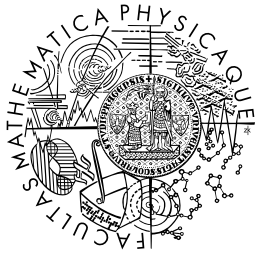

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Denis Drobný

Extracting Information from Database Modeling Tools

Department of Distributed and Dependable Systems

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

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

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

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

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

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

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

In ..... 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 systems. If the system is a database, tables and columns are visualized along with SQL transformations. 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 that on one hand brings further description of the database objects, as well as introduces a whole new perspective on data in a system by business lineage aimed for non-technical users. The additional metadata enriching data lineage are extracted from data modeling tools that are widely used in database design process. The solution extends Manta Flow lineage tool, while taking advantage of its features at the same time.
mention PD & ER/Studio or not?

Keywords: Data Lineage Data Modeling

Contents

1. Introduction

There is no business today that can live without being backed by a database. No matter what field an enterprise is focused on, we can enumerate many reasons why a database storage helps a company to be more effective and its deployment is a good idea. We will justify it using 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 a 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 at doctor's office or 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 like 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.

And the list goes on.

Once the decision is made and the usefulness of a database for our business is proved, there may be still a long way until everything runs as expected and we 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 knows what real life aspects need to be captured in a database. To convert this idea into a working solution, the company would hire a database designer.

A discussion between an expert in the business domain where the enterprise operates and a database professional follows, 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, better solution is to translate the debate into more intuitive and standardized piece of documentation, into a

conceptual data model. Once the initial model is created the next steps are going more and more toward an final implementation of the database. After, a database designer works on development of a logical model and the most high level concepts are transformed into the one that is combining high level perspective with more technical aspects, but the description of them remain independent of a database type.

Finally, the organization of the database is pointed out and captured in a physical model of the analyzed system, from this point we have a solid documentation and it is straightforward to finally deploy a database that is described in the low-level model as is has one-to-one mapping with 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 will it 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 logic that computes how the price of the remaining tickets should be raised and update the records in database representing the not taken tickets accordingly, so the valid information is shown to customers. The logic takes places thanks to by 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 possibly huge number of sources and transformations hidden in scripts.

Data lineage is the answer for the struggles with being overwhelmed by 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 eg. the conceptual is for business people while the physical one is read by database engineers. But when it comes to data lineage, it only displays the level of abstraction that is understood by database professionals, while people with not that good technical background that would want to make decisions based on how data flows in their system 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 exist a documentation of their systems in form of data

models, we will try to reuse to bring the desired functionality. Data models also store valuable metadata that can make data lineage, even the technical one, more readable and transparent. Even though the business lineage will provide 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 im the company stores upon request. With 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 parts of it. Although data lineage does not make a company automatically GDPR-compliant it is a shortcut to get there.

1.1 Goals

- Develop a component that extracts metadata from database models that were created using SAP PowerDesigner
- Develop a component that extracts metadata from database models that were created using ER/Studio
- Provide a description by means of a programming language for a general scenario of metadata extraction from a data modeling tool output and passing the information to a data lineage tool
- Propagate data lineage acquired by analysis of how is database used and constructed to more abstract data models than is the physical one, to the logical and the conceptual models.

1.2 Glossary

Let us introduce some crucial terms used throughout the 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. 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 user 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 Chapter overview

In the chapter On Database Background, we will go through the concepts fundamental for understanding the domain to which our tool contributes. The prerequisite are to understand databases, data modeling and data lineage.

The chapter Modeling Tools describes more specifically what are the pieces of software used for creating data models capable of doing with. Also we will discuss the specific tools from which we will extract metadata.

The fourth chapter is concerned about analyzing the aspects of modeling tool we will work for. Based on the analysis we identify requirements for our software and propose a high level architecture of solution that we compare to the already existing one.

Lastly, the implementation chapter is discussing the resulting classes and their responsibilities which the final product is made of.

2. On Database Background

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 we 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 we want to use a database and roughly mentioned what are the pieces of data that we want to save there. Now let's take a look at what are differences between in database implementations and what to take in 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 build around both, Relational Database Management System (RDBMS)

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 an object whose attributes are represented by columns and instances by rows. The important aspect is that relational tables tables carry both data that need to be stored by user and the relationships between the data as well. To store an atomic piece of data about instance a proper 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 table and is used when referring to the record.
A *Foreign Key* is a Key that uniquely identifies a record from a table (may be the same or a different one).
They are known also as SQL databases by the language - *Structured Query Language (SQL)* which is used in RDBMS for managing data.

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

In this work will focus only on the databases that are of the relational kind, even though there are also NoSQL or Non-relational 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) modeling of these databases quite a new

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

discipline and is hard to find an intersection among different approaches to NoSQL modeling. Also concepts of higher abstraction models are omitted. [7]

The fact to consider is that once a database is Relational we more or less know what to expect from it. The structure of these databases has a fixed skeleton. So a tool that would extract metadata from relational data models is potentially more powerful as it can be applied to more database technologies than a similar tool aimed for some specific type of Non-Relational database.

Lastly, despite the Non-Relational may be growing in numbers and became a serious alternative, as it suits some use-cases better, the Relational still are, and in the near future will be, far more widely used.

Means of Database Access

Databases can be managed directly using Database Management Systems by a user who is using query language for accessing a database. However third party, or application, programs need also 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 in the extraction phase. A solution is to provide them with an application programming interface (API) that provides a set of methods available in the programming language that the application program was written in, so it can use them. Most commonly when the API is called its implementation translates the request so that to a specific DBMS driver that it is passed after 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 it is made of pairs of keywords and values separated by semicolon, the keywords are parameters of the connection.

APIs to DBMS

- Open Database Connectivity (ODBC)
  General, language independent, ER/Studio, PowerDesigner
- Java Database Connectivity (JDBC)
  The Java ecosystem, Manta Flow, PowerDesigner
- ADO.NET
  .NET Framework

2.2 Database Modeling

Modeling is a crucial phase of 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, some toilets and expect a good result. Most probably some building would be produced, 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 the analogy to the database domain.
When deploying a database from a scratch we 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 the 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 expansive and leads to numerous outages.

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

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

- Costs reduction.
  Errors are identified thus can be caught in early stages, when they are easy to fix.

- Better documentation.
  Data models form a nice piece of it, 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.

- Determining of consistency of the system.

- Deeper understanding.
  During the design process we may learn a lot about properties of the data that we 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 if so which DBMS is the one for us, or to look for a non-relational one.

2.2.1 Data Model Perspectives

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, view of the conceptual level.

- 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.

On the other hand, to standardize 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 our 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, for example the first level is today omitted, to the one that is recognized nowadays. The differentiation takes takes into account what is the audience that will work with a data model, whether it is someone who knows all about databases or 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.

- 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 showing all specific internal details that may be overwhelming in the case that the reader is a computer specialist.

[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.

Horizontal Division

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.

conclusion Conceptual and logical data models are usually represented by ER data models. The question is what specific data model type is used for physical models. As the most low-level model type is tied directly with how a database is organized, physical models must obey the structure of database.

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.

Characteristics

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

A real world object is captured by an *entity* in conceptual model.
For further description of objects that we are interested in *attributes* are used, those are properties of entities. Only the important ones are listed. [3]
Also *relationships* between objects are necessary to provide full view of the section of the world that a data model resembles.

To illustrate it on an example, if our modeling domain is education, then an entity may be a teacher or lesson. A lsalary number 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 who, that would be captured using relationships.

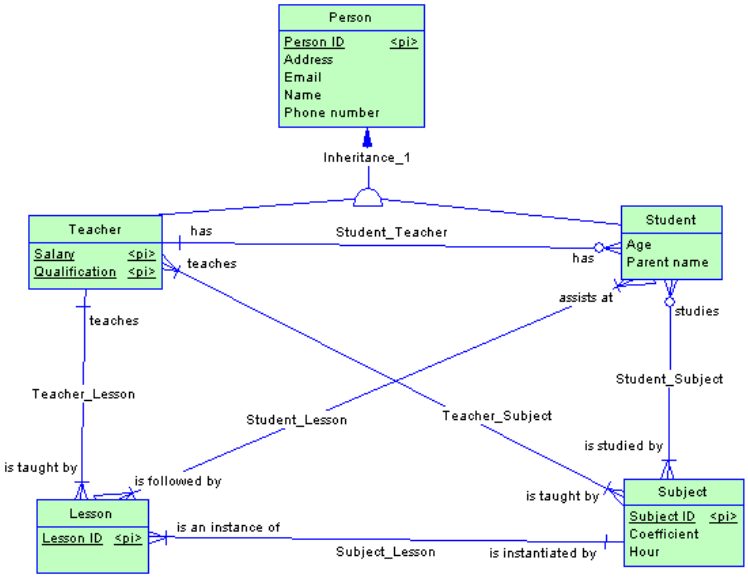

Figure 2.1: Conceptual diagram[13]

2.2.3 Logical Data Model

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

Characteristics

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

[3]Definitions varies and in some literature can be even found that a conceptual entity lacks attributes. We assume that the entity can contain important attributes as it is more common interpretation and modeling tools have attributes support on conceptual layer as well.

- Normalized up to third normal form.

Entities, *attributes* and *relationships* from a conceptual model are present on this layer as well. Relationships are not that abstract as before and keys that actually make relationship happen between entities are added as their attributes.

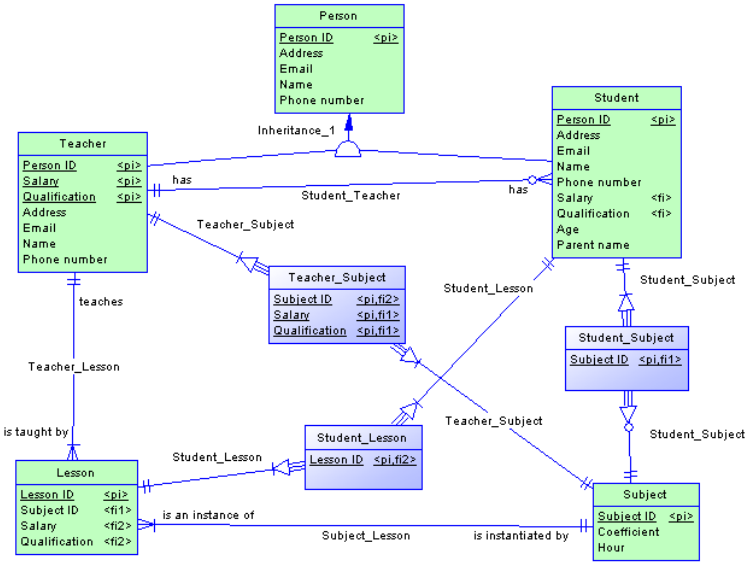

Figure 2.2: Logical diagram[13]

2.2.4 Physical Data Model

A physical data 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 actual implementation. Its main message is to communicate how the data are stored.

Characteristics

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

Objects in physical models should reflect database organization and at the same moment related higher-level concepts should be transformable to physical level. *Tables* should store records that corresponds to logical entities and *columns*

represent previously described attributes in memory. Commonly schemas[4] are present. A *schema* is basically container for tables that logically groups them. Users have usually schemas assigned and can access only the tables contained in those schemas.

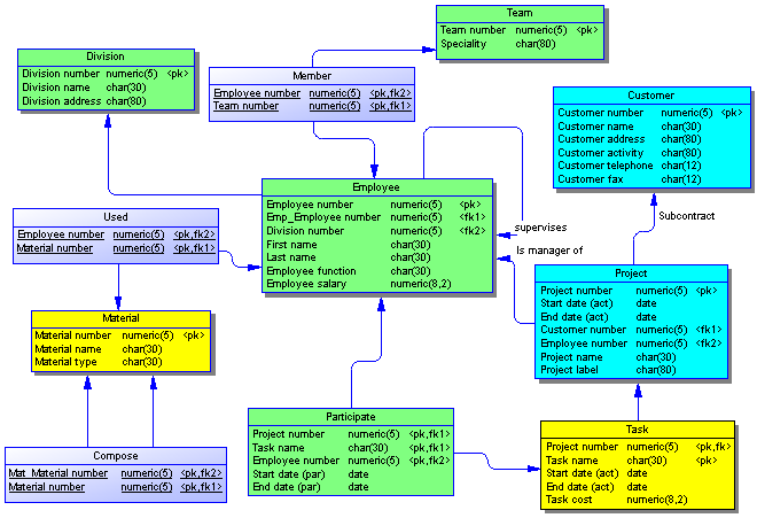

Figure 2.3: Physical diagram[13]

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 and what are the implications.

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 should a database store and what are the relations between stored object. 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 case of database development when a client identifies a high-level need for a database and hires an expert in this domain to make it happen.

The other way to create 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 of bottom-up strategy is how doctors 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 and after they are logically grouped to form bigger units, entities, and so on until the full hierarchy is known.

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

Maps-to Relation

In order to capture how high-level concepts are actually realized by more precise object a relation that we will call *maps-to* is used. The relation leads between objects that are semantically equivalent on different levels of abstraction, sometimes even mapping between objects on the same layer are allowed but we will not consider this, as we consider it be mixing two different concepts together - data modeling with data lineage. To be more precise what we mean by semantically equivalent objects in data models is that we will assume maps-to edges solely source is model and target is model, entity and table, attribute and column, or the other way around. Following these mapping links is extremely useful when a person wants to gain an overall overview of the system and comprehend it. For example when a user sees a data table in physical model that has a technical name that obey some naming convention and due to normalization does not represent any object straightforwardly, he can follow mapping links that leads to higher layer providing greater abstraction over the implementation and the motivation why the table was created should be much clearer then. It is worth mentioning that usually the mapping relations between objects of different layers simple one-to-one relationships but the cardinalities may vary greatly. For example one logical attribute may be 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 $<$ number of logical objects $<$ 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 to play. We consider the mapping relation symmetrical.

2.3 Data Lineage

Data lineage brings a way of tracking data from its origin throughout the whole life cycle taking into account every process that manipulates the data until it reaches its final destination. It is like a telling the story of a piece of data including where does it come from and how it interacts with other data. It should provide answers for questions where the data in given solution come from, whether it can be trusted or not, how it gets from point to point 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 on the subpart coping with databases, data lineage is a general concept where the sources and targets are not necessarily databases, data may come from, let's say, a user interface and ending in an output of a reporting software. This overview of the system, that data lineage provides, is crucial when taking decisions about the infrastructure since the understanding of the consequences should more clear. Also it makes much easier to find errors in systems, since they can be tracked down from where the undesired behavior came to the surface to

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

where the affected data originates. Surely somewhere between these two points the malfunctioning part is and thanks to data lineage the domain of suspicious operations should be reduced and visible. Therefore much time spent on solving issues should be saved. Data lineage is a discipline of *business intelligence*. define

To present data lineage a visual representation is most commonly used. Generally, we can think of the visualization as of a graph explain graph elements.

Having a reference point of interest we can divide data lineage into three types by what 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 and 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 lineage* displays precisely flow of physical data as is in underlying components (eg. applications) of the system is made of.

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 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 the likes of 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 that are used for management of the database we can reliably determine how the data flows once the database is being used.

The idea of data lineage construction is as follows. First precondition is to have access to all metadata related to the database under analysis 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. We identify 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.

an oversimplified example where data lineage would not be much of use as its importance grows with system's complexity.

2.3.1 Manta Flow

Manta flow is a product of Czech startup company MANTA. It is a tool that automatizes data lineage creation by and analysis of programming code. It is able to cope with SQL, altogether with various of its sub-dialects, and Java. Uniqueness of the software 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, IBM IGC etc. where it is integrated.

Our aim to interconnect the component that is subject of this work with Manta Flow to enrich the data lineage that it produces by metadata that can be obtained from relevant data models and can bring better understanding of the system under analysis.

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 types, 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 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 really system works and data move. 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 this lineage by hand. It may be useful at the time when the database is not yet implemented and there is 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 we will bring a data lineage that corresponds to how a database is deployed and used in reality. Then thanks to mapping relations we can propagate the lineage to objects capturing more abstract concepts on conceptual and logical level where the lineage edges will be created by interpolation.

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 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 use *computer-aided software engineering (CASE) tools*. The tools are here to help with development of quality software. The CASE tools are divided into multiple categories, our interest will be focused on the one that deals with Business and Analysis modeling. Graphical modeling tools. E.g., ER modeling, object modeling. The main motivation behind using the tools is that they facilitate creating and previewing data models. Here is an overview of different ways how a data model can be created using them.

3.1.1 Modeling

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 be the result of some process and be based on a model with higher level of abstraction(see the Generating section) and then adjusted or resemble a live database that and to be obtained by reverse-engineering (see the Reverse Engineering section).

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. relationships 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 one on different abstraction level can be derived from it. Modeling tools usually support translating objects to semantically equivalent ones either towards either greater smaller abstraction.

Of course models created like this are not full-featured models but may be a better starting point for a database designer to takeover. For example when conceptual data model is arranged and logical model should be created based on it it is really helpful not to start from a scratch but to generate an outline of the logical one by generating from the conceptual. Then it may be reshaped into 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.

ER/Studio allows creating logical and physical data models.

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

3.3 PowerDesigner

PowerDesigner is a software for data modeling owned by company SAP SE.

The tool supports conceptual, logical and physical data models. The first two are of extended-entity-relationship data model type and physical is relational data model.

4. Analysis & Design of the Solution

The purpose of Metadata Extractor is to obtain metadata from data models created using modeling tools ER/Studio and PowerDesigner. The solution will be able to connect to Manta Flow and to bring data lineage to objects that have greater extent of abstraction than the physical ones which are currently the only objects supported by Manta.

Here we describe how we proceeded when analyzing the solution and discuss crucial features of ER/Studio and PowerDesigner in detail, so we can identify what to focus on, when implementing the tool.

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 data models, how they are organized and what metadata can be obtained that are relevant to be brought into data lineage.
2. Find out what is the format of files that the tools save data models in. Together with how the data we assumed interesting in 1) can be reconstructed.
3. Determine how the file format can be parsed.

4.1 Analysis of the Problem

We already presented that the modeling tools are capable of creating data models. The output files of the 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 do so we needed to identify what objects the solution will look for in the files and how they are related to each other. In Section 2.2 about database modeling we introduced the standard layout of every data model type. We will quickly review the basic skeleton of each model type once again.

In conceptual data model is focused mainly on entities, which may have attributes. An entity may be related to other entities.

On logical layer also entities with attributes can be found and the entities may have relationships.

Physical data models are made of tables. A table belongs to a schema and is composed of columns.

These are the objects we must find in the file formats to recreate the main object hierarchy.

Then we will need to figure out how the models in maps-to relation refer to each other across levels of abstraction. For example how a logical model and a physical model, being the realization of the logical one, are tied together.

4.1.1 File Format

Firstly, we need to identify what is the output of the analyzed modeling tools, to know which what is stored in a file produced, what information are stored there, to decide what is interesting to bring to data flow and what not, and how it is done, to be able to create a software automatically reconstructing the desired information.

ER/Studio

The modeling tool uses its custom file format. It uses .DM1 extension. The file stores plain text and it is made up of many tables. A table in this context is a CSV (comma-separated values) structure but as there is not only one CSV table identifying name is also included. So by table we understand its name, definition of columns (or fields) and records. How more complex objects can be stored in files like this is not clear from the first sight and it required some work to get the idea behind it. insert an image of a table here An attentive reader may find these terms, that we used for describing the data structure, familiar since we already used them where introducing relational databases. And he would be true. When investigating these tables we can notice columns that are shared, meaning relation between the tables that have them in common. This is pretty much how primary and foreign key work in relational databases. Now we face the challenge of reverse-engineering the tables to rebuild the composite objects that are deconstructed and saved in the tables. It would be quite exhausting to try to restore the relationships by hand, so we developed a little tool that helped us to get an overview of the format that will work on one hand with tables' metadata to find the intersecting columns as well as with the records to see whether there are similarities in data stored in different tables to uncover relations between them and the core concepts behind the file format. Further details and ideas related to the ER/Studio file reverse-engineering tool can be found in Section 5.1.1

Surely, when we want to load an arbitrary file a component that parses it will be needed. We mentioned that the file of our interest is basically a sequence of CSV tables. The question was whether to reach out for an existing CSV parser 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.

Once we understand the structure and can tell how the data we are seeking for are stored we can reconstruct them. In one .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 are supporting it. We can imagine why ER/Studio behaves like this. The motivation may be that once there is a problem (if there is no challenge, no data modeling is needed) it is obligatory 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.

PowerDesigner

In the case of PowerDesigner need to handle files with three types of extensions .pdm, .ldm and .cdm. They stand for physical data model, logical data model and conceptual data model respectively. All of them are XML (Extensible Markup Language) based file formats. Actually, there is an alternative to the XML files which PowerDesigner also works with. Each of the models can be saved into a binary version instead as PowerDesigner can process them more quickly. However, we need a user to save his models as XML, only this way is supported currently by Metadata Extractor.

Since we have three different output file types from the modeling tool it is easy to see that the logic of how data models are saved varies from ER/Studio's approach. While ER/Studio groups data models into solutions every model created in PowerDesigner is saved independently. Set of files that are currently opened in PowerDesigner form its state. Such state is called a workspace here and can be saved into a .sws file, but these files do not bring us any interesting information. The information captured stores only what files were at some time opened in the environment and does not tell anything about logical links between the captured files.

When parsing XML files there are basically two major ways we can face the problem.

The first approach is SAX (Simple API for XML) that is an event-driven parser, which process an XML document sequentially by a single pass. By default the processing is state independent and handlers are triggered when an event occurs. It is a simple (for some cases may be even too simple) and lightweight parser.

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

In the next section we describe what we want to retrieve from the PowerDesigner data model files. We will see that the objects and their properties are quite complex and composite using a DOM parser will be 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 with SAX.

4.1.2 Metadata to Collect from Data Models

The main desired feature is to to bring data lineage to the conceptual and logical level. To do this, we had to identify which metadata to collect from models on these levels. The bare minimum is to be able to reconstruct high-level entities to have at least something to visualize data flow between, according to knowledge gained in later stages. But we will aim to bring as much information as possible and try to make use of every relevant (meta)datum saved by a modeling tool. They are exhaustive pieces of software with many features and ability to capture plenty of aspects of modeled system, so we must determine what subset of the information will be extracted. In this section we discuss what specific types of

objects can we obtain and what are means to describe these objects even further by some properties of theirs. On the other hand we will not pay attention to relationships in entity-relationship model. This is given by nature of Manta Flow, it is not a modeling tool thus it does not work with them and links between objects are used solely to represent dependencies determined by data lineage. The only relationship type from data models we will need to cope with is the inheritance relation. It is present in enhanced-entity-relationship models. The exception is made because if the is-a links are not captured, entities' structure are be not described completely and their attributes may be missing. Other categories of metadata we will not extract are the ones that describe some constraints on the actual records saved in database themselves. We will work exclusively with database metadata and don't have access to what is really saved there thus we cannot neither monitor nor enforce anything on the database entries. That is why likes of keys and data types defined in data models will not be in our domain of interest.

Conceptual & Logical Data Model

Here we list objects that appear in both conceptual and logical data models, together with what additional information about them can be inserted by user.

ER/Studio

CDM

ER/Studio does not support conceptual data models.

LDM

justification of each of property or not? & the item lists are too much

- Owner
  Owner is a concept equivalent to a schema - it is a container for logically related entities. Every entity belongs to an owner.
- Entity
  An entity has the following properties
  - Name
    Identification of the entity.
  - Attributes
    Attributes assigned to the entity.
  - Definition
    Further description of the entity. Plain text or RTF (rich text format).
  - Note
    Notes are used when a documentation about the entity is generated. Plain text or RTF.
  - Where Used
    Shows objects that are in maps-to relation with the entity. Those which were created by generating.

  - User-Defined Mappings
    Shows objects that are in maps-to relation with the 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
    Owner is a concept equivalent to a schema - it is a container for logically related entities. The entity belongs to an owner.

- 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
  - Definition
  - Annotation
  - Keywords

- Entities
  - Name
  - Attributes
  - Code
  - Comment

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

Physical Data Model

ER/Studio

- Physical Model
- Type of Data Model (DBMS technology)
- 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 Recreating of Modeled Objects

We defined what are the objects and their properties that we will try to obtain from data models. The objects live in the common environment of a data model, therefore they must be organized in some hierarchy. The above enumeration of the objects may help reader to see that the very basic layout of objects captured by a model is resembling a tree-like structure. The reason is that the basic skeleton of a data model goes like shown on the figure figure. A file can store one or more data models, the models may have several owners, each of them may own zero or more entities/tables which are comprised of none or multiple attributes/columns. Surely further relations between the objects will come to play, like inheritances or mappings discussed later in the Section 4.1.4, making the diagram of actors in the system more complex.

These objects can be seen then as nodes of the tree, whereas their properties are attributes of the corresponding nodes.

Metadata Extractor will build the tree from top to bottom. Firstly, it reconstructs the root standing for a file, then link it to its children, data models, and so on and so forth.

ER/Studio

In the case of ER/Studio, each type of the objects is defined in some table that defines its type as such table is used for storing all instance of the type. It has an id relative to its table used for identification among other realizations of the same type. Basically all links to other objects are done using foreign keys, the only necessity is to know from which tables the keys come from. So if an object has a reference to its, if we stick with the tree terminology, parent we can get it by looking at what is the id being referenced and identifying the reconstructed object using this information and as we are descending down the tree the object is already loaded and we can plug the child in.

PowerDesigner

XML files form a tree structure by definition what makes storing hierarchy of objects with the same nature very much natural and straightforward. This way a parent object of a child is simply its predecessor in layout of XML elements. Other properties of an object are stored as child elements as well or attributes of the object's representation. Also in this case creating our resulting tree structure top-down makes sense. As first, on our way from the XML root, build objects higher in the hierarchy and only if a parent is build we examine its children.

4.1.4 Maps-to Relation

Once we identified objects across the data models it would be handy further explanation to know which ones are related even though they are not defined at the same level of abstraction.

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, since we have relationships available for defining relationships like that. Possibly it could indicate that the objects 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 only the following mappings we will extract:

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

ER/Studio

We already listed two types of mapping relation that entities, tables, attributes and columns in ER/Studio can dispose of. In fact their meaning is the same, the only difference is that the where used mappings are generated automatically and the user-defined are drawn by user. We assumed that all the objects in maps-to relation are in the very same solution but there is also an option to create a mapping to objects that 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 the pairs in sync there is the mapping creation option. We are interested in the first scenario where models may come from two solutions. The compared models' objects are listed side by side and mappings can be created between pairs of them. These mappings are referred to as universal.

Our focus is on how mappings are saved in an ER/Studio solution.

We will look at what is required to do in order to extract the mappings. Let's start with the seemingly easier case of mappings between objects inside the same solution. After some analysis we found a CSV table defining them named Where_Used_PD. In the table there are four crucial attributes namely id_A, id_B, Meta_Table_A, Meta_Table_B. 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 we know to which tables we should look for the keys that are referenced. The meta tables also allows us to check if the objects are actually compatible with each other.

At the first sight solving the universal mappings may appear more difficult as it looks like we will need to search for object in different solution than the one that is analyzed and reconstruct them. But the way it is really solved in ER/Studio is much simpler. We don't need to go anywhere else as the external objects referenced by a mapping are saved in the solution as well. They are described briefly in a table called External_Mapped_Objects by XML structures. Also a table Universal_Mappings using the same concepts as Where_Used_PD allows Metadata Extractor to reconstruct them easily.

PowerDesigner

By the nature of how PowerDesigner saves every data model into a separate file to resolve mappings will be not as straightforward as in the case of ER/Studio. So every mapping we take into account is an external one, using the terminology introduced above. There is a further division into two categories. Similarly as in the first tool, the mappings may be either generated or user-defined.

Before we will go through how they are represented we must mention the way objects taking part in the relation are identified. Every standalone object in PowerDesigner has a unique identifier stored in its attribute named ObjectID. This sequence of characters (string) is used when referring to an object in the XML.

When an object is created out of an existing one it is reflected in the structure of the object's definition. In such case an element History is present where all the ids of objects that made an impact on the objects creation are listed there date as well.

User defined mappings are stored as separate relations. A composite XML element describe one relation the name. The structure is formed by a pair of mapped entities/tables may be empty and if their underlying attributes/column are in maps-to relation as well they are contained as in children elements of the mapping element name.

So we are reading a file where the mappings are defined but we are only able to reconstruct a single object in the relation out of two. The only property of the second one known is its id. We will need to find the object corresponding to the id in the file where it is defined in order to gather all the required metadata about it. Thus in situation like this a data model file are somehow dependent on other(s). To learn about the needed files there is an XML element Targets. When a model is generated from a file a dependency is created in both of the data model files, the one that was generated just like in the one that it was generated from. In other words, if we imagine an oriented graph where a file is a node and an edge leads from file a to file $b \iff b$ is listed as a target file of a, then bidirectional edges are created when models are created by generation. Whereas when user-defined mapping is created from an object in a source model a to an object in a target model b, only the $a \rightarrow b$ edge is created and the b file has no knowledge about the mapping. We must solve how a resolution of the foreign objects will be done. As we have nothing but a target object's id, not even information what file does it from a naïve approach would be hugely inefficient. It would go search for every demanded id across all targets. But that potentially leads to a great amount of file opens as well as having to reconstruct the same objects over and over, leading to a big time overhead. Surely we can improve this solution by collecting the ids and postponing the resolution to the and so when processing a single files its target would be opened only once and the reconstruction of the object would take place one time as well. But that is still expansive in terms of time. If we went in a different direction and processed each of the model once, then stored all the objects that may be referred to and once all the data models on input are loaded, resolve mappings. Logic like this would decreased count of reconstructions and file openings to ideal amount but eventually, if number of inputted data models would be too big the size of memory claimed by Metadata Extractor could become unbearable. To achieve a solution that would have advantages of the both naïve approaches we need to split the set of input files into disjoint subsets that represent the smallest group of logically tied files. We will transform all of the unidirectional edges in the dependency graph described above into bidirectional and will find connected components, that will be our searched logical groups. Therefore we can make the resolution at the end as storage requirements for objects of such subset should be reasonable enough. Based on the assumption that the far most common use-case is having three data models - logical, conceptual and physical (or few physical ones).

But there is one more matter remaining that may cause some problems. The basic scenario how a user will behave is that he will works with PowerDesigner data models in some directory, for example 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, the paths pointing to targets of models became are not correct since they have no reason to be updated and still depend on the files in C:/PowerDesigner/Project/. We want to work only with the files that user explicitly marked as to process, those are only the ones that are present in the input directory. Also if this problem is not thought of, it may cause undesired and unexpected behavior. For example we have file a referring to b in input but a change of external object in C:/PowerDesigner/Project/b would

have affect on a. So we will try to come up with a fallback for this situation and will try to deduce by the former path the one in input folder. As first, we will try the ideal scenario and check whether the target is in the input directory. If yes, we are done with this one. example If not we will assume similar structure of the both directories as well as that the names of files were not changed.

4.1.5 Business Lineage Creation

The ultimate goal of the work is to develop a tool capable of creating business[1] data lineage automatically. We decided to build the high level lineage based on the physical one since it provides the most precise foundation in terms of correctness as it captures the real data movement in a live database. Manta Flow primarily analyzes physical data flow but there is already implemented a functionality which can propagate the lineage to objects that are mapped to the physical objects taking part in it. In our domain by *interpolation* we mean the process of making data flow edges based on information given by lower layer. We must ensure the objects we collect in physical data models are correctly merged as described in Section 4.1.6 to make the most of the interpolation.

an example showing table a to table b, table c to table d and a and c are mapped to entity A, b is mapped to B, d to D. Therefore A has flow to both B and D on upper layer.

4.1.6 Database Connections

Modeling tools usually can make connections to a databases. It is useful for multiple reasons. For example already familiar reverse-engineering would not be possible without this ability since metadata are fetched directly from a database in order to capture its most up-to-date state. Also one of the features of modeling tools is keeping data models and databases that are tied with them in sync, so basically there is a mechanism for comparing actual state of a database with a model.

Manta Flow also extracts metadata of databases for needs of data lineage creation. It is done via JDBC connections.

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 - type of metadata and their organization varies greatly. Instead we 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 it is created by Manta Flow based on the real metadata of physical objects that are present in 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 leading between the

[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

objects. To bring together both features of Manta Flow and our tool, that brings more metadata and links to higher abstraction data models, we need to merge equivalent physical objects which come from the both sources - from Manta's extraction as well as from our tool. So only if we are on the same page and we know what database at which server the modeled objects belong to we 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

is the list complete?
All the above can be stored in one property called *connection string*.
To this set of properties we will refer as a *connection*.
As 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.

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.

we can get type describe somewhere the format precisely

PowerDesigner

In PowerDesigner a user has multiple options for connecting to a database to choose from. Either ODBC or JDBC connection may be used. We would like to have the ability to find out what connections with what parameters were used for connecting to a database corresponding to a physical model. So a great help would be if there was a trace left after every each connection is made. We cannot enforce it but in PowerDesigner these traces can be created when connecting using .dsn or .dcp files. The tool has nice user environment for creating or using connections to a database where a user is guided through set up nicely and can test if he did set up everything correctly.

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 technology to technology.

It has a structure of an .ini files but 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 we have .dcp files. They can store information about native DBMS connection or about JDBC connection. The nice fact about them is that they are not that flexible and once we know whether we deal with native or JDBC respectively we know what exact structure expect. It is also a 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 JDBC connection URL - in other words connection string which should sufficiently define a connection. In case of 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, namely that there is no link between the connection file and a model that is result of reverse-engineering of the connection. So we will need to create a workaround. the workaround

One more solution, which is not native to PowerDesigner but is standard for Manta Flow, is to use auxiliary .ini just like in the ER/Studio case described in Section 4.1.6.

4.1.7 Output Representation

The output of Metadata Extractor will be a graph. Earlier we discussed what are going its nodes and edges stand for. 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 of the output at local machine, in contrary of sending it to the Manta Server where the graph database is.

4.2 Requirements/Desired Features

To summarize the analysis, the overall goal of the developed software is to collect metadata that will allow it recreate physical, logical and conceptual modeled objects with all attributes that may be interesting when shown in data lineage.

The analysis forms a set of functional requirements or features we want Metadata Extractor to have.

- Load objects from data models and reconstruct their hierarchy.
  - 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.
- 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.
- 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[14]. In data lineage diagram business terms can be displayed along with the related data assets to ensure better traceability[15].
  - 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[16]. 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 [17].

- 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. links 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 consists of two major parts where the first one handles ER/Studio models and the second one processes PowerDesigner files. Further division of each of the parts is that they are split into four modules forming a system. Then the most high-level view of the system's architecture is following:

- 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. Independent of Manta

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

[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.

- Reader
  Puts together the model 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 of a model **??**. Communicates 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.

- Manta Flow
  The external part capable of crating data lineage on database objects.

A figure showing cooperation of the most important components

5. Implementation

Precise documentation of the program comes in form of JavaDoc as well. Here we will mention the most important actors concepts or approaches.

5.1 ER/Studio

5.1.1 File Reverse-Engineering Tool

We need to decrypt relations between tables in ER/Studio .DM1 files in order to be able to deconstruct objects stored in the files. We described the file format together with the need for help in section 4.1.1. The task is to find out what links are leading between tables in the format. The links are done via primary and foreign keys.

Input is an arbitrary .DM1 file and output must describe the logical layout. We proposed the parallel with relational databases which can be represented understandably by a relational diagram, we will try to visualize the result showing organization of the format by such diagram. For creation of relational diagram a modeling tools are used. If we are able to generate a SQL code definition of the file format enriched with definition of primary and foreign keys, modeling tools can transform such code into a visual representation of database structure that is creates.

Modeling tools commonly have the ability of reverse engineer a database, however they do it based on metadata from a deployed database where keys are defined. If we just described tables in .DM1 file format by SQL creates, there is no tool that could effectively infer relations. That is why we need this reverse-engineering tool. As the development of such tool is not the focal point of the work, this will be not an out-box general solution for deducing keys for relational databases, but will provide a basic overview of .DM1 file organization. Also it is not a complete solution but is primarily aimed on the objects and properties we picked by analysis described above in subsection 4.1.2.

The decision what programming language should be used for the discussed utility fell to Java as the final Metadata Extractor was going to be written in the language and there are units we want to reuse in this tool as well as in further parts. Although there may be scripting languages that would make some operations, like joins on tables easier. For example in order to go through the file format a parser is needed. Instead of writing it multiple times we will write it once in Java and use here just like in Metadata Extractor itself.

The idea is to look at the tables from two views.

The first one is to take into account only metadata which is represented by the class `DependencyCreator`. It treats column that looks like keys (eg. those which name ends with "ID", the policy is determined by `isOnBlacklist` method) as the same across all tables. The main method here is createDependencies which pairs potential key columns and tables in what they are used. Then there is the class `RelationFinderByTableDefinitions` that allows querying over the structure found by `DependecyCreator` showing what tables are pos-

sibly related through a series of joins. Among other there is the general method `getDependenciesWithPaths` is 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 the data stored in tables as well. When analyzing what are relations between columns, we see a table as a collection of columns, where a column has a set of values. This way we can examine if columns with same names have similar contents. In contrast when getting records from a table we see a collection of rows. The `RelationFinderByTableContents` class is used for the further search for key columns.

It filters tables that without content, thus irrelevant. Has method for determining if a column can be considered a primary key of a table, that is when it contains only distinct ids. Method for finding out if a column can be foreign key corresponding to primary key column, thus the first one contains subset of values for the other. In .DM1 file format there is a table Identity Value that explicitly pairs tables with their primary keys. The class also inspects if the primary keys are defined in strict order or if there is some pattern key definitions follow.

The classes above are used for being able to look from multiple perspectives and to comprehend the format. To put it all together and finally generate the desired output in form of a SQL definition of .DM1 tables in SQL with keys out of the analysis the class `TablesToSQLConverter` is used. The main method is `writeMySQLSource` that at first defines create table procedure for each table from ER/Studio source. Then the phase of creating key constraints takes place. It operates on candidate columns that satisfy policies in `DepedencyCreator` to be key column and are present in at least one table so at least one join can be made using them. There is also space for defining manually the knowledge we gained by inspecting the format using `RelationFinder*` classes by inserting pairs table-primary keys. So primary keys are defined from user-defined information, the information stored in Identity Value table and are inferred based on policies taking into account name of a column and name of the table it belongs to. If a column is set as a primary key in one table for each of the remaining ones that contain it, it is marked as a foreign identifier. In case there are columns that are candidates for being keys but no source table was assigned to them in previous steps one of the tables is chosen as the source, that may cause inaccuracies in the final result but the main message should not get lost.

5.1.2 Parser

Existing CSV parsers are made to process a single CSV structure 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 on a 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 quotes. If a double quote appears in a field, it must be in the enclosed section and the quote itself must be doubled. [18]

We know the structure we need to parse contains tables consisting 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 seems to be easier to come with an own parser for .DM1 files that processes the files into a set of tables identified by their names.

To represent a single table we have 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 due to use of a parser. In the reverse-engineering tool we inspect what values are stored in a column, examine the domain of these values etc. That is the case when a table contents is for us set of columns. Whereas, if we see a table as set of records, each record having properties based on columns of the table, from this perspective a table is a set of rows. The second case is needed when reading the contents instead of inspecting relations between tables and columns.

Getting a name of a table is an easy job as its just a single textual line. To resolve a record is a slightly more challenging task. We designed an automaton that accepts well formed records of our CSV table.

We will 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, and lambda. I is the initial state. Transitions are illustrated in the following figure.

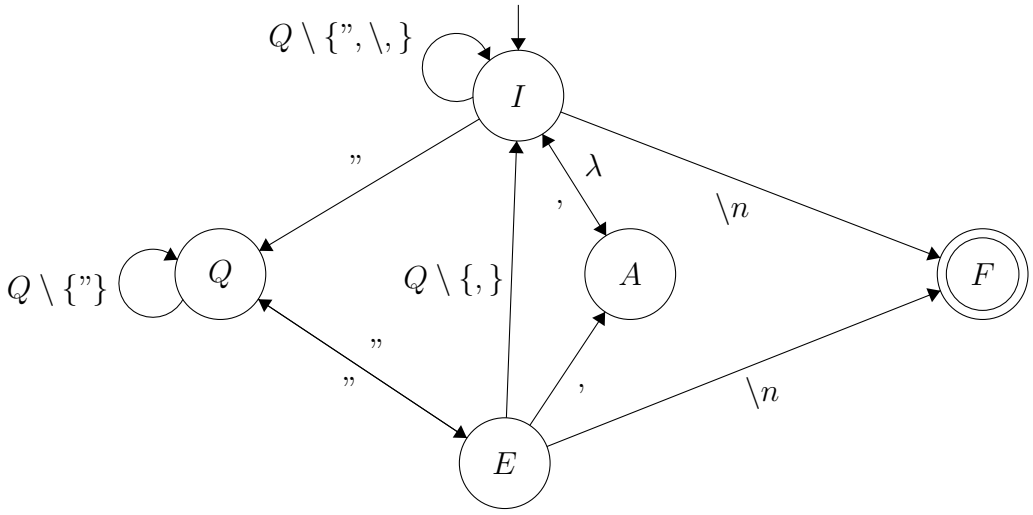

Figure 5.1: Non deterministic finite automaton accepting a line of a CSV table in .DM1 file. In the state A and F, end of a CSV field is recognized and accepted.

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 happens. Once the automaton gets to A, end of a field is indicated. Border 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 unit 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 the file.

The crucial objects and the important properties of theirs result from the analysis of ER/Studio data models listed in subsection 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.

`SimpleDataObject` is used by 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 only based on to which data model, physical or logical, it belongs, 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 objects fulfilling the contracts specified by interfaces defined in the model.

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 comparison to the interfaces from the model, 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 data structures is how to gather the 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 follow, `CompositeObject`s 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.

To put it all together, `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.

a class diagram?

5.1.5 Reader

`ErStudioDm1Reader` crawls through directory with input .DM1 files processing them calling the parser on each of the files. Parser's product is handed to resolver resulting in `ErStudioFileModel` that is output of the unit.

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`.

Once the node representation for all objects is created, it is the right moment to create mapping edges between them. As we create the mappings symmetrically, the full information is captured on both levels. This way, we are able to create all the mapping edges by going through all objects from a single level of abstraction, no matter which level though.

5.2 PowerDesigner

5.2.1 Parser

For parsing PowerDesigner data model files Manta's XML DOM reader is used.

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 fac-

tory 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 `PowerDesignerXmlComponentReade` 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.

5.4 Technologies

- 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

5.5.1 ER/Studio

Parser

Resolver

Data Flow Generator

5.5.2 PowerDesigner

Resolver

Data Flow Generator

Conclusion

We have developed a piece of software capable of extracting metadata from two data modeling tools - SAP PowerDesigner and ER/Studio Data Architect. To achieve this a thorough analysis was required to identify relevant features of the tools, 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] Sap powerdesigner documentation.

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

[15] Lineage vs traceability understanding the differences. https://www.collibra.com/blog/lineage-vs-traceability-understanding-the-differences/.

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

[17] 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.

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

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

A.2 User Documentation

A.2.1 Tutorials

A.3 Cooperation with Manta Flow

A.4 Full List of Modeling Tools Properties

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
    By owner assigned to an entity permissions are realized.
  - Keys
    As we work exclusively with metadata we cannot enforce key constraints on the actual records in database, thus there is no need for them.
  - Relationships
    Manta Flow does not support them.

  - Constraints
    The same reason as with the keys above.
  - Naming Standards
    The property is used when creating/generating data models but as we don't modify or add anything that would need to apply naming convention to, it is irrelevant for our case.
  - Data Lineage
    We already mentioned that we want to create a real data lineage not just something that was drawn by user because it could have nothing to do with how the data actually flows.
  - Security Information
    There would not be much of use if extracting security information since they are currently not supported in Manta Flow.
  - Attachment Bindings
    Manta Flow does not support attaching external pieces of information like ... to objects.

- 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

    Technical properties, many of them are effective only on some specific technologies (Organization adjusts only behavior of Netazza tables). describe every?
  - Storage
    Specifies storage option
  - Dimensions
  - Properties
  - DDL
    Code to create the table.
  - 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