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:

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 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 are extracted from data modeling tools, such as ER/Studio and PowerDesigner, that are widely used in database design process. The solution extends the Manta Flow lineage tool, while taking advantage of its features at the same time.

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 a 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 large amount of patient's data is a 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 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 since 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. In that moment, data modeling comes into play. Instead of a transcript of the conversation, a better solution is to translate the debate into more intuitive and standardized piece of documentation, that means into a conceptual data model. Once the initial model is created, the next steps are going more and more toward a 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, a 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. At 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 commonly increase price when there is 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 database representing the not-yet-sold tickets accordingly, so that valid information is shown to customers. The logic takes places thanks to SQL queries applied on a database. As the amount of business processes grows, the ability to justify 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 for the struggles with being overwhelmed by the complexity of a big data solution. It brings an 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 less profound technical background, that would want to make decisions based on how data flows 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 daily basis. We assume big companies approach database development responsibly, thus there exists a documentation of their systems in form of data models. We will reuse the models for creating the

business perspective on 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 that stores 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 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. link with the goals

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 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 will 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 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 implicit meaning as stated in literature [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 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 how data moves 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 of seeing 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 will go through the concepts fundamental for understanding the domain to which Metadata Extractor contributes. The prerequisite 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 will 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 Extract. We also preview in the chapter programs solving similar problems as we do.

Lastly, Chapter 5, Implementation, is discussing the resulting classes and their responsibilities which Metadata Extractor is made of.

2. On Database Background

In this chapter we will introduce terms and concepts that are help us understand the domain to which Metadata Extractor contributes. To understand requirements, design and need for the 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 purpose of a 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 database implementations and what to take into account when comparing database technologies. That may be helpful when choosing the best suitable option for some specific data set to store or to see how storing of great amount of structured information can be approached.

The basic division of databases 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 terminology related databases.

Relational Databases

A *relational database* is a set of tables. A table consists of rows (also records) and columns. We can see such table as a class whose attributes are represented by columns and instances by rows. The important aspect is that relational tables carry both data that need to be saved by user and the relationships between the stored data as well. To store 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 the table and is used when referring to the record.
A *foreign key* is a key that uniquely identifies a record from a table (the reference 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 order. [1]

---

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

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

The main reason behind this is the fact that NoSQL databases have a flexible schema or are schema-less (there is no point in determining a database schema when data types of attributes or keys are not defined), 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 established as a standard. Also data models bringing a greater level of abstraction are not used in this field. [7]

The fact to consider is that once a database is of relational kind, it is more or less know what to expect from it. The hierarchy of objects in these RDBMS has 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 technologies. An equivalent tool aimed for some non-relational data model would be effective for a narrower set of technologies as No-SQL databases are divided 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. 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 as the way data are organized in them is substantially different.

Lastly, despite the non-relational databases may be growing in numbers and became a serious alternative, as it suits some use-cases better, 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 who is using a query language for accessing data. However, third party or application programs also need to access the DBMS. In our work two types of programs will be connecting to databases when fetching metadata - modeling tools when undergoing a reverse-engineering process and Manta Flow. The way applications uses 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 a textual information used to identify a data source and establish a connection with it. It is made of pairs of keywords and values separated by semicolon, where the keywords are parameters of the 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. Every one will agree that no construction work can go without solid design and documentation. It would sound a bit

strange to hire construction workers straight ahead and tell them that 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 we will agree that expectations and requirements of the later inhabitant could not be met properly. Surely there are good reasons why the usual steps are followed strictly. Let us move on from this analogy to the database domain.

When deploying a database from scratch one may think of two short term advantages. Firstly, the time needed to have data stored somewhere would be much shorter and secondly the initial cost of the system could 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 one 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 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 tries to understand the system, he can choose a data model on an appropriate level of abstraction that will introduce him 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 properties of the data that they need or have, and will be stored. These information are 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 will discuss data models from two basic perspectives. The vertical division talks about on what levels of abstraction data models can be designed. 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 Division

American National Standards Institute [8] came with a database structure called three-schema architecture. 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, change of the implementation that is tied with 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 finished product, it does not say anything about the design process that leads 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 [9] identified four levels of view of data, 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 have undergone some modifications, as for example the first level is today omitted, to the one that is recognized nowadays. The differentiation takes into account what 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 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 and 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 is that the path is specific for a DBMS technology.

- Logical Data Models (implementation, representational)
  In the middle between the two other model types there are representational data models, which on the one hand are comprehensible by end-users and on the other hand are not too abstract so that they can be used as documentation for an actual database implementation of the modeled data.

- Physical Level Data Models(low-level)
  In contrast to conceptual models, the physical ones are tied with how data are stored physically at storage media and showing all the specific internal details that may be overwhelming unless the reader is not a computer specialist.

Horizontal Division

We will 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. A reader should be already familiar with the concepts used in the models from the sections describing terminology of SQL databases. All the terms such as 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 for the four level architecture[9] that covers the highest two levels and may be a basis for unified view of data.
It was an opposition to the three major data models that were used - relational, network and entity set model. His 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 concept sub-classes and super-classes, known as inheritance or is-a relationship, between entities was brought.

Summary

To put the two perspectives together - conceptual and logical data models are usually represented by ER data models. As the most low-level model type is tied directly with how a database is organized, physical models must obey the structure of a database. Therefore, as we work with relational databases, we will assume physical data models to be of relational kind.

Now we will describe the standard three levels of data models in more detail.

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 a software used to manage the data, DBMS, data types, etc.

- It is not normalized.

   Object Types

- A real world object is captured by an *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 full view of the section of the world that a data model resembles.

We illustrate the concepts used in conceptual 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 an information to store when describing 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 captured using relationships.

---

[3]Definitions varies and in some literature 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.
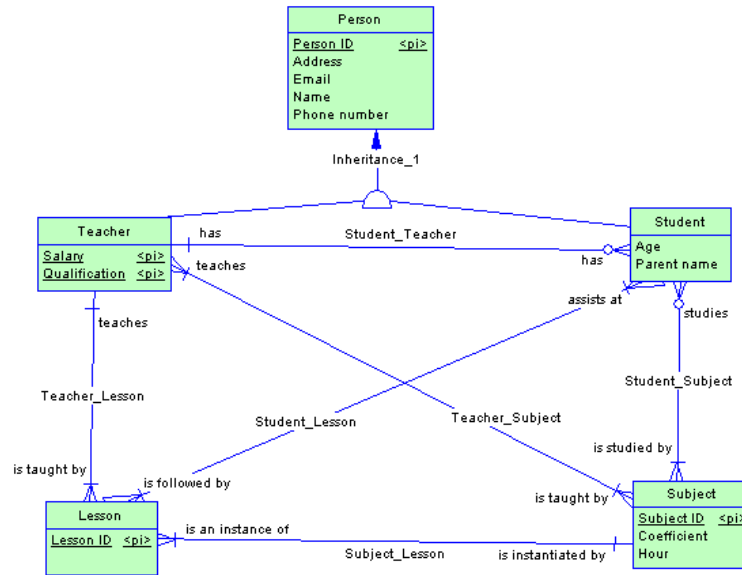
Figure 2.1: The figure shows a diagram of a conceptual data model created in PowerDesigner, which is using EER data model. There are 5 entities, each of them has at least one attributes. Teacher and student entities inherit from the person entity. Blue lines represent relationship between entities.

2.2.3 Logical Data Model

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

The Main Characteristics Include

- It is independent of a software used to manage the data or DBMS.

- Each entity has the 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 layer do not represent solely real world objects, in Figure 2.2 we can see entities were creating by 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.

13

- *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 logical data model can be seen in Figure 2.2. It describes the same system as Figure 2.1.
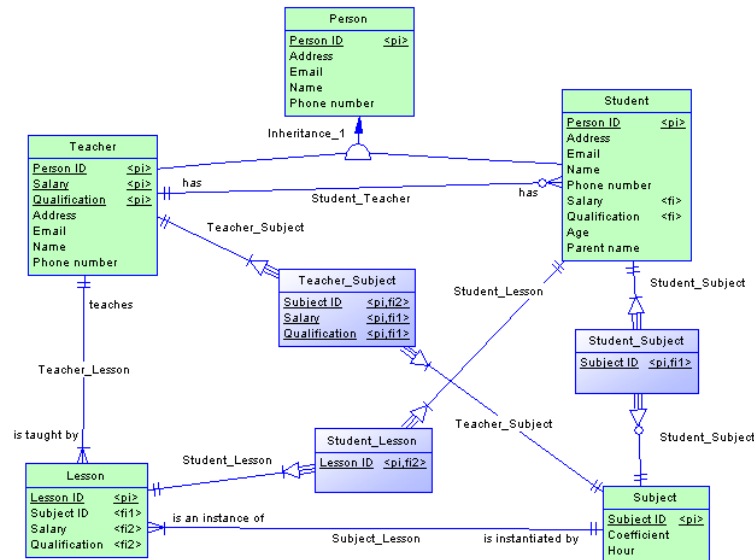
Figure 2.2: The figure shows a diagram of a logical data model in PowerDesigner. The main difference in contrast to Figure 2.1 is that many to many relationships are deconstructed to entities.

2.2.4 Physical Data Model

A physical data model is a description of a database implementation, so it is necessarily tied with one specific database technology as it should have one-to-one mapping to the actual implementation. Its main message is to communicate how the data are stored.

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

The Main Characteristics Include

- Exact data types (DBMS specific) and default values of columns are outlined in the model.

- DBMS's 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 corresponds to logical entities.

- A *schema* is basically container for tables that logically groups them. Database users have usually schemas assigned to them and can access only the tables contained in those schemas[4]. Not every DBMS supports this concept, though.

- *Columns* should represent logical attributes in memory.
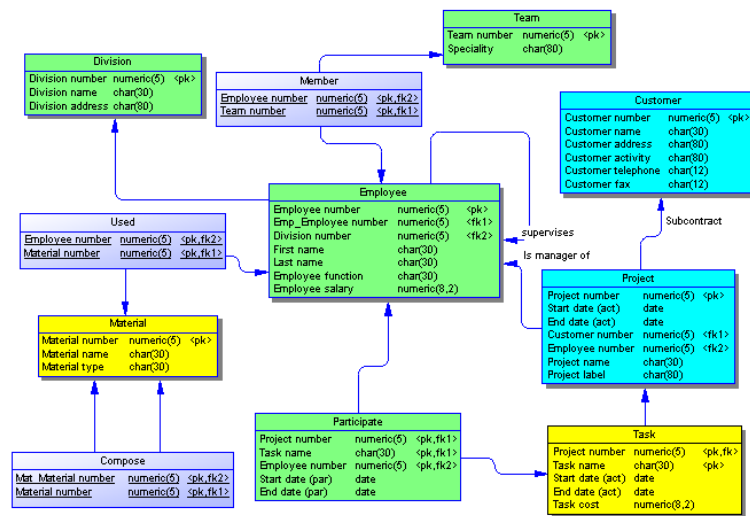
Figure 2.3: The figure contains a diagram of a physical data model with 11 entities. We can see specific data types of columns. Relations are pictured based on key columns.

2.2.5 Relations Between the Models

We described what the role of each of the layers in a database design process is. Now we will 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 divisions, we should think about how database design can proceed. The basic approach is the *top-down approach* to database modeling. It is natural to start with a general idea what information should a database store and what are the relations between stored objects. An end-user defines this high-level logic, and as time goes, 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 the full view of a database is the *bottom-up approach.* It can be harder to imagine what would be use-cases for this approach, but there are some problems that are bottom-up in nature. A nice real world example

---

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

of the bottom-up strategy is how physicians work. They start with "low-level" details such as symptoms and they're trying to build the whole image of patient's condition. So in the field of software, 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 objects that are semantically equivalent in different data model types vertically. Sometimes even mapping between objects on the same abstraction layer are allowed, but we will not consider this, as it be mixes two different concepts together - data modeling with data lineage. To be more precise, what we mean by the semantically equivalent objects is, that we will 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 these mapping links is extremely useful when a person wants to gain an overall overview of the system and to comprehend it. For example, when a user sees a data table in 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 leads to higher layers providing greater abstraction over the implementation, uncovering the motivation why the table was created. What is the reason for the table being in a database should be much clearer then. It is worth mentioning that usually, the mapping relations between objects of different layers are not a simple one-to-one relationships, but also the cardinalities may vary greatly. One logical attribute may be commonly realized via multiple database columns. Normally, more technical models are composed of bigger count of objects so one conceptual entity may be realized by multiple database tables in the end. Generally, it is assumed that number of conceptual objects is smaller than the number of logical objects, which is smaller than the number of physical objects. It is natural that when capturing important high-level aims less entities is 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 for 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 the system consists of. Although 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 taking decisions about the infrastructure, since understanding the consequences of the actions should be more clear. Also it 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 affected data originates. 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 the visualization as a graph. An example of data lineage graph is demonstrated in Figure 2.4.
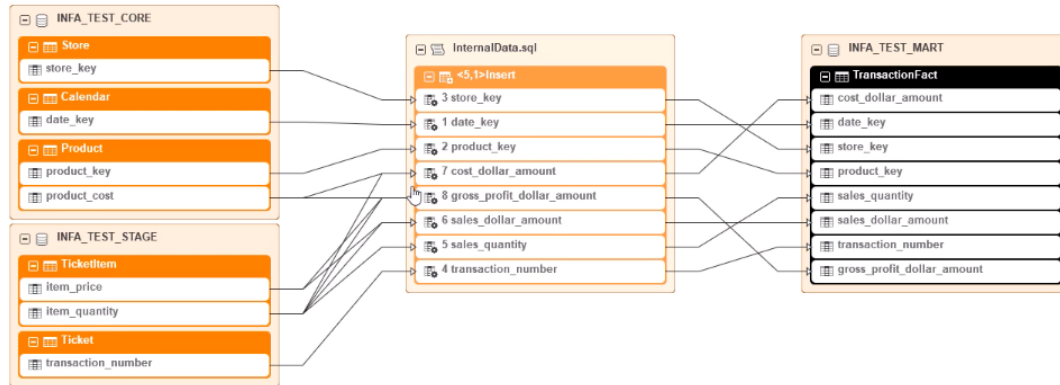
Figure 2.4: Data lineage as it is visualized in Manta Flow user interface. The simple example illustrates how the TransactionFact database table is filled with data originating in five different tables via InternalData.sql transformation that inserts the data from sources to the target. The directed 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 movement of data towards the destination, *backward data lineage* creates 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.

Other differentiation of data lineage is the business one versus the technical one. *Business data lineage* highlights only transformations and aggregation of data in a simplified way to the target business user, whereas *technical data*

---

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

lineage displays precisely 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.
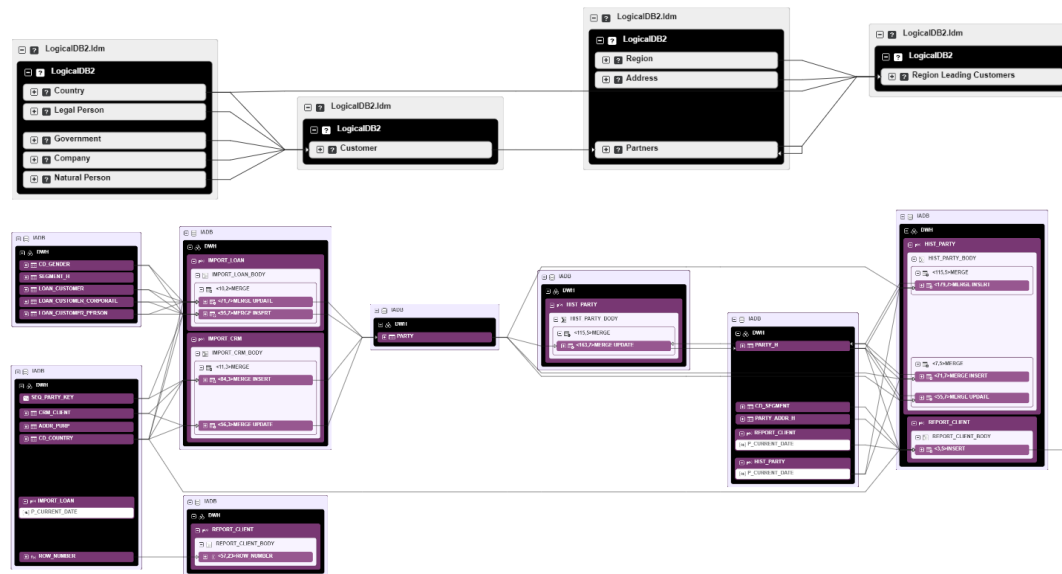
Figure 2.5: The two visualization of data lineage show the very same database system from two different perspectives. The business lineage hides transformations and speaks natural language, using concepts that appear in real world providing cleat motivation of the business process. Whereas technical data lineage precisely illustrates what happens with data in a database management system.

describe the mappings visually or verbally?

Now we will focus on how data lineage can be created to describe 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* are the data describing other data. The metadata we will use when analyzing a database are include database name, names of tables, columns in tables, names of columns, procedures, data types, etc. When we have these information describing all the records that can be stored in the database, together with all SQL scripts which are used for management of the database, we can reliably determine how the data flows 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 SQL queries that modify data are examined. They are stored in .sql files and usually a node is added for each of the files. Then the data lineage creation tools, like Manta Flow, identifies what tables and columns are the sources of input data for queries and where outputs of the 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 with the query in such manner that every edges 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, and Java. Uniqueness of the software product is in its capability of handling code that is hardly readable by human. Thanks to this feature, Manta Flow can automatically process databases consisting of millions of records and create a map of data flow across 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 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 currently Manta Flow is able to scan, these are the supported relational database types it can handle. That means when physical models are aimed on one of the following database systems, we can create 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

2.3.2 Data Lineage in Modeling Tools

It is quite common that modeling tools provide some kind of view 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 logical (or conceptual) data lineage. The reason why not to take into account this feature is, that it may be completely away from how the system really works and how the 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 the tools is that a user draws the lineage graph by hand. It may be useful at the time when the database is not yet implemented and there is a type of dependency relationship that cannot be captured other way. But once a database is running, the lineage may get misleading, as there is no way to enforce correctness of the data flows specified. 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 objects 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 will assume using of *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 user interface of a CASE tools from scratch dragging a 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) resemble a live database can be transformed into 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 whose aim is to find out principles of how things are done or works in a system that is already running and try to gain deeper understanding of the system. Applied to our domain, the reverse engineering approach to 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 systems 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 process as well. The model is an exact image of the database and one-to-one mapping between the model and database should be secured.

3.1.3 Generating

Given a data model on some level, another model on different abstraction level 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 logical model should be created based on it, it is really helpful not to start from a scratch. Instead an outline of the logical model can be created by generating from a corresponding conceptual data model. Then it can be reshaped into the the desired condition more quickly. Generation sources and targets are in maps-to relationship implicitly.

3.1.4 Importing

Finally a CASE modeling tool may be able to import data models that were created using a 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 with possibility of forward and reverse-engineering the models. Tens of database platforms can be targeted by a physical model.

Logical model is realized by an entity-relationship data model, whereas physical models are of relational type.

The extension of the product, ER/Studio Data Architect Professional, comes with model repository which 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 at 16.6 [16].

PowerDesigner provides range of various modeling techniques such as, application UML modeling, business process description, enterprise architecture, data movements models and, most importantly, data modeling where conceptual, logical and physical data models are supported and physical models are compliant with more than 60 database management systems. Forward and reverse-engineering of the models can be done. The first two levels are of extended-

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

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

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 will 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 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 tool that we will need to focus on, when finally implementing Metadata Extractor.

In this chapter we seek answers for 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 the modeling tools are capable of creating data models. These models are saved into files. The output files of the modeling tools are going to be the input of Metadata Extractor, that uses them for reconstruction of the objects and information contained in the data models. In order to come with logic of the reconstruction, we need 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 may belong to a schema.

These are the fundamental metadata Metadata Extractor must load form the files. Consequently, it must correctly load hierarchy of these objects as it is modeled. For example attributes must be assigned to entities correctly, etc.

Next, we will look at how to find the pairs of objects that are in maps-to relation, and how do they refer to each other across levels of abstraction. In example, how a logical attribute and a physical column, are tied together across 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 format. Knowing this Metadata Extractor we can find the required objects (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's call these models a solution. An *ER/Studio solution* is set of data models, describing a problem on both logical and physical levels (the two layers are only that ER/Studio supports). In a solution one logical model must be present whereas 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), 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 is allowed in a single solution.

.DM1 is a textual file format which is organized into many tables. Such table is a CSV (comma-separated values) structure with a modification, since the file format consists of many of these tables, (i) each table has a unique name which 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 the first sight, it is not clear how complex objects can be stored in files that we just described. To get the idea behind it, we did some work that we are going to described 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 13 records.

An attentive reader may find the terms we used for when describing a .DM1 table familiar. As we already seen 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 for 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 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 Metadata Extractor knows what data model objects are stored in the files and how the complex objects are deserialized there. The parser prepares an input file, thus Metadata Extractor can reconstruct the objects and their metadata stored in it.

PowerDesigner

PowerDesigner produces files storing data models with three types of extensions - .pdm, .ldm and .cdm. They stand for physical, logical and conceptual data model 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 state is called a workspace and can be saved into a .sws file. However, the workspace files do not bring us anything interesting. The information captured stores only what data models were at some time opened in 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 the XML counterparts. However, we need a Metadata Extractor user to save his models as XMLs, that is the only this way currently supported.

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 process 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 more memory and time consuming, but translates everything stored in the input file into data structure straightforwardly. Then it can be conveniently worked 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 and to figure out how the objects, Metadata Extractor is seeking for, 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 an advantage as the context of the XML elements matter, since, for example a definition, Metadata Extractor has to obtain, can be attached to an entity, an attribute and many other kinds of objects. Therefore, using a DOM parser is much more suitable and having the ability of doing 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 may be visualized when presenting a flow of data. Modeling tools are exhaustive pieces of software with many features, thus have the ability to capture plenty of aspects that a modeled system has. We determined types information, that can appear in data models of PowerDesigner and ER/Studio, makes sense to extract by analysis. Let us mention two important categories of metadata that we assume irrelevant to retrieve by Metadata Extractor.

Firstly, we will 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 and edges between objects are used solely to represent flow of data in a given system. The only relationship type from data models we will need to cope with is the inheritance relation, which is present in the enhanced-entity-relationship model. The difference is that without capturing the is-a links, an entity may not be described completely and the attributes that are inherited may be missing.

The second category of metadata that Metadata Extractor will ignore, are the ones 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 the 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 a 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 mapping are user defined. They can contain description of a relation, but we will 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 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)
  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 be not 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.

- Entities

  - Name
  - Attributes
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords

- Attributes

  - Name
  - Code
  - Comment

- Definition

- Annotation

- Keywords

- Parent Entity

- Inheritances

  - Parent Entity
    Predecessor.

  - Child Entity
    Inheriting entity that takes over attributes of the parent.

Physical Data Model

In this section we list objects from 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

- Tables

  - Name
  - Columns
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Schema

- Columns

  - Name
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Table

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 the common environment of a data model, are related to each other into 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.

Data Model		Owner/Schema		Table/Entity		Column/Attribute
-properties		-properties		-properties		-properties
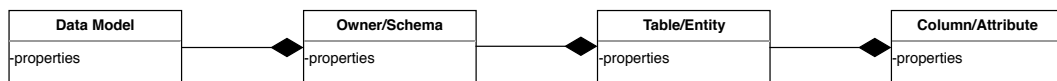
Figure 4.2: General hierarchy of objects in a data model. The UML diagram pictures a data model, that contains arbitrary number of owners (schemas respectively), they own 0 to N tables (entities resp.), while each entity consists of zero or more columns (attributes resp.). These objects can be seen as 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.

31

Metadata Extractor builds the tree from the top to the bottom. In the next two subsections we will 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 the objects is defined in a single table. Such 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 the 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 the records of the main objects therefore have also column with foreign keys 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 that the 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 build, Extractor examines its children. This way the context is clear and once, let's say an attribute, is created the program knows what is its parent entity trivially, as 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 keep the data models readable, and to keep track of what tables are implementing which high level concept.

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

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 how the mapping are realized in the chosen data modeling tools.

ER/Studio

In Section 4.1.2 we mentioned two types of mapping relations in ER/Studio for entities, tables, attributes and columns. In fact, 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 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, whose 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, third, kind of maps-to relation is referred to as a universal mapping.

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

Let's start with the seemingly easier case of mappings between objects inside the same ER/Studio solution - where-used and user-defined. After some analysis using the reverse-engineering tools (see Section 3.1.2) we found the table that stores the 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 that are referenced by the foreign keys. The Meta table types also allows 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 the first sight, solving the universal mappings may appear more difficult. It looks like Metadata Extractor will need to search for object in different ER/Studio solutions and reconstruct the external objects. But the way the universal mappings are realized in ER/Studio is much simpler. Metadata Extractor does not need to open other files, as the external objects that are 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 the all the mappings to external objects are defined, is using the same concepts as Where_Used_PD allowing Metadata Extractor to reconstruct them easily.

PowerDesigner

PowerDesigner saves every data model into a separate file. Therefore, to resolve mappings will be not as straightforward as in the case of ER/Studio, since every mapping of an object to another that is on a different abstraction layer leads across PowerDesigner files. Firstly, let's 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 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 objects 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 holds all the IDs of objects that the final object was generated from.

User defined mappings are represented by a composite XML element. 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 mapping 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's 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 of the mapped objects 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. Thus in a situation like this, data model file are 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 it 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 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 will 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 gets reconstructed. That potentially leads to a great amount 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 target 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 model 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, resolve mappings. Advancing like this would decreased 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. 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).
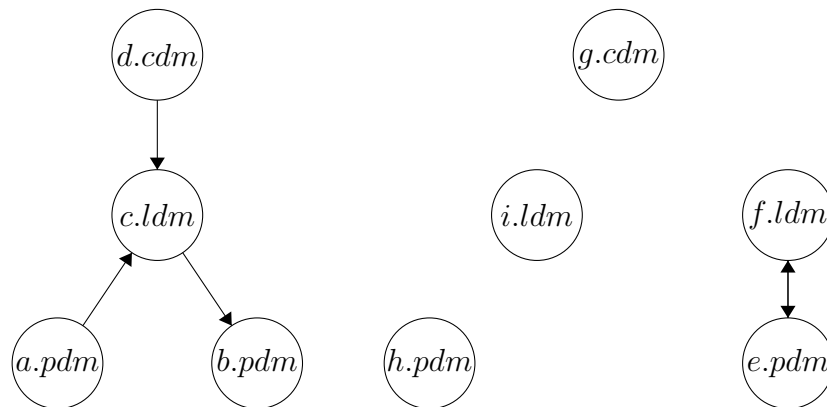
Figure 4.3: On the example we have four five components made of PowerDesigner data models. If dependencies of data models are like this, a.pdm should get processed and resolved in one batch with b.pdm, c.ldm and d.cdm. Then f.ldm must be handled with e.pdm, while rest of the models are independent and 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 in 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 input, but a change of an object in C:/PowerDesigner/Project/*b* affects *a* that is using the changed object due to the described problem. 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 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 a similar structure of the former and input directory, as well as that names of data model files did not changed.

4.1.5   Business Lineage Creation

One of the main goals of this work is to develop 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 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 physical level to 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 a 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 will go through 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 we will do 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 varies greatly. Instead we

---

[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

will make use of the fact that Manta does has connectors that do the job for us, stores the metadata in its own local database and has unified API for getting the metadata independently of database engine.

And why would we want to request another metadata when that is just what we are extracting 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 the 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.*
To this set of the listed properties and a connection string we will 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 to obtaining the database details.

ER/Studio

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

We cannot really work with that, so we will leave it up to a user to define connection parameters by hand. A .ini files is used for that where sections are named by physical models that they correspond to 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 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 tool has a nice user environment for creating or using connections to a database where a user is guided through a set up and 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 stores 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 parser for each supported database engine.

On the other hand there is 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 we deal with a native or a JDBC connection respectively, the structure can be parsed easily. The file consisting of property=value map. There are 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 which sufficiently defines a connection. In 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 approaches - there is no link between the 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. 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 indicate flow of data between them. The remaining part is how we will represent the output. We require a 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 already an existing browser-based user interface, using which the data flows can be shown and previewed interactively. Given that we would like to comply with the graph database and merge our graphs to the storage, and having the ability to reuse the visualization for presentation of the outputs, the most natural solution is to stick with the very same representation of 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 output of modeling tools and reconstruct 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 models, 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 development of an automated solution that delivers business lineage. In order to justify, that we are not reinventing a wheel let's have a look at the software 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 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 system's stakeholders to avoid misinterpretations when it comes to high-level terms.

    – Collibra
Works with business assets that connects 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 thorough 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 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 company 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 the version 18.0 is out today. Similarly with their PowerDesigner connector, it is not easy to find a documentation and even if something related is mentioned the information seem 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]

  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 convenient fashion.

- Resolver

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

- Parser

  Loads file to data structures that can be further worked with.

- Reader

  Puts together the model, parer and resolver unit - creates a model from a parsed structure of a file using the resolver and hands the result to the data flow generator.

- Data Flow Generator

  Creates a graph representation out of output of the reader component using the modeling common module.

These two modules are shared:

- Modeling Common

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

- Manta Flow

  The external part capable of extracting objects from databases, analyzing SQL scripts that are transforming those 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 context.
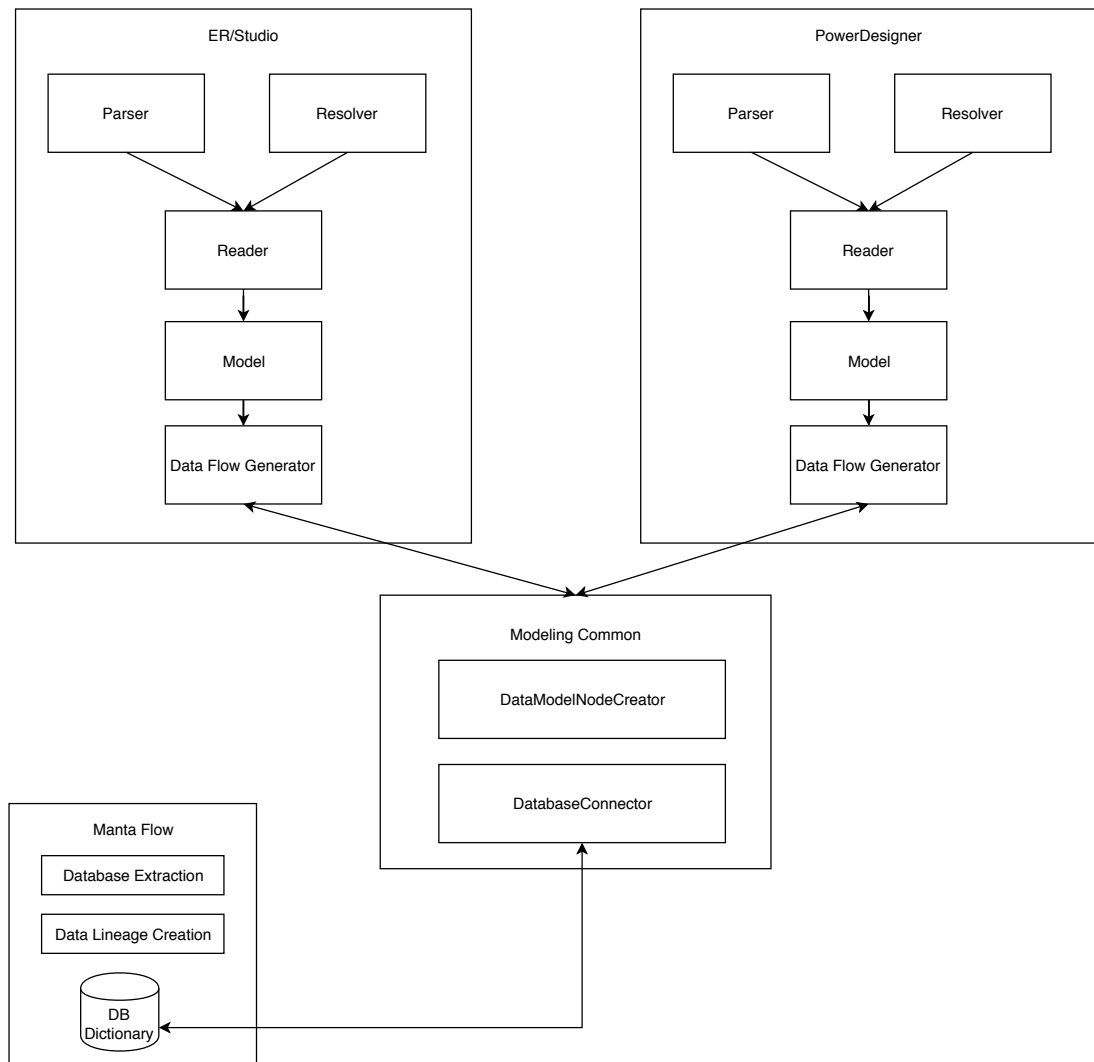
Figure 4.4: A high level view on software architecture of Metadata Extractor

5. Implementation

In this chapter we will mention the most important classes of our software and go through their main responsibilities. Here we describe the main three parts of the software - ER/Studio module, PowerDesigner module and the common part used by both of them. The shared module is a general artifact that is designed to be helpful when designing an extractor of metadata from an arbitrary modeling tool that is aimed to be included in Manta Flow.

We will go through the implementation from a higher perspective mentioning some classes and their methods from the source code of Metadata Extractor. Not every class and method will be described in this chapter, however precise programmer's documentation containing these information comes in a form of JavaDocs.

5.1   ER/Studio

5.1.1   File Reverse-Engineering Tool

We introduced that file format that ER/Studio uses for storing data models consists of a relational databases like CSV tables (see Section 4.1.1). In order to be able to deconstruct objects stored in the files, we need to decrypt relations between tables in ER/Studio .DM1 files, as it is not easy to see how structured information is stored in those tables. The task is to find out what links are leading between tables in the format. Relationships in this type of tabular data storage are done via primary and foreign keys. For this purpose we developed a little reverse engineering tools that should help us to reconstruct complex objects, as aspects of such objects are usually stored in multiple tables.

Input for this tool is an arbitrary .DM1 file and output must describe the logical layout. We observed the analogy of the file format logic with relational databases, which can be represented understandably by a relational diagram. Thus it would be nice if output of the tool would be such diagram showing the organization of the .DM1 format. However to create a diagram directly would be overkill we will use the knowledge that for creation of relational diagrams modeling tools are used. The tools usually have the ability to transform some procedure defined in SQL into visual representation. For example based on the statement CREATE TABLE T modeling tool can draw a box in diagram representing table T. SQL surely can define columns of a table and create key constraints on columns as well. Modeling tools are able transform a foreign key column in table B that is a primary one in table A into a relationship between the two tables showed graphically in a diagram.

Thus, if we are able to generate a SQL create procedure for each of the tables from the file format and define by the query language primary and foreign keys of the SQL tables, modeling tools would 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

43

straightforward once we parsed a .DM1 file. On the other hand deciding which column of a table is primary, which columns are foreign keys and to primary column of what table they refer to, is a more complicated task. So we need to take the reverse-engineering process one step further - to be able to infer the keys. Modeling tools commonly have the ability to reverse-engineer a relational database, however they are not able to tell relations between database tables. If so the knowledge is based on metadata from a database where keys are already defined, while the plain create statement in SQL do not provide any additional information like that. In this situation we have no better option than to come up with own solution, as we have not found any utility inferring relations between tables solely based on column names or content of database tables. Output of the reverse engineering tool therefore will be a SQL script containing table and key constraint definitions.

As the development of the reverse-engineering utility is not the focal point of this work, the utility will not be an out-of-the-box general solution for deducing keys of relational databases, but should provide a basic overview of .DM1 files organization. Also it is not going to be a definitive foreign key deducing solution but is primarily concerned with the objects and properties picked by the analysis described above in Section 4.1.2.

In order to create the reverse-engineering utility we must decide what programming language we will write it in. Although there may be scripting languages that would make some operations, like joins on tables easier, we decided to use Java as 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 suffices to write the parser once.

Firstly, an analyzed .DM1 file is parsed into a set of `CsvTable`s.

When trying to find out what are columns with 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 a table. This approach is represented by the class `DependencyCreator`. It treats columns that look like keys (eg. those which name ends with "ID", the policy is determined by `isOnBlacklist` method) as the same across all loaded tables. The main method here is `createDependencies` which pairs potential key columns and tables containing the candidate columns. 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 a sequence of joins needed to do in order to put put together records from table A with those from table B.

The second view takes into account data stored in the tables as well. When analyzing what are relations between columns, we consider a table as a collection of columns. A `CsvColumn` stores a set of values from all records of its table at position of the column. This way we can examine if columns with same names have similar contents. In contrast when getting records from a table we see it as a collection of `CsvRow`s. The `RelationFinderByTableContents` class is used for the further search for key columns.

It filters tables that have no content, thus are irrelevant for this approach. The relation finder class which is analyzing contents of tables has a method for

determining if a column can be considered a primary key of a table. That is true only if a columns 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. If column A contains foreign keys of primary keys specified in column B, each entry of A must be present in B, thus A must be subset of B. In .DM1 file format there is a table `Identity Value` that explicitly pairs tables with their primary key columns. Of this information the reverse engineering tool takes advantage as well. The class also inspects if the primary keys are defined in strict order or if there is some pattern in 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 and to finally generate the desired output in form of a SQL code defining .DM1 tables and their keys out of the analysis done the class `TablesToSQLConverter` is used. The main method is `writeMySQLSource` which at the beginning defines SQL create statement for each table from ER/Studio source file. Then the phase of creating the key constraints takes place. It operates on candidate columns that satisfy policies in `DepedencyCreator` to be key columns and are present in at least one table so one or more joins can be made using the keys. There is also space for defining manually the knowledge we gained by inspecting the format using `RelationFinder*` classes by inserting pairs table and its primary key. At first place primary keys defined from user-defined information are set, then the information stored in `Identity Value` table is used for key definition. Lastly, key inference based on policies a name of a column in context of its table takes place. For example if a table is called tableA_id it is very likely it is a primary key if it comes from table called tableA. If a column is set as a primary key in one table for each of the remaining tables that contains the column with same name, the column is marked as a foreign key. In the case there are columns that were chosen as candidates for being keys but no constraint was assigned to them in previous steps, in one of the tables which contain such column is marked as a primary and in the rest as a foreign one. That 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 information saved in the file format, we 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 CSV record is, that each entry is located on a separate line, fields in an entry are separated by a comma, the last record is followed by line break. A field may contain a comma or line break but must be enclosed by 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].

The structure we need to parse contains tables consists of a name on a single line, definition 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, definition of columns, in other words names for each of its properties `CsvColumnDefinition`, and finally records themselves - instances of `CsvColumn` which are contents of a table.

However, the view on a table may vary by usage of the parser. Using the reverse-engineering tool, we inspected what values are stored in a column, examined the domain of values across all records etc. On the other hand, if we see a table as records, each record having properties, one property is one column, from this perspective a table is a set of rows. The second case is needed when reading the contents of a table for the sake of obtaining its records.

Getting a name of a table is an easy job as its just a single textual line. To resolve a record, 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 our CSV table.

We propose a non-deterministic finite automaton, even though it can be translated to a deterministic one, we consider an 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.

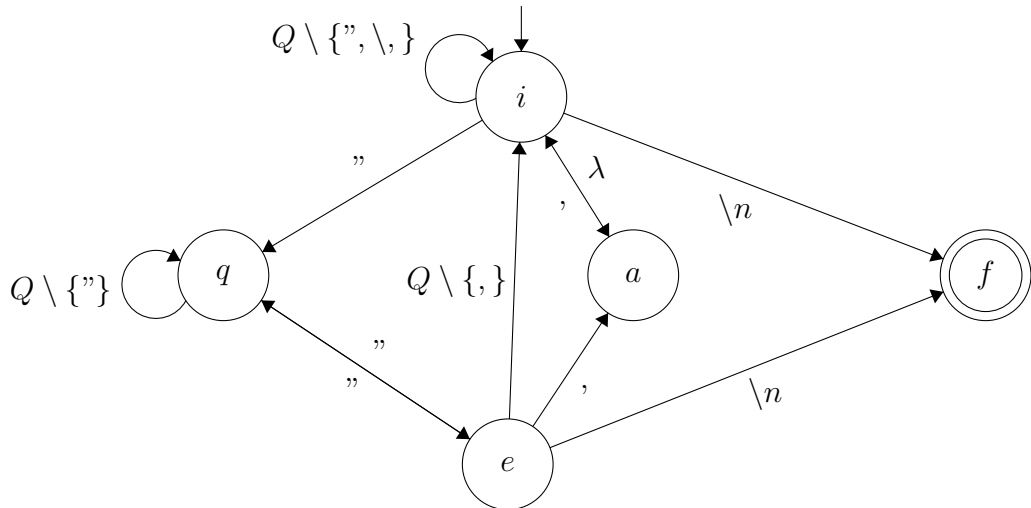Figure 5.1: Non deterministic finite automaton accepting a line of a CSV table in .DM1 file. In the state a and f, the end of a CSV field is recognized and accepted.

When parsing a CSV record consisting of multiple fields not only we want to determine the end of a CSV entry but to collect the fields 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` taking a file to process and returning pairs of table name and an instance of `CSVTable`.

5.1.3   Model

The purpose of the model component is to provide an access for reading to both a raw structure of a processed source file and a fully loaded hierarchy of objects we reconstructed from a file.

The crucial objects and the important properties of theirs result from the analysis of ER/Studio data models listed in Section 4.1.2. Those are the ones the model is required to capture.

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

An ER/Studio solution - a set containing one logical data model and arbitrary number of physical models implements `ErStudioSolution` interface. All the objects contained in the data models stored in such 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`.

Since a .DM1 file is not restricted to storing a single solution, but may reference external models as well, thus there may be objects from multiple solutions saved in a file. If name of the solution's origin file is identical with the name of .DM1 file from which it was loaded, it is an internal file. The overall structure of a file is represented by `ErStudioFileModel` that has a main solution - the internal one and possibly references external solutions.

A base behavior of both logical and physical data models is defined in `DataModel`. A data model has a name and contains owners. An instance fulfilling `DataModel` interface must have property of type `AbstractionLayer` set to either `Logical` or `Physical` just like every object that is contained in a model.

`PhysicalDataModel` has an important addition, that it can tell the database platform the low-level model is designed for. In enum class `Platform` all the database management systems ER/Studio supports are captured with an extra entry for an unknown DBMS.

`Owner` is either `Logical` or `Physical` according to what objects it can own and what type of model it may belong to. It has name and allows access to the objects owned by it.

The objects that can be mapped to another object implement interface `Mappable<T>` where the type parameter defines what is the kind of the mapped counterpart.

Objects in a data model that can be described further by definition and note while having a name can fit into `DataObject` interface. These can be either `CompositeDataObject` or `SimpleDataObject`.

An interface for the composite ones describes what is expected from a table or an entity. They have the very same properties that is why a single class is enough to capture their structure. Which of the two it is is decided by `AbstractionLayer`. The composite one can be mapped to another

`CompositeDataObject` and may contain simple objects.

The `SimpleDataObject` class is used to represent columns or attributes. Equivalently, layer of abstraction is crucial to determine whether the object fulfilling the interface can be stored in `PhysicalDataModel` or `LogicalDataModel`. It must be invariant, that the whole subtree of objects, beginning in a data model must have the same abstraction layer defined.

The reason to represent pairs of, at first sight, different concepts of tables-entities and columns-attributes in a single classes is based on how ER/Studio treats them internally. Given that columns and attributes are stored in a single CSV table and the distinction if such object is of one or another 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. This can be applied to composite objects as well. Even data models are treated similarly, only there is an additional attribute related to DBMS type in the case of physical models, while the logical ones do not need such information.

class diagrams?

5.1.4 Resolver

The purpose of the resolver unit is to actually create the objects which are form data models which Metadata Extractor obtains from modeling tools. These results of the resolver module, thus are fulfilling the contracts specified by the functionality defined in the model.

Outline of what the resolver module does is that it via `ErStudioFileModelBuilder` creates the whole structure representing an input .DM1 file as instance of `ErStudioFileModel`. It builds an internal solution, then external solutions are created and at the end resolving of mappings that lead across the solution using `ExternalMapper` takes place.

Typically, for each interface, we will create an implementation. In Java it is common to call the classes fulfilling an 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 the model component, need to dispose of methods for setting up and adding properties or substructures as the model's reading methods would not be very meaningful otherwise. These classes can be found in the imp package.

We defined what is expected from the classes and added the functionality helping to set up the objects, once we gathered required information. So the last missing piece of puzzle needed to create the objects is how to gather the required information. The logic taking care of collecting the needed data is hidden in the package builder. The skeleton of how an instance of `ErStudioSolution` is created is prescribed in the `AbstractDataObjectsBuilder`'s method `buildErStudioModel` that 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 internal or external type. The template method enforces a tree of objects in a solution is build from the top down. The very first is to step is to create a root of such tree - an instance of `ErStudioSolution` that is being build, then `DataModel`s from the solution are loaded, `Owner`s fol-

low, `CompositeObjects` after and finally `SimpleObject`s. Child instances are appended to their parents just when created. The abstract builder contains common methods for construction of objects. Those methods are not dependent on representation, only require a set of properties as input to be able to attach them to resulting objects to make them complete. The groups of properties needed to be collected about each type to proceed the construction are described by interfaces in the package modeledobjectproperties.

The specific way of gathering the information about objects to be reconstructed is tied with details of how the objects are saved in the file format. Concrete builders `InternalDataObjectsBuilder` along with `ExternalDataObjectsBuilder` provide implementation to the abstract methods. In the case of internal solution, information about objects are retrieved from CSV tables, where for each table there is a dedicated and references across the tables are resolved. On the other hand, in case of external solutions, they are fully represented in a single CSV table that stores XML structures describing these objects. The implementations of the placeholders in `ExternalDataObjectsBuilder` uses a simple SAX XML parser to retrieve the desired data described in modeledobjectproperties.

`InternalDataObjectsBuilder` has one more responsibility and that is to create mappings between `Mappable` objects in a solution with a help of `InternalMapper`.

The `InternalMapper` class only has knowledge about the layout of a CSV table containing internal mappings, just like the `ExternalMapper`, that we will need later, has about the definitions of external mappings. However the logic of putting objects to maps-to relation is coded in the `AbstractMapper` class. It goes through a table which definition fulfills the interface `MappingTable` and links pairs listed in the table by mapping, if their types are equivalent.

a class diagram?

5.1.5 Reader

The reader component puts together the parser module with resolver and produces a data structure by what the model component defined.

`ErStudioDm1Reader` crawls through directory with input .DM1 files processing them. Each time the `read` method is called the parser processes next file from the input folder. Parser's product is handed to resolver resulting in `ErStudioFileModel` that is what `read` returns.

5.1.6 Data Flow Generator

Having constructed the objects defined in the model unit, Metadata Extractor sends them to Data Flow Generator unit, where the model get transformed into an output graph and connects to Manta Flow to enrich the graph of ER/Studio objects by data lineage.

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 out case we will use a single task whose input will be

a data structure described by the model unit and output a graph with extracted information.

Namely, `ErStudioDataflowScenario` reads an input file, and executes the task - `ErStudioDataflowTask`.

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.

A suitable `Connection` corresponding to a physical data model is made of information from a data model and .ini file to be able to pair physical data model objects with those from database dictionaries.

Once assigned the connections to physical models, we can traverse the data models trees, creating nodes using `DatabaseConnector` in the case of physical level, whereas when it comes to logical objects Metadata Extractor takes advantage of `DataModelNodeCreator`.

As the tool creates the mappings symmetrically, the full information is captured on both 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 though as each mapping edge is contained in both of the levels. For the sake of simplicity, as there usually is less logical models generally a logical model contains fewer objects than a physical one, 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. Then physical objects are created as nodes and if the just constructed node has a mapping, MAPS_TO edge is set between the mapped nodes.

5.2 PowerDesigner

5.2.1 Parser

The files storing the data model objects we want to reconstruct from PowerDesigner 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 in our case should be loading the inputs to a DOM tree data structure. We did not try to reinvent a wheel as there is many DOM parsing services. Given that in the ecosystem of Manta software there is an internal artifact made for transforming XML to DOM Metadata Extractor will make use of it.

5.2.2 Model

Let's start describing the main structure the model has to capture generally. In the case of PowerDesigner a model is saved in a single file. Thus `DataModel` has the name of the file where it is stored and its sub objects, then the hierarchy goes like this `Schema` has `CompositeObject`s which contain `SimpleObject`. This basic skeleton of a tree structure is present in each of the three data models. What more, on logical and conceptual level, as they are represented by an EER diagram, an entity can inherit from another one, thus an interface `Entity` introducing the concept of parent entity will be used by them. Each of actors may have some metadata providing further description and extends the interface of `NamedObject`.

These interfaces are generic and provide the common structure. For every model type there exists a package, where specific objects' contracts lies. These concrete interfaces on one hand takes over the common concepts. On the other hand may be easily extended if a new functionality, which is data model specific, is identified.

`Mappable` interface is present to be realized on instances of `CompositeObject` or `SimpleObject`, however a target of a mapping is a globally unique id of an element, not another instance directly. Since by the nature of mappings in PowerDesigner the actual representation may be unreachable yet.

5.2.3 Resolver

The impl package contains the implementational counterpart to the model unit. Common concepts with no that do not take part in data models in reality and are present to minimize redundancy and provide way to handle similar objects, for example `CompositeObject`, are implemented as abstract classes - their names are prefixed Abstract. The actual objects that are implemented via *Impl classes.

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

The API for creating a data structure representing a data model is simply defined by `DataModelBuilder` on what the method `buildDataModel` may be called and the result is collected from `getResult` operation.

However, there is a different strategy used for each of the data models based on type it is. So when processing a PowerDesigner file the program must determine the suitable implementation of `DataModelBuilder` accordingly. A factory template class `BuilderFactory` is used to determine which implementation - `PhysicalDataModelBuilder` or `LogicalDataModelBuilder` or `ConceptualDataModelBuilder` by the extension of the processed file.

The outline how the builders work is proposed in the `AbstractDataModelBuilder`. Also the functionality independent of 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 providing the general features.

The implementations take an advantage of the fact that forming a tree like structure from an XML format is natural and straightforward, that is why for now we will omit a general point of view on a construction data model that would hide the convenience and context provided straightforwardly by DOM nodes. Requests for specific DOM nodes in the builders are done using XPath, which allows querying fully loaded XML trees.

5.2.4 Reader

The reader unit formed by `PowerDesignerXmlComponentReader` reads a directory containing output file of PowerDesigner that are at the same time inputs for Metadata Extractor - .pdm, .ldm or .cdm files and identifies related groups of them. Then proceeds one group after another, in parsing the XML files into a DOM document and passing the DOM trees to resolver that creates a corresponding set of `DataModel` objects using resolver's `DataModelBuilder`, which can be further passed to the data flow generator unit.

Input directory containing file to process is the input for `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` and a simple handler `TargetSAXHandler` created for the purpose of getting paths to related models from an XML. The found 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 input are collected component creation takes place so the input set is split into smaller logically connected groups.

Then when the main API method of a reader, the `read` method, is called, it returns one processed component - a set of related reconstructed `DataModel`s. 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

When translating a component of reconstructed data models, a suitable `GraphBuilder` must be chosen based on the layer of abstraction for each data model using `GraphBuilderFactory`. Then the tree of data model objects is traversed. When dealing with physical data model, `PhysicalGraphBuilder` is used, underlying `DatabaseConnector` searches for the modeled database objects. A suitable `Connection` corresponding to a physical data model is made of information from a data model and .ini file, loading connection details from .dsn and .dcp files is not supported yet. On the other hand, physical and logical data models use the generic `ModeledGraphBuilder` for creating nodes representing objects brought from data models of greater abstraction. The generator collects nodes that have mappings by their ids and once all the data models in the processed components are fully built, it means we can resolve the mapping as there are both ends of the MAPS_TO type edge we need to create them. Then we simply look at to what ids are objects mapped, obtain an actual representation of these ids put them into the relation by creating the edge.

5.3   Extensibility

For the sake of extracting metadata from modeling tools we developed a unit where a component for a specific tool meets manta flow. The artifact where the common bridging logic is stored is called manta-dataflow-generator-modeling-common. Physical data models need to be paired with corresponding connections to DBMS in order to correctly pair the same database objects that appear both in a physical model and in a database dictionary extracted by Manta Flow. This functionality is provided by implementations of interface `DatabaseConnector`. Details about the target database of the low-level data model are kept in `Connection`. The API to database dictionaries created by Manta is provided by `DataflowQueryService`. Having collected information of a physical object, either a column or a table, the query service tries to find it

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 our tools copes with the situation in such a way, that it creates 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 exists or we provided inaccurate/incomplete data to identify it.

So the first general 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 match 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 object do not match the ones from database that take part in data lineage, there is no flow to be propagate 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 is responsible for creation of node representation of a type of object that may appear in the data models. Most commonly a modeling tool will support only subset of the types the node creator is able to construct. However that is not a problem as the hierarchy is not strict and it can be built to be compliant with concepts 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 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 conceptual and logical level are described by ER or EER diagrams. The important fact is that the possible kinds of objects will be the same in both models, that is why a 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. `Resource` differentiates technologies, in our case it makes sure objects extracted from modeling tools are aggregated by what specific tool is their origin.

To attach metadata about a node, such as its definition or a comment, object that is being transformed into node representation has to implement interface `NodeMetadata`. The contract forces the implementers to expose the metadata to be attached to node representing them in the output in form of strings. More precisely, the objects have to fill a `Map<String, String>` data structure where a key is the name of a property. These metadata are shown when presenting the output graph and further describe the extracted objects.

Basically, the outline 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 model component. The objects that will be represented as standalone nodes in data lineage visualization at the end have to implement `NodeMetadata` interface.

2. Depending where from the data models are obtained, this phase generally retrieves data models from a 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 some representation of data models was reached. This stage transforms the acquired object into objects conforming aspects described by the model module.

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

The quality of 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.

5.5 Testing

Automated unit tests validating the crucial parts of the implementation are a huge help to determine correctness of Metadata Extractor's logic.

ER/Studio Parser

The correctness of parsing process is checked by class `ErStudioFileFormatParserTest` which is controlling how many CSV tables from a parsed .DM1 files was 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 problem on our side and parser does not pass the tests.

Reader & Resolver

Testing of the output of the reader module shows tells a lot about the resolver's correctness, since the logic of reader is actually provided by this module. From the nature of reader, that it connects resolver and parser, these tests rely on the fact that resolver works correctly and we assume this is true as far as the parser module tests are passing. The goal of the reader tests is to find out whether all objects from data models were successfully loaded into data structures described by the model module. In case of ER/Studio the testing compares expected statistics about data models such as number of physical models in a solution, entity count of logical model etc. with the amount of objects that were loaded actually by reader. Secondly, some specific objects from 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 another objects as well.

PowerDesigner must create components correctly, so is present a test for checking if number of components is correct. The main reader test consists of reading file by file, thus 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, we are interested if objects loaded by 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 same the behavior of the output building process is correct.

The previous test checked the representation of the layers independently, in `*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 expected set of maps_to edges.

Conclusion

We have developed a piece of software 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 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 another modeling tools may take advantage of it and can be aware of what contract to meet in order to create business lineage and to enrich the physical one.

One of our aims was to explore possibilities of modeling data lineage in modeling tools. We went through it and described the important aspects of the 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 as users continue to report problems with Instagram and WhatsApp more than 14 hours after global meltdown.

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

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

[4] Right of Access. `https://gdpr-info.eu/issues/right-of-access/`.

[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`.

[7] `https://www.ebayinc.com/stories/blogs/tech/nosql-data-modeling/`.

[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, 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`.

[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`.

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

[17] PowerDesigner Main Features.

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

[19] Lineage vs traceability understanding the dif-ferences. `https://www.collibra.com/blog/lineage-vs-traceability-understanding-the-differences/`.

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

[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`.

[22] ERSTUDIOBUS Release Notes. `https://isp.asg.com/Rochade/ERStudio/V2.3x/ERSTUDIOBUS_237_001_Release%20Notes.pdf`.

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

[24] Sap powerdesigner documentation.

List of Figures

List of Tables

List of Abbreviations

ER	Entity-Relationship
EER	Enhanced-entity-Relationship
CDM	Conceptual Data Model
LDM	Logical Data Model
PDM	Physical Data Model

A. Attachments

A.1 Building

The build process is ensured through Maven with dependencies specified with its pom.xml files. However, in order to build Metadata Extractor "secret" dependencies owned by MANTA are needed. Otherwise the project will not compile successfully. Also in order to gain the whole functionality of our tool a context provided by Manta Flow's database extractors, post processing done on server, where the interpolation takes place, and visualization of data lineage which is also part of the server's functionality is needed. There is a command line user interface putting all the parts together and must be backed by manta server with graph database that is running user interface as well. The crucial parts of Manta Flow though can not be built without having access to MANTA's code repository.

Even with the access to the repository, MANTA is a proprietary software and running its products requires a valid license.

Thus only those jar files that are needed for running tests are attached. They cannot be rebuilt by a user.

A.2 User Documentation

To run Metadata Extractor the prerequisite is to have Java 8 on a computer.

As we mentioned in Appendix A.1 with this work come attached only files needed to demonstrate functionality on tests due to licensing of Manta Flow product

A.2.1 Tutorials

to include or not

A.3 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, further divided by technologies. Subdirectories erstudio and powerdesigner are located in input. Both have the same following layout: It contains systems, a system is logical group of modeling files to be processed. System consists of two items - one is containing the models from which metadata will get extracted and a .ini file where connection specifications to physical models are defined.

The systems to be processed from input are specified by .properties file /scenarios/etc. For each system in input a property file specifying a name of the system folder to be processed must be present.

To proceed the extraction user runs one of the executable files provided _run.sh for Unix-like, _run.bat for Windows machines respectively.

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

– Relationships

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

- Entities

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

Properties We Will Not Extract

- Number
- Generate
- Identifiers
- Rules
- Stereotype

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

    Properties We Will Not Extract

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

- Stereotype

- Inheritances

  - Parent Entity

  - Child Entity

Objects We Will Not Extract

- Relationships

- Identifiers

- Associations and Association Links (CDM only)

- Domains

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

- Tables

  - Name
  - Columns
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Schema

    Properties We Will Not Extract

  - Number
  - Generate
  - Dimensional type
  - Type
  - Indexes
  - Keys
  - Triggers
  - Procedures
  - Check
  - Physical Options
  - Preview
  - Lifecycle
  - Stereotype

- Columns

  - Name
  - Code
  - Comment
  - Definition
  - Annotation
  - Keywords
  - Table

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

• Users, Groups, and Roles

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