. The reverse-engineering tool takes advantage of this information as well. The finder class also inspects if the primary keys are defined in some predefined order or if there is some similar pattern of how primary keys are listed in a .DM1 table.

The classes above allowed us to look from multiple perspectives at metadata and data of tables in the analyzed file format. To put it all together, to finally generate the output in the form of SQL code defining .DM1 tables and their keys based on the analysis, the class `TablesToSQLConverter` is used. Its crucial method is `writeMySQLSource`. At the beginning it defines a SQL create statement for each table from an ER/Studio source file. Then takes place the phase of creating key constraints. It works with candidate columns that satisfy policies for key-containing columns defined in the `DepedencyCreator` class. Also, these columns must be present in at least one table, so one or more joins can be made using the keys the columns store. The converter class also provides space for a manual definition of pairs table and its primary key that were identified by inspecting the file format using `RelationFinder*` classes. Then resolution of keys goes like this: at first place, primary keys defined by a user are set, then the information stored in *Identity_Value* table is used. Lastly, the key inference based on policies taking into account column names takes place. For example, if a column is called tableA_id, it is very likely that it stores primary keys if it comes from a table called tableA. If a column is set as primary key storage in one table, for each of the remaining tables that contain a column with the same name, the column is marked as a foreign key referring to the first table. In the case that there are columns, which were chosen as candidates for being key containers, but no constraint was assigned to them in previous steps, it goes as follows. One table which contain the given column is marked as the source of primary keys and the remaining are considered to store references to the chosen table. The last approach may cause slight inaccuracies in the final result, but the main message should not get lost.

5.1.2 Parser

We already discussed in Section 4.1.1 that the files storing objects we want to recreate from ER/Studio are using format, which is basically a sequence of CSV tables. In order to be able to further process the data saved in the file format, programs need to load the contents of those tables into corresponding data structures. That is what parsers are for.

Already existing CSV parsers are made to process a single CSV table per file. There is no unified definition for the comma-separated-value files, but usually, they do not allow naming CSV tables as present in .DM1 files. What is important to say about a CSV table is that it consists of entries. Entry is located on a separate line and consists of fields and the last record is followed by a line break. Fields in an entry are separated by a comma. A field may contain a comma or line break, then it must be enclosed in double quotes. If a double quote appears in a field, it must be in the enclosed section and the quote itself must be doubled [23].

What more, the structure we need to parse consists of many CSV tables. Each of them has its name, definition of columns (a special entry that is the first entry in a table and its fields are names of columns), and entries. Two tables are separated by a single empty line.

Putting it all together, it seemed to be easier to develop our own parser for .DM1 files that processes the files into a set of tables identified by their names.

A single table is represented by an instance of the class `CsvTable`, which contains its name as a string, references definition of its columns stored in a realization of the class `CsvColumnDefinition`, and finally holds records themselves, they are instances of the `CsvColumn` class.

However, the view on a table may vary by a specific usage of the parser. Using the reverse-engineering tool (Section 5.1.1), we inspected what values are stored in columns, examined ranges of values across all records of a table, etc. On the other hand, if we see a table as records, where each record has properties, one property corresponding to one column, from this perspective a table is a set of rows. The second case is required by Metadata Extractor when it is reading the contents of a table for the sake of obtaining saved records. Once the most important responsibility of parser is correct - recognizing individual parts of a CSV table, the module can be modified easily to produce data structures compliant with either of the views.

Getting the name of a table is an easy job, as the name is represented by a single line of text. To resolve a record - to get the individual fields separated by a comma correctly, is a slightly more challenging task since the rules described at the beginning of the chapter must be taken into account. We designed an automaton that accepts well-formed records of a .DM1 CSV table and recognizes its fields.

We propose a non-deterministic finite automaton, even though it can be translated to a deterministic one, we consider the NFA to be more clear and descriptive in this situation. Set of states is $Q = \{i, q, e, a, f\}$. Alphabet Σ is a set of chars that can appear in a string, since that is what the automaton processes, together with λ - an empty string. i is the initial state. Transitions are illustrated in the following Figure 5.1.

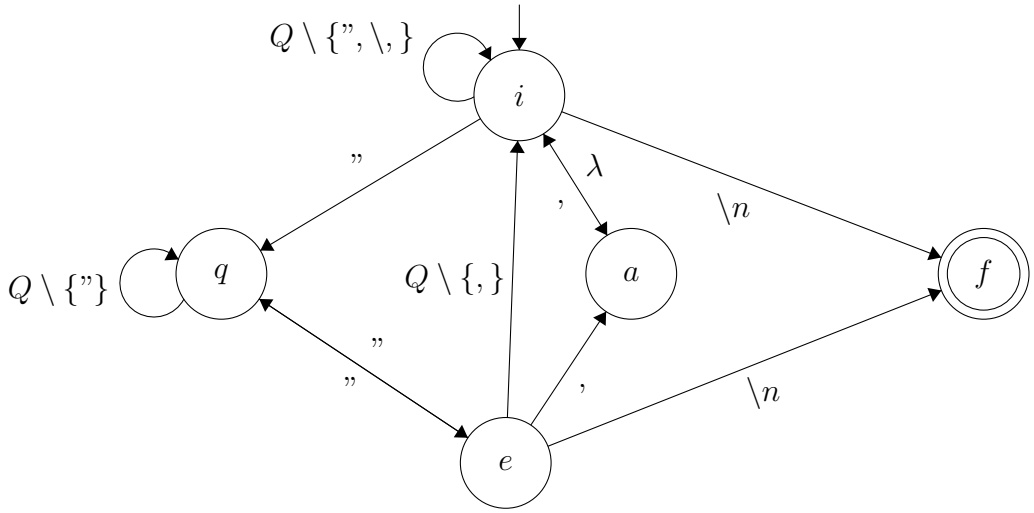

Figure 5.1: Non deterministic finite automaton accepting a record of a CSV table in .DM1 file. In the state a and f, the end of a CSV field is recognized and added to the record.

When loading a CSV record consisting of multiple fields not only the parser must determine the end of an entry, but has to collect its fields correctly as well. That is what the, at first sight redundant, state a expresses. Once the automaton gets to the state a, the end of a field in the input is indicated. The boundary of the last field of an entry is, however, recognized in the accepting state f.

The parser's interface is a single method - `readFile`. It takes a file to process and returns table name and instance of `CsvTable` pairs.

5.1.3 Model

The purpose of the Model component is to provide read access to both a raw structure of a processed source file and a fully loaded hierarchy of objects Metadata Extractor has reconstructed from a file.

The crucial objects and the important properties of theirs which Model is required to capture result from the analysis of ER/Studio data models listed in Section 4.1.2. They are described by functionality that is expected from them, making Model a set of interfaces. Here we describe them in more detail.

The perspective of a raw file structure loaded to memory is represented by the interface `ErStudioFile`, where either all tables from a file can be retrieved via `getAllCsvTables` method or a single table may be obtained by its name using the `getCsvTable` method.

An ER/Studio solution - a set containing one logical data model and an arbitrary number of physical models, implements the `ErStudioSolution` interface. All the objects contained in the data models of a solution can be retrieved using the interface's methods. A solution is defined in a file, the relative name of the file where the solution is an internal one is returned when called `getFileName`.

A .DM1 file is not restricted to store solely objects from a single solution but may reference external models (objects that are originating in another solution) as well, thus there can be objects from multiple solutions saved in a file. Each object is assigned to a solution where it originates. If the name of the object's origin

solution is identical with the name of .DM1 file from which it was loaded, it is an internal object, otherwise, we call it external. There is a single internal solution per .DM1 file. The overall structure of a file is represented by a realization of the `ErStudioFileModel` interface that consists of a single internal solution and the file may reference some external solutions.

A base behavior of both logical and physical data models is defined in the `DataModel` interface. A data model has a name and contains owners. An instance fulfilling `DataModel` contract must have property of type `AbstractionLayer` set to either `Logical` or `Physical` just like every underlying object in a model. It must be invariant that the whole subtree of objects, a root is a data model, has defined the same abstraction layer.

`PhysicalDataModel` interface requires an additional feature - its realization can tell the database platform it is designed for. In the enum class `Platform`, all the database management systems ER/Studio supports are captured with an extra entry for an unknown DBMS.

Models are storing instances of complying with the `Owner` interface. An owner is either logical or physical according to what objects it can own and what type of model it may belong to. It has a name and allows access to the objects owned.

The generic interface `Mappable<T>` is used by objects that can be mapped to `DataObject`s (see the following paragraph) which are on the different abstraction level. The type parameter defines what is the specific kind of the mapped counterpart.

Important objects in a data model that can be described further by definition and note, while having a name extend the general `DataObject` interface which extends the general `NodeMetadata` contract (see Section 5.3). These can be either `CompositeDataObject` or `SimpleDataObject`.

An interface for composite objects describes what features are expected from a table or an entity. The two types have the very same set of properties that is why a single class is enough to capture both. Which of the two kinds an instance represents is decided by its `AbstractionLayer` attribute - the logical layer indicates an entity, the physical a table. A composite one can be mapped to another `CompositeDataObject` and may contain simple objects.

The `SimpleDataObject` interface is used to represent columns or attributes. Equivalently, the layer of abstraction is crucial to determine, whether an object fulfilling a simple object's contract can be stored in `PhysicalDataModel` or `LogicalDataModel`.

The reason to represent the pairs of, at first sight, different concepts of tables-entities just like columns-attributes in a single class is based on how ER/Studio treats them internally. Given that columns and attributes, tables and entities equivalently, are stored in a single CSV table and the distinction whether an object is of one or the other type is made based only on the fact to which data model, physical or logical, it belongs to. The two types have the very same set of possible attributes. Even data models are treated similarly, only there is the additional attribute related to DBMS type in the case of physical models, while the logical ones do not need such information.

5.1.4 Resolver

The Resolver unit is here to provide the construction logic for building the objects which the model component describes.

Outline of what the Resolver module does is that using an instance of the `ErStudioFileModelBuilder` class it creates the whole structure representing an input .DM1 file as an object described by the `ErStudioFileModel` interface. It builds an internal solution, puts inner objects to maps-to relation, then loads external objects and at the end resolves mappings that lead across solutions using an instance of the `ExternalMapper` class. Parser provides an input for Resolver.

Typically, for each Model's interface we create an implementation. In Java, it is usual to call the classes fulfilling a contract `*Impl` where * is a placeholder for the name of an interface. We stick to this naming convention. The `*Impl` classes, in contrast to the interfaces that are used to describe Model, need to dispose of methods for setting up and adding properties or sub-objects. These classes can be found in the "imp" package.

We defined what is expected from the objects collected by Metadata Extractor and added the functionality to set up these objects. So the last missing piece of the puzzle needed is to put together the gathered information to recompose the final objects and their metadata.

The logic taking care of the objects reconstruction can be found in the Resolver's package "build". The skeleton of how an instance of `ErStudioSolution` is created is prescribed by the method `buildErStudioModel` in the class `AbstractDataObjectsBuilder`. The method follows the paradigm of the template method design pattern and defines the steps needed to take, in order to construct an instance of `ErStudioSolution` solution, no matter if it is of the internal or external type. The template enforces a tree of objects in a solution that is built from the top to the bottom. The very first step is to create a root of such tree - an instance of `ErStudioSolution` that is being built, then `DataModel` structures from the solution are loaded, underlying `Owner` implementations follow, `CompositeObject` realizations after, and finally `SimpleObject` instances. Child objects are attached to their parents just after they are created. The abstract builder contains common methods for the construction of the needed objects. Those methods are not dependent on representation, they only require a set of properties as the input, to build the resulting objects from. The groups of properties needed to be collected about each type of complex object from a data model to proceed the construction are described by the interfaces in the package " modeledobjectproperties".

The specific way of gathering information about objects to be reconstructed is tied with details of how the objects are saved. Concrete builders realized by the classes `InternalDataObjectsBuilder` and `ExternalDataObjectsBuilder` provide implementation of the abstract object builder. In the case of an internal solution, information about objects is retrieved from CSV tables, where for each table there is a dedicated and references across the tables are resolved. On the other hand, in the case of external solutions, they are fully represented in a single CSV table that stores XML structures describing these objects. `ExternalDataObjectsBuilder` uses a simple SAX XML parser to retrieve the data required for object reconstruction from the XMLs.

`InternalDataObjectsBuilder` has one more responsibility, it is to create mappings between `Mappable` objects in the internal solution with a help of `InternalMapper`.

The `InternalMapper` class only has knowledge about the layout of a CSV table containing internal mappings. Similarly, the `ExternalMapper` class knows about the layout of the table storing external mappings. However, the logic of putting objects to maps-to relation is coded in their common predecessor of these two classes - `AbstractMapper`. The mapping logic goes through a given CSV table, which definition fulfills the interface `MappingTable` and links pairs listed in the table by mapping, if the objects to be mapped are compliant.

5.1.5 Reader

The Reader component puts together the parser module with the resolver module and produces data structures described by the model component.

An instance of the `ErStudioDm1Reader` class crawls through a directory with input .DM1 files and processes them. Each time the `read` method is called, it forwards the next file from the input folder to the parser. Result of the parser is handed to the resolver, which produces a result fulfilling the contract of `ErStudioFileModel` and that is what the `read` methods returns.

5.1.6 Data Flow Generator

Having constructed the objects defined in the model unit, Metadata Extractor sends them to the Data Flow Generator unit, where an instance of `ErStudioSolution` gets transformed into an output graph. This component puts together the functionality of Manta Flow with Metadata Extractor.

Before describing the workflow of the generator unit, let us mention its layout. There is a scenario that executes independent tasks. A task is a routine that has an input and an output. In our case we use a single task, whose input is a data structure described by the model unit and output a graph with extracted information.

Namely, realization of the class `ErStudioDataflowScenario` reads an input file, and executes the task - given by an instance of the `ErStudioDataflowTask` class.

The method which tasks must override and gets called is the `doExecute`. In this particular task it is needed to go through the whole hierarchy of the input data structure - `ErStudioFileModel` unroll solutions - internal as well as external.

It is crucial to identify what database an extracted physical data model corresponds to. This is important for Metadata Extractor to be able to pair physical data model objects with those from database dictionaries. The correct pairings allow Manta to interpolate technical lineage on the logical layer, as well it brings extracted metadata to database objects in the technical lineage. A suitable realization of the `Connection` class corresponding to a physical data model is made of information extracted from a data model - DBMS type and schema, the remaining details - database name, server entered by a user in a .ini file.

Once the connections are assigned to physical models, Metadata Extractor traverses trees of data models, and creates nodes using a realization of the `DatabaseConnector` interface in the case of physical level. When it comes to logical objects, Metadata Extractor takes an advantage of an implementation of the `DataModelNodeCreator` interface (both contracts and their default realizations are defined in the modeling common component described later Section 5.3).

The only item left is to create mapping edges for the corresponding nodes. Metadata Extractor creates the mappings symmetrically when resolving. It means the full information is captured on both interconnected levels. This way, it is possible to create all the mapping edges by going through all objects from a single level of abstraction, no matter which level as each mapping edge is contained in both of the levels. Thus, Data Flow Generator goes through all the logical objects, creating their node representation and collects the attributes that have to be linked to a column. When a node representation of physical objects is takes place Metadata Extractor checks if the just constructed node has a mapping, a MAPS_TO edge is set between the logical and physical node.

5.2 PowerDesigner

Now we move on to describe the implementation of Metadata Extractor's components which process data models created by SAP PowerDesigner.

5.2.1 Parser

To load data models that are the output of PowerDesigner and to reconstruct objects they save, Metadata Extractor needs to parse them. The files are of XML type. In the analysis section, we already discussed the possibilities when it comes to reading XML files. We concluded that the most suitable approach for us should be loading inputs to a DOM tree data structure. We did not try to reinvent the wheel, as there are many DOM parsing services that can be reused. Given that in the ecosystem of Manta software there is present an internal artifact capable of transforming XML to DOM, Metadata Extractor will make use of it.

5.2.2 Model

The Model component is the mean we use for the description of the main modeled objects that Metadata Extractor has to obtain Let us start describing these objects generally.

Thus the `DataModel` interface requires the name of the file where it is stored and its children. Then the hierarchy of objects goes like this: a model has at least one realization of the `Schema` interface, schemas store instances of the `CompositeObject` interface, which contains implementations of the `SimpleObject` contract. Every PowerDesigner's data model is stored in a separate file, thus a data model is identified uniquely by where it is stored on disk. This basic tree structured skeleton is present in each of the three data models - conceptual, physical and logical. What more, on the logical and conceptual level, since they are represented by an EER diagram, an entity can inherit from another one, thus the interface `Entity`, which introduces the concept of a parent entity,

will be used by on these layers. Each of the important objects may have some metadata providing further description and extends the interface of `NamedObject` (see Section 5.3). These interfaces are generic and provide a common basis for corresponding specific interfaces. For every model type, there exists a package, where those specific contracts of modeled objects lie. The concrete interfaces, on one hand, take over the common concepts, while on the other hand may be easily extended, once a new functionality, which is specific for any abstraction level, is identified.

`Mappable` interface is present to capture objects that are in maps-to relation. It realized via instances of `CompositeObject` or `SimpleObject`, however a target of a mapping is a string every time - it is a globally unique ID of the mapped counterpart, not another instance directly. That is because mappings in PowerDesigner leads across different files, thus an actual representation of the counterpart may be unreachable yet.

5.2.3 Resolver

The Resolver component provides means of creating objects, which are described in Model using data that DOM parser obtained from an input file. The output of Resolver is an instance of `DataModel`.

Abstract classes, prefixed with `Abstract*`, capturing common concepts are present to minimize code redundancy and to provide a unified way to handle similar objects, even though they do not represent any complete object of a data model. The actual objects that have backing in data models are implemented via `*Impl` classes.

The construction logic of a data model is concentrated in the "build" package.

The API for creating a data structure representing a data model is simply defined by the `DataModelBuilder` class, which is containing the method `buildDataModel`. And once a model is built, the result can be collected from `getResult` operation.

Despite the API for data model creation is unified, there is a different strategy used for building each of the data models, based on abstraction level. Therefore, when processing a PowerDesigner file Metadata Extractor must choose the suitable implementation of `DataModelBuilder` accordingly. The class `BuilderFactory` is used to choose the correct builder via its factory method which implementation - `PhysicalDataModelBuilder` or `LogicalDataModelBuilder` or `ConceptualDataModelBuilder` by the abstraction level of the data model file that is about to be resolved.

The outline of how the builders work is proposed in their common base class `AbstractDataModelBuilder`. Also, the functionality independent of the type of a specific data model is written here, the way to do it is to look on objects from the perspective of abstract classes that are providing the general features needed for the process of data model creation.

The specific implementations take advantage of the fact that forming a tree-like structure from an XML format is natural and straightforward. That is why, for now, Metadata Extractor omits a general view on the construction of a data model. Ignoring the convenient structure of the XML tree would hide the context provided straightforwardly by DOM nodes. Requests for specific DOM nodes

during the process of object creation are done using XPath queries on the fully loaded XML document.

5.2.4 Reader

The Reader component interconnects the resolver with the parser. Input for this component is a directory, where the reader searches for files containing saved data models. It collects a batch of files from the input directory, loads them using the two components, and sends the batch in the processed form of a set of `DataModel`s to the Data Flow Generator unit.

Input directory containing PowerDesigner files to process is the input for the reader's most important class `PowerDesignerXmlComponentReader`. The reader recursively discovers the directory and collects the file that will have to be resolved using the `collectFiles` method. When a data model file is found, its dependencies are checked using a `SAXParser` with a simple handler class `TargetSAXHandler`, which is obtaining paths to related models from an XML. The found dependent files, however, must be in the input directory, so if it is not the case, the reader tries to resolve their paths and check if there is no file with matching sub-path within the input directory. If it is a bidirectional edge between the files is created. Once all the files from the input are collected, component creation takes place so the input set is split into smaller logically connected groups.

When the main API method of a reader - `read` is called, the parser and resolver start processing one component. Result of their work is a set of related reconstructed `DataModel`s which is the return value from the method. That means the reading method is stateful and `canRead` checks if there are any component left to be returned.

5.2.5 Data Flow Generator

The purpose of this component is to create a correct graph representation for the objects described in the model component. Data Flow Generator has to collect information to identify databases that physical models resemble.

When translating a set of reconstructed data models and their underlying objects, a suitable instance compliant with the `GraphBuilder` interface must be chosen based on the level of abstraction for each data model in the input. Metadata Extractor does it using the `GraphBuilderFactory` class. Then the tree of data model objects is traversed.

When dealing with a physical data model, an instance of `PhysicalGraphBuilder` is used, underlying `DatabaseConnector` searches for the modeled database objects. A suitable `Connection` realization corresponding to a physical data model is made of the information from a data model and .ini file, as loading connection details from .dsn and .dcp files is not supported yet. A user has to specify in the .ini file type, server and name of a database for a physical data model, in order to identify the corresponding database successfully.

On the other hand, physical and logical data models use the generic `ModeledGraphBuilder` class for creating nodes representing objects obtained from data models of higher abstraction.

Lastly, during the phases of translation objects to nodes, Metadata Extractor collects by IDs the nodes that have a mapped counterpart. Once all the data models in the processed components are fully built, the generator can resolve the mapping, as both ends of the MAPS_TO edge must be present. Then Metadata Extractor simply looks at to what ids are objects mapped, obtain an actual representation of these ids put them into the relationship by creating the edge.

5.3 Extensibility - Modeling Common

For the sake of general information retrieval from modeling tools, we developed a unit where a component for metadata extraction from an arbitrary modeling tool meets Manta Flow. This module, where the common bridging logic is stored, we call modeling common.

Physical data models need to be paired with corresponding connections to DBMS in order to identify the database they resemble. Only with this knowledge, Metadata Extractor can correctly pair the same database objects that appear both in a physical model and in a database dictionary extracted by Manta Flow from a live database. This functionality is provided by implementations of the interface `DatabaseConnector`. Details about the target database of a physical data model are kept in a realization of the `Connection` contract. The API to database dictionaries created by Manta is provided by the `DataflowQueryService` interface. Having collected information of a physical object, either a column or a table, the query service tries to find the object in the dictionaries using `createColumnNode` or `createTableNode`. If the service succeeds, matched node from a dictionary is appended to the output graph, otherwise null is returned from a `create*` method and Metadata Extractor tool copes with the situation in such a way that it constructs a node that has no backing in a live database, marking the node's attribute source type to MODEL to make clear in the output, that it is an artificial node based only on a physical data model. It can be from two reasons - the database object does not exist or we provided inaccurate/incomplete data to identify it.

So the first requirement for a general extractor from data modeling tools is to be able to organize extracted metadata in such a way that `DatabaseConnector` may be called and ideally matches the modeled objects with the extracted ones. This is a very important step, as this pairing is the prerequisite for business data lineage creation. If the physical modeled objects do not match the ones from a database that take part in data lineage, there is no flow to be propagated to higher levels of abstraction via MAPS_TO edges and the result of interpolation will be an empty set of edges.

Secondly, we have defined a common way of creating nodes representing objects extracted from data models of higher abstraction than the physical ones - conceptual and logical. We define a set of methods, where each of them is responsible for the creation of node representation of a type of object that may appear in the data models. Most commonly, a modeling tool will support only a subset of the object types the node creator is able to construct. However, that is not a problem as the node hierarchy is not that strict and it can be built to be compliant with the layout of a specific modeling tool. For example, it is not necessary to build an attribute under an entity node if a modeling tool does not

allow such a concept, just like an owner node is not necessary, as there is no strict rule if the tools should support it or not.

We have discussed the modeled objects on the conceptual and logical level are described by ER or EER diagrams. The important fact is that the possible kinds of objects are going to be very similar in both models, that is why the single interface `DataModelNodeCreator` for high-level node creation may be used. However, it must be implemented by two different classes `LogicalDataModelNodeCreatorImpl` and `ConceptualDataModelNodeCreatorImpl`, so details about node's type can be specified, therefore the nodes will not mix between the layers. A realization of the `Resource` interface differentiates technologies, in our case it makes sure objects extracted from modeling tools are aggregated by what specific tool is their origin.

To draw an object as a node and attach its metadata to a node, such as definition or comment, the `NodeMetadata` interface should be implemented by the object that is being transformed into a node representation. The contract forces the implementers to define a name for the node and to expose the metadata to be attached to the node representing them in the output in the form of strings. More precisely, the objects have to override the `getName` method and to fill a `Map<String, String>` data structure where a key is the name of a property. The metadata accessible in the map are shown when presenting the output graph and further describes the extracted objects.

To create a guideline what to do when implementing a metadata extraction from a different modeling tool is the following:

1. Define the crucial data model objects, relations between them, hierarchy and properties interesting enough to be captured in data lineage by read-only interfaces. That is the model component. The objects that will be represented as standalone nodes in data lineage visualization at the end have to implement the `NodeMetadata` interface.

2. Depending where from the data models are obtained, this phase generally retrieves data models from storage. It may be by calling a web API to get data mode objects from a server, retrieving them from a database, or like in the case of Metadata Extractor, by parsing a file.

3. Resolving takes place once a representation of data models was obtained. This stage transforms the acquired structures into objects conforming the aspects described by the model module.

4. Data flow is generated from the output of Resolver. This module translates data structures describing modeled objects to graph nodes. Database objects from physical data models are expected to match the ones in database dictionaries of Manta that were extracted directly from databases. The communication with Manta is done via `DatabaseConnector` interface and a concrete `Connection` that is assigned to a physical model. Logical and physical modeled objects are transformed to node representation using the corresponding implementation of the `DataModelNodeCreator` interface - logical and physical respectively. Once the nodes are created out of the

extracted objects, the resolving of mappings between the layers takes place. Linking correctly physical nodes that were paired with the ones from the database dictionary, with higher level nodes is crucial to bring business lineage. However, there is no strict skeleton for this process as the approaches to how modeling tool realizes mapping across data models may be substantially different, as we have seen in case of PowerDesigner and ER/Studio.

5.4 Error Handling

We require Metadata Extractor to be as stable and as robust as possible. When an unexpected event occurs, the program should not fail on an exception and stop computing. Metadata Extractor should ideally print a message describing the given event to log using the SLF4J logger with level `error` or `warn` depending on the severity.

If there is something wrong in a data model definition, Metadata Extractor tries to get as much information as possible. For example, if a part of an XML tree describing one PowerDesigner table is malformed, partial information about the table should be extracted so the program tries to retrieve it. In case the table is damaged irrecoverably, it should be skipped and continue with the processing of other tables directly. One way or another, one wrong table must not affect any other table in the given file.

5.5 Technologies

The quality of the programmer's toolbox may have a huge influence on his productivity when developing a more complex piece of software.

- Java 8
  To be able to integrate Metadata Extractor into the ecosystem of Manta Flow the best solution is to use Java as the APIs of the data lineage software are written in this programming language.

- Maven
  For dependency management, the most usual tool when programming in Java is used - Maven. In order to build the program, all the Manta Flow artifacts which Metadata Extractor depends on must be obtained successfully. However, those artifacts are reachable only within Manta's private network.

- Spring
  For program configuration, inversion of control via Spring's XML files is used. Along with .properties, where variables used in the spring files may be (re)defined.

- JUnit 4
  The standard for unit testing Java programs is the JUnit framework we use.

5.6 Testing

Automated unit tests validating the crucial parts of the implementation makes much easier to determine the correctness of Metadata Extractor's logic. We developed test suites that should tell in what shape is the program.

ER/Studio Parser

The correctness of the parsing process is checked by the class `ErStudioFileFormatParserTest`, which is controlling how many CSV tables from a parsed .DM1 files were obtained. Given that the format consists of a constant number of tables if the count is mismatching it indicates a flaw in detecting borders of tables. Also, every CSVs should be valid in the way that each row in a table must have the same number of fields. If it has not, we assume there is a problem on our side and parser does not pass the tests.

Reader & Resolver

The testing output of the Reader module shows a lot about Resolver's correctness since the logic of the reader is actually provided by this module. By the nature of Reader - it connects Resolver and Parser, these tests rely on the fact that parser works correctly and we assume this is true as far as the parser module tests are passing. The goal of Reader tests is to find out whether all objects from input data models were successfully and correctly loaded into data structures described by the model component. In the case of ER/Studio, the testing compares expected statistics about data models such as the number of physical models in a solution, entity count of the logical model, etc. with the number of objects that were loaded actually by the reader. Secondly, some specific objects from the hierarchy of a data model that is on input are picked and checked if they were loaded accordingly. Tests check metadata of those objects and mappings with other objects as well.

PowerDesigner must create components correctly, so we have a test for checking if the number of components is correct and if dependencies on files with obsolete path can be fixed. The main Reader test consists of reading file by file, thus the size of a component is one every time (the components creation correctness is ensured the previous test). The expected values and objects are loaded from JSON sources file where they are stored in a deserialized form.

Data Flow Generator

In Data Flow Generator module, we are interested if objects loaded by the reader are transformed into nodes as expected. The skeleton is similar for both modeling tools.

`*GraphBuilderTest` classes are testing independently drawing of each of the layers. The graph created by the `*GraphBuilder` classes can be saved as a .txt file describing the graph. For each test, we have an expected output of how a

correct output graph should look like. The two files are compared if they are the same the behavior of the output building process is correct.

The previous test checked the representation of the layers independently, whereas in the test classes `*DataflowTaskTest` all layers from a solution, or from a component, when talking about PowerDesigner, are constructed with mapping edges leading between them. Outputting graph is also serialized to a textual file. Next, mapping edges are filtered from the file and compared with the expected set of maps_to edges.

Conclusion

We have developed Metadata Extractor, a software system capable of extracting metadata from two data modeling tools - SAP PowerDesigner and ER/Studio Data Architect. The part of Metadata Extractor coping with ER/Studio models was already successfully released in the Manta Flow product. To achieve this, a thorough analysis of the tools was required, in order to identify features relevant for data lineage. As well as studying the way data models are represented in memory.

A framework bridging Manta Flow with objects extracted from data models was developed, therefore further support of other modeling tools may take advantage of it and can be aware of what contracts to meet in order to create business lineage and to enrich the physical one.

One of our aims was to explore the possibilities of modeling data lineage in modeling tools. We went through it and described the important aspects of such lineage and compared it to the one that Manta Flow creates. Initially, there was an idea that if a tool allows the functionality, it would be nice to compare the actual lineage, physical or business, computed by Manta Flow and Metadata Extractor, with the modeled one. However, the modeling tools we analyzed do not provide any API that would allow us to correct or compare the flows specified in the tools. An option would be to try to manually adjust the lineage, but this approach would be not easy and fragile at the same time.

Bibliography

[1] Facebook blames 'database overload' for most severe outage in its history. https://www.dailymail.co.uk/news/article-6806775/Facebook-Instagram-running-following-EIGHT-HOUR-outage-affecting-users-worldwide.html. Accessed: 2019-03-27.

[2] HealthCare.gov's Heart Beats For NoSQL. https://blogs.wsj.com/cio/2013/12/03/healthcare-govs-heart-beats-for-nosql/. Accessed: 2019-05-15.

[3] PostgreSQL in Mission-Critical Financial Systems. https://www.pgcon.org/2010/schedule/events/204.en.html. Accessed: 2019-05-15.

[4] Right of Access. https://gdpr-info.eu/issues/right-of-access/. Accessed: 2019-05-15.

[5] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems (7th Edition)*. Pearson, 2015.

[6] DB-Engines Ranking. https://db-engines.com/en/ranking. Accessed: 2019-05-15.

[7] NoSQL Data Modeling. https://www.ebayinc.com/stories/blogs/tech/nosql-data-modeling/. Accessed: 2019-05-15.

[8] SPARC-DBMS Study Group. Interim Report: ANSI/x3/SPARC Study Group on Data Base Management Systems, 1975.

[9] Peter Pin shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1976.

[10] Evaggelia Pitoura. *Access Path*. Springer US, Boston, MA, 2009.

[11] Abraham Silberschatz Professor, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Education, 2010.

[12] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *IBM Research Report, San Jose, California*, RJ599, 1969.

[13] MANTA + Teradata — Data Lineage Automation Solution. https://www.youtube.com/watch?v=jbVoMgXvJD4. Accessed: 2019-05-15.

[14] B.B. Agarwal. *Software Engineering And Testing: An Introduction (Computer Science)*. Jones & Bartlett Publishers, 2009.

[15] ER/Studio Data Architect. https://www.idera.com/er-studio-data-architect-software/overview. Accessed: 2019-05-07.

[16] History, Editions and Licensing Models. https://www.powerdesigner.biz/EN/powerdesigner/powerdesigner-licensing-history.php. Accessed: 2019-05-07.

[17] PowerDesigner Main Features. https://www.powerdesigner.biz/EN/powerdesigner/powerdesigner-features.html#InfoArchitecture. Accessed: 2019-05-15.

[18] Bussiness Assets. https://university.collibra.com/knowledge/collibra-body-of-knowledge/data-governance-operating-model/structural-concepts/asset-types/business-assets/. Accessed: 2019-05-15.

[19] Lineage vs Traceability Understanding the Differences. https://www.collibra.com/blog/lineage-vs-traceability-understanding-the-differences/. Accessed: 2019-05-15.

[20] Business glossary. https://www.informatica.com/content/dam/informatica-com/global/amer/us/collateral/data-sheet/business-glossary_data-sheet_6922.pdf. Accessed: 2019-05-15.

[21] Data lineage and business lineage reports. https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_11.3.0/com.ibm.swg.im.iis.mdwb.doc/topics/c_analysisReports.html. Accessed: 2019-05-15.

[22] ERSTUDIOBUS Release Notes. https://isp.asg.com/Rochade/ERStudio/V2.3x/ERSTUDIOBUS_237_001_Release%20Notes.pdf. Accessed: 2019-05-15.

[23] Common Format and MIME Type for Comma-Separated Values (CSV) Files. https://tools.ietf.org/html/rfc4180. Accessed: 2019-05-15.

[24] SAP PowerDesigner Documentation. https://help.sap.com/saphelp_pd1654_data/helpdata/en/c7/c304506e1b101483efbf3d87db9a8f/frameset.htm. Accessed: 2019-05-15.

List of Figures

List of Abbreviations

ER	Entity-Relationship
EER	Enhanced-Entity-Relationship
RE	Reverse-Engineering
CDM	Conceptual Data Model
LDM	Logical Data Model
PDM	Physical Data Model

A. Attachments

A.1 User Documentation

To run Metadata Extractor and RE tools the prerequisite is to have Java 8 on a computer. The project management process of our software is ensured by Apache Maven.

To build and consequently run the RE tool, the following is required:

1. Navigate to the directory dm1-reverse-engineering-tool.
2. Run the "mvn clean compile assembly:single" command producing the executable dm1-reverse-engineering-tool-1.0-SNAPSHOT-jar-with-dependencies.jar file.
3. To start execution on, for example, model.dm1 file (sample models can be found in resource folder) the .jar with "java -jar .jar model.dm1". SQL script called MetaModel.sql is produced.
4. Let modeling tool create a model out of the script.

Metadata Extractor, in fact, consists of five artifacts: manta-connector-erstudio-aggregation, manta-dataflow-generator-erstudio-aggregation, manta-connector-powerdesigner-aggregation, manta-dataflow-generator-powerdesigner-aggregation, and manta-dataflow-generator-modeling-common.
To build each of them, this must be done:

1. Navigate to the top-level folder of an artifact.
2. Run the "mvn clean install".

However, in order to build and run Metadata Extractor itself, inaccessible dependencies owned by MANTA are needed, they are stored at MANTA's private repository. Otherwise, the maven projects cannot be built successfully. Also, the functionality of Metadata Extractor is subject to Manta Flow. The data lineage tool is what interconnects the artifacts. Not only between themselves, but also puts them together with other crucial parts of its ecosystem as database extractors, transformation analyzers, and post-processing on Manta server. The interpolation and visualization of acquired data lineage are part of the server's functionality as well. That is the reason why a user is not able to run the install command successfully unless he has access to the maven repository of MANTA.

We will not discuss running Metadata Extractor as Manta Flow is a proprietary software and its products require a valid license. The command line interface which allows a user to run Metadata Extractor, as well as the rest of Manta Flow's features, is not attached to this work.

Nevertheless, the source code of Metadata Extractor is fully accessible.

A.2 Cooperation with Manta Flow

In the case a user has access to Manta Flow server and command line interface (CLI) with corresponding licenses, the full functionality of Metadata Extractor can be manifested.

The CLI has an aggregate input directory, which is further divided by technologies. Subdirectories "erstudio" and "powerdesigner" are located in the input folder. Both have the same following layout: They contain input systems, such a system is a logical group of modeling files to be processed. A system consists of two items - one is a folder containing the models from which metadata will get extracted, the second one is a .ini file, where connection details paired with physical models are defined.

Each system of models that has to be processed is specified by its .properties file in manta-flow-cli/scenarios/etc/erstudio, manta-flow-cli/scenarios/etc/powerdesigner respectively. For each input system, a .properties file specifying the name of the system folder to be handled by Metadata Extractor must be present. Templates of such property files can be found at the locations.

To proceed with the extraction, a user runs one of the executable files provided _run.sh for Unix-like, _run.bat for Windows machines respectively. In order to achieve the full functionality of Metadata Extractor, it is highly recommended to let analyze databases corresponding to captured in physical models by Manta Flow as well.

1. Create a system of files in input. Insert the files to process and .ini file with connections.
2. Create a .property file with name of the system folder, place it to scenario scenario directory of the corresponding technology.
3. Set up other features of Manta Flow (optional but recommended).
4. Execute _run.sh, _run.bat respectively.
5. Preview the extracted graph with computed data lineage.

A.3 Contents of the Attached CD

```
root
├── readme.txt
├── impl
│   ├── metadata-extractor
│   ├── dm1-reverse-engineering-tool
│   │   └── meta-model.dm1
└── text
    └── thesis.pdf
```

A.4 Full List of Modeling Tools Metadata

Conceptual & Logical Data Model

ER/Studio

CDM

ER/Studio does not support conceptual data models.

LDM

- *Owner*
- *Entity*
  - Name
  - Attributes
  - Definition
  - Note
  - Where Used
  - User-Defined Mappings
  - Owner

    Properties We Will Not Extract

  - Permissions
  - Keys
  - Relationships
  - Constraints
  - Naming Standards
  - Data Lineage
  - Security Information
  - Attachment Bindings
- *Attribute*
  - Name
  - Definition
  - Notes
  - Where Used
  - User-Defined Mappings

Properties We Will Not Extract

- Datatype
- Default
- Rule/Constraint
- Reference Values
- Naming Standards
- Compare Options
- Data Lineage
- Security Information
- Attachment Bindings
- Data Movement Rules

Objects We Will Not Extract

- *Relationship*

PowerDesigner

Conceptual and logical data models in PowerDesigner have so much in common that we will propose unified view on what may be stored in them. The properties/object that are specific for either of them are marked with information in brackets saying "CDM/LDM only".

CDM & LDM

- *Data Item* (CDM only)
  - Name
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords

Properties We Will Not Extract

  - Data type
  - Length
  - Precision
  - Domain
  - Stereotype
- *Entity*

  - Name
  - Attributes
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Parent Entity
  - Dependencies

    Properties We Will Not Extract

  - Number
  - Generate
  - Identifiers
  - Rules
  - Stereotype

- *Attribute*
  - Name
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Parent Entity
  - Dependencies

    Properties We Will Not Extract

  - Data type
  - Length
  - Precision
  - Domain
  - Primary Identifier
  - Displayed
  - Mandatory
  - Foreign identifier (LDM only)
  - Standard Checks

  - Additional Checks
  - Rules
  - Stereotype
- *Inheritance*
  - Parent Entity
  - Child Entity

Objects We Will Not Extract

- *Relationship*
- *Identifier*
- *Association and Association Link* (CDM only)
- *Domain*

Physical Data Model

ER/Studio

- Type of Data Model (DBMS technology)
- *Schema*
  - Name
  - Tables
- *Table*
  - Name
  - Columns
  - Schema
  - Definition
  - Note
  - Where Used
  - User-Defined Mappings

Properties We Will Not Extract

  - Storage
  - Dimensions
  - Properties
  - DDL
  - Indexes

  - Foreign Keys
  - Partition Columns
  - Distribute Columns
  - Distribution
  - Organization
  - Partitions
  - Overflow
  - Constraints
  - Dependencies
  - Capacity Planning
  - Permissions
  - PreSQL & Post SQL
  - Naming Standards
  - Compare Options
  - Data Lineage
  - Security Information
  - Attachment Bindings

- *Column*
  - Name
  - Definition
  - Notes
  - Where Used
  - User-Defined Mappings

    Properties We Will Not Extract

  - Datatype
  - Default
  - Reference Values
  - Naming Standards
  - Compare Options
  - LOB Storage
  - Data Lineage
  - Security Information
  - Attachment Bindings
  - Data Movement Rules

Objects We Will Not Extract

- *View*

PowerDesigner

- *Table*
  - Name
  - Columns
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Schema
  - Dependencies

Properties We Will Not Extract

  - Number
  - Generate
  - Dimensional type
  - Type
  - Indexes
  - Keys
  - Triggers
  - Procedures
  - Check
  - Physical Options
  - Preview
  - Lifecycle
  - Stereotype
- *Column*
  - Name
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Table
  - Dependencies

Properties We Will Not Extract

  - Data type
  - Length
  - Precision
  - Domain
  - Primary Key
  - Foreign Key
  - Sequence
  - Displayed
  - With default
  - Mandatory
  - Identity
  - Computed
  - Column fill parameters
  - Profile
  - Computed Expression
  - Standard Checks
  - Additional Checks
  - Rules
  - Stereotype

- *User, Group, and Role*

Objects We Will Not Extract

- Primary, Alternate, and Foreign Keys
- Indexes
- Views
- Triggers
- Stored Procedures and Functions
- Synonyms
- Defaults
- Domains
- Sequences
- Abstract Data Types

- References
- View References
- Business Rules
- Lifecycles