Department of Informatics, University of Zürich

**BSc Thesis**

# Data Lineage and Meta Data Analysis in Data Warehouse Environments

Martin Noack

Matrikelnummer: 09-222-232

Email: `martin.noack@uzh.ch`

January 31, 2013

supervised by Prof. Dr. M. Böhlen, D. Christopeit and C. Jossen

University of Zurich UZH

Department of Informatics

## Abstract

This thesis aims to provide new insights on *data lineage* computations within the Credit Suisse data warehouse environment. We propose a system to compute the lineage, targeted at business users without technical knowledge about IT systems. Therefore we provide complete abstraction for end users. Furthermore, we process only conceptual *mapping* rules from the metadata warehouse, in contrast to other approaches which record transformations at runtime, and consequently do not rely on access to potentially sensitive data. In order to process mapping rules, we developed an algorithm that is capable of extracting components generically, based on their semantic meaning and relation to each other. This thesis describes some patterns in lineage investigations that result from our approach and gives an outlook to future projects that could be based on this work.

# Zusammenfassung

Diese Arbeit erläutert unsere Erkenntnisse bezüglich der Berechnung von *Data Lineage* im Bereich des Data Warehouse der Credit Suisse. Hierzu stellen wir eine Methode vor, um Data Lineage für Business Anwender zu errechnen, welche keine tieferen Kenntnisse bezüglich der IT Systeme besitzen. Dem Benutzer wird mittels des vorgestellten Ansatzes vollständige Transparenz und Abstraktion vom technischen Problem ermöglicht. Wir benutzen lediglich die konzeptionellen Abbildungsvorschriften in *Mappings* aus dem Metadata Warehouse, wohingegen andere Ansätze die Abbildungen zur Laufzeit aufzeichnen. Daher sind wir nicht auf den Zugriff auf potentiell vertrauliche Daten angewiesen.
Um die Mapping Regeln auszuwerten, benutzen wir einen Algorithmus, der generisch Komponenten aus Mappings ausliest. Dabei werden sowohl die semantische Bedeutung der Komponenten, als auch deren Beziehung zueinander berücksichtigt. Weiterhin beschreiben wir in dieser Arbeit Muster in der Lineage Berechnung, die sich aus unserem Ansatz ergeben, und bieten ein Ausblick auf zukünftige Projekte basierend auf unseren Erkenntnissen.

# Contents

# List of Figures

# List of Tables

# 1. Introduction

One of the core competencies of a a bank is the ability to maintain and efficiently utilize a data processing infrastructure [Lle96]. Especially analyzing large data sets to make market-based decisions or create a risk assessment for certain events provides valuable advantages [NB04]. As a middle-ware layer between transactional application and those decision supporting applications, data warehouse systems are the technology of choice [Win01]. They serve as means to decouple systems used for the support of decision making from the business transactions. But by using those systems, data from different sources is quickly mixed and merged. In several steps complex computations are performed and results re-used in other steps. That makes it hard to keep track of distinct sources and influences during the numerous transformations that are necessary to compute the final result [EPV10], even though knowing the derivation history of data can be quite useful in many cases. For instance finding out about the trust-level attributed to certain sources and then assessing the reliability of the result is often very important [MM12]. Moreover, providing an audit-trail is common practice in many financial applications and has to be kept in great detail. In the simplest case, the user might just want to see and understand how exactly the result was created and which data items were used in the process.
The information about history and origins of data is called data provenance, or also data lineage [BKWC01]. In this thesis, we will address the issue of computing and analyzing the data lineage in data warehouse environments. In contrast to other approaches, we will not trace executed transformations, but use the conceptual logic behind single transformation rules to compute the flow of data independent from the technical implementation. This enables us to follow the lineage of data across various systems and implementations.
The transformation rules that we use are therefore not concrete implementations, such as SQL statements, but conceptual specifications of rules. They define on a conceptual level how data is transformed and moved along the system. Those rules are stored within the existing Credit Suisse metadata warehouse and called *mappings* [JBM$^+$12]. By following a sequence of mappings in reverse order, we can therefore find every source that may contribute to a given result according to the utilized sources in each mapping. In some cases, a sequence of mappings completely excludes data items from one or more sources by applying exclusive filters in different mappings. To recognize those restrictions, we process the conditional logic that is incorporated in every mapping and evaluate the combined conditions of the mapping sequence.
Consequently we propose a system to search for the source which contains the original data where a given result is effectively based on. We call this the *golden source* and the computation process *active lineage*. However, the provided approach allows for many possible extensions in different directions, such as impact analysis or optimization of mappings.

The structure of this thesis is as follows: we will first describe the practical use case for our application in section 2. Then we define the scope of the thesis and describe the concrete task in chapter 3. To give the reader an idea of alternative solutions an how our approach compares to those, we will present the most common concepts in section 4. After presenting the existing elements on which we base our work in section 5, we will then explain the lineage computation in our system in section 6 and describe the developed algorithm in great detail. That is followed by an outline of common patterns that emerge from the proposed technique and what conclusions we can draw from them. Chapter 8 illustrates how the devised prototype implements the presented algorithm. In the last chapter 9 we will then give an outlook on future extensions of the algorithm based on its current limitations.

# 2. Use Case Description

As explained in chapter 1, lineage information describes the origins and the history of data. That information can be used in many scientific and industrial environments. A well documented example of the importance of lineage information are tasks using SQL queries [MM12]. Here, the users often wants to find out why a particular row is part of the result or why it is missing. Those questions can usually be answered without much effort, since the user has direct access to and knowledge about the used queries.
While we will focus on a business related application, the usage of data lineage is of course not limited to those. Scientists in several fields such as biology, chemistry or physics often work with so-called curated databases to store data [BCTV08]. In those databases, data is heavily cross-connected and results originate from a sequence of transformations, not dissimilar to our application. Here it is vital for researchers to obtain knowledge about the origin of data. Depending on the source, they can then find the original, pure datasets and get a picture of their reliability and quality.

In the scope of this thesis, we primarily consider the internal processes in an internationally operating bank. Here exist many data sources that are part of different business divisions, such as Private Banking or Investment Banking. All of them serve different purposes and are maintained from different organizational units. Now, to create a reliable analysis or report as a basis for important decisions on a global scale, a system to support this has to take into account data from a potentially very large number of sources. In order to provide the expected result, data items may undergo a sequence of transformations over several layers in the system and are mixed or recombined [JBM$^+$12]. We call such a transformation *mapping*, where a sequence of mappings defines a *data flow*, i.e. the path that data items take in the system, from their respective source to the final result. The full concept of those mappings and the implications will be explained in chapter 5.1. When the responsible user finally gets access to the report, there is practically no way for the human reader to grasp the complete data flow and its implications on single data items. Usually that would not be necessary for a business user who is only the initiator of the process and receiver of the result. But what if the user discovers a discrepancy in the generated data values? She might be able to verify the result or identify a possible error if she was presented with the details, since she is familiar with the purpose and operational origin of the data. Yet she can not be expected to have knowledge about the system on a technical level, especially not about the physical implementation of mappings. We therefore aim to provide a system that enables the user to gather lineage information about a given data item independent from her technical skills and knowledge. In order to built such a system, there are a number of issues to be addressed.
For example not every user has access to every part of the system. Especially in the financial sector where sensitive customer data is very common, there exist very strict regulations on

data access. For this reason we need to rely on the mapping rules instead of actually recording the transformations, where access might be required that is not granted for ever user.
Additionally we need to represent both the results and potential user input in easily understandable form. There is no gain in providing the user with yet another abstract interface or even special query language if we want to make the system accessible to a broad spectrum of business users. We solved this issue by supplying a verbose syntax based on a well defined grammar as basis for representations, which will be explained in section 5.2.
To utilize existing mapping data from the current system, we have to face another challenge. Credit Suisse is successfully running a metadata warehouse [JBM$^+$12], hence we build our system on the available infrastructure and model. This has an major impact on the design since the system deals not with flat files or relational databases, but uses the RDF (Resource Description Framework) data model. In this implementation data resides in graph structures instead of a traditional flat representation. Yet we ignore the specifics of RDF, since the general approach is unchanged for data in any type of graph, independent of the actual storage technology. We generally assume the set of all mappings to form a connected graph and data items to follow paths in this graph according to mapping rules.
In order to provide the promised level of abstraction to the user, we need to automate every step of the computational process. We decided to modularize those steps so that similar requests can be computed with only small changes to the algorithm without affecting the rest of the system. Similar challenges might include impact analysis, where the user is not interested in the origins of a result, but in which results the original data is used. This could be the continuation of the above example, where the user spots an error in a data source. She might be interested where else the error-prone data was utilized in order to learn about other affected computations. Or from a more technical perspective, an IT user could want to speed up the system by optimization. Finding loops of dependencies or dead-ends in data-flows might help to reduce the complexity of the system.

# 3. Task

From the use case specification in chapter 2 we can take away that there is a high value in lineage computations in a company's data system. Following that, the main goal of this thesis is to provide new insights into the matter of data lineage in the context of data warehousing within Credit Suisse. In this chapter we will provide an outline of the scope of the task that is given for this thesis and then show the power of the resulting lineage computations.

## 3.1. Description

As mentioned in chapter 2, we built our system on the existing metadata warehouse infrastructure and model of Credit Suisse. Accordingly, we assume the data sets to form a directed graph, where nodes are sources or (intermediate) results and edges symbolize mapping transformations of data. In this context, data is moved along a number of paths through the graph, starting on each involved source and ending at the final result. To determine all the original sources of a given data item, we need to traverse those paths backwards along the graph until we reach the starting points of the sequence of applied transformations.
However, traversing the graph simply in a brute-force manner might result in an exponentially growing number of potential paths to consider. Therefore we try to reduce the number of followed paths by evaluating the logic that is incorporated in every mapping. We call this the active lineage that dynamically processes the transformation logic behind every step, in contrast to the passive lineage where paths are considered only statically. Detailed explanation of this distinction follows in chapter 3.2.
Following this, we are not only observing isolated mapping rules but the sequence of mappings and can make assumptions about their combined impact. For example mappings with contradicting conditions on data processing are individually seen both valid and relevant for lineage computation, but applied in sequence may exhaustively restrict the conveyance of data items. This potentially allows to reduce the number of followed paths significantly and gives us the chance to determine the lineage of data even more accurately by excluding false positives. Figure 3.1 illustrates the potential benefit of active lineage for theoretical data flows. Assuming we are interested in the lineage of items in H, the evaluated paths and resulting golden sources are painted red. In the case that both sequences of mappings $e_2 \circ b_2$ and $e_2 \circ c_2$ could be excluded based on the logic in the respective mappings, source B can be identified as a false positive. Passive lineage, however, evaluates only the static data flows and therefore includes B as a golden source.

The system to be built uses the already existing rules about the structure of mappings, that are given in the form of a context-free grammar. By using those grammar rules, it is possible to generate a generic system that filters the path according to certain criteria such as particular value ranges or attributes. Those filters may be user specified or a result of active lineage computations. To achieve that, there are two basic steps necessary:

1. Extract logic and meta information from mappings
2. Traverse the graph according to those results

We will explain both steps in great detail and then take a look at common patterns that occur during graph traversal based on the results. Additionally we will present a prototype that implements the results of our considerations as an empirical proof of concept. Finally there is an outlook on the possible extensions of the system as well as its current limits, especially in regard to the developed algorithm and the underlying system.

## 3.2. Lineage Specification

In order to explain our solution for the given task, we first need to elaborate on the possible degrees of lineage computations. We distinguish two types of lineage for our purposes: the *passive lineage* and *active lineage*. What we want to offer to the user is the more advanced active lineage. In chapter 4 we will compare those two forms of lineage to other common definitions.

### Passive Lineage

Passive lineage denotes the more basic, static computation of data lineage. It finds all potential sources of data items for a sequence of transformations. However, it does not consider further conditions or constraints that are given for data flows and therefore includes sources that effectively do not contribute to the result. For example data flows that are restricted through exclusive transformation rules may still be listed in passive lineage, even if no data item reaches the result table.

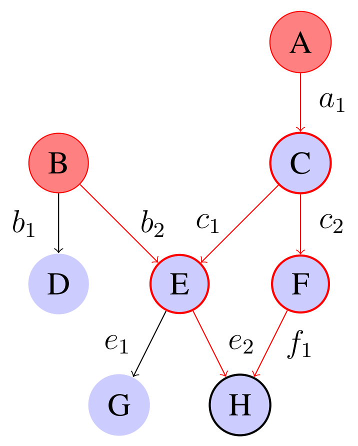
Let us consider the given database in 3.2 where we apply to both tables query $Q$:

$Q$ : SELECT b.X, a.Z FROM A a, B b WHERE a.X = b.X AND b.V = $v_1$

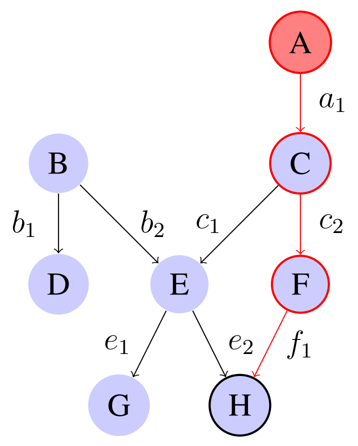
Now we are interested in the passive lineage of the result. For this it is sufficient to find the exact location from where data items were copied. Hence we look at $Q$ and find that the SELECT statement reads both A and B but returns only data from A. This is apparent from the fragment "a.X, a.Z". Therefore we identify the passive lineage to be $\{A_1\}$.
Additionally we consider another query $Q_2$ which produces the same result but by different means:

$Q_2$ : SELECT b.X, a.Z FROM A a, B b WHERE a.X = b.X AND b.V = $v_1$

The passive lineage is now $\{A_1, B_1\}$.



(a) Passive lineage



(b) Potential active lineage

Figure 3.1.: Active lineage may reduce the lineage paths (red) for data items in H.

## Active Lineage

In contrast, active lineage actively computes the conditions that need to be fulfilled in order to include a source in the lineage. This potentially reduces the number of sources, for example if an accumulated set of conditions along the way evaluates to *false* for a particular source. In the end, active lineage is effectively the passive lineage with evaluated conditions on the data flow. Let us consider the given database in figure 3.2 and again apply query $Q$, then the active lineage results in $\{A_1, B_1\}$ with the attached condition $a.X = b.X$ AND $b.V = v_1$. We call such an effective source of data a *golden source*.

**Definition 1** *While a source may be in general both target and data source for transformations, a* **golden source** *is a source that serves only as data source and is not targeted by any transformation.*

This makes the golden source the storage location of some or all of the original data items which were used in the computation of the given result. It does however not include sources that occur solely in conditional parts of the transformation.
Furthermore a source is only a golden source if its data items may in fact be part of the result and are not logically excluded by a combination of transformation steps beforehand. However, a golden source is not determined by the actual values of data items. An empty source may still be a golden source.

A

| | X | Y | Z |
|---|---|---|---|
| $A_1$: | $x_1$ | $y_1$ | $z_1$ |
| $A_2$: | $x_2$ | $y_2$ | $z_2$ |

B

| | X | U | V |
|---|---|---|---|
| $B_1$: | $x_1$ | $u_1$ | $v_1$ |
| $B_2$: | $x_1$ | $u_2$ | $v_1$ |
| $B_3$: | $x_3$ | $u_3$ | $v_3$ |

Result of $Q$

| X | Z |
|---|---|
| $x_1$ | $z_1$ |

Figure 3.2.: Example database for provenance considerations

# 4. Related Work

The following sections present the most common categorizations used in other works. We provide the reader an overview of the scope of similar procedures and how our approach, as outlined in chapter 3, relates to those.
While there exist many different ideas on how to solve the data lineage question, we go with a new and innovative approach. As described in the use case, the ultimate goal is to offer a service for users who have little or no knowledge in the technical area, but are interested in the lineage of data for further analysis. This makes a huge difference to other approaches, where the user may be required to understand the process in order to produce results. For instance Karvounarakis et al. use a custom query language in order to pose requests to the system [KIT10]. Glavic et al. designed their provenance system *Perm* as an extension of the PostgreSQL DBMS and expect the user to execute SQL queries for provenance information [GA09].
Furthermore most of the existing lineage algorithms work on the assumption that the actual content of the database is known and available. They trace or record real transformations by running them, like Cui et al. do in their work on lineage tracing in data warehouse transformations [CW03]. But we have only access to structural information about transformations, not the transformed data. Therefore we do not only record the lineage at run-time, but evaluate the computational steps based on the structural information.

Regarding common terminology, it can be noted that many works use data provenance as synonym to data lineage [Tan07] or also pedigree [GD07, BT07]. Moreover it is a common approach to define three major categorizations for provenance information. Those are traditionally why-, and where-provenance [BKWC01] as well as how-provenance in later works [CCT09, GM11]. Every category aims to find out a specific form of provenance. In order to classify our approach we will shortly describe the main differences between each of those categories.

### Why-Provenance

The search for why-provenance leads to the *contributing source*, or *witnesses basis* as others call it [CCT09, GD07]. Why-provenance captures all source data that had some influence on the existence of the examined data item. Intuitively it explains *why* any data item is part of the result. This does not only include the actual data that is present in the result, but also data items that have to be used in order to select the final data. But even though many data items might be part of the computation (the set of witnesses), why-provenance denotes only the minimal set of data items that is necessary to produce the result (the witness basis). A simple example would be the following scenario: Take again two tables A and B as shown in figure 3.2.

For a given query like

$$Q : \text{SELECT a.X, a.Z FROM A a, B b WHERE a.X = b.X AND b.V} = v_1$$

we get the result that is presented in the figure. Now we can immediately see that the origin of the result is solely table A, more precisely $A_1$. But according to the given explanation, we have to include both $B_1$ and $B_2$ into our consideration, since they were used by the query. However, there is no need for both of them to exist in order to derive the same result. Either $B_1$ or $B_2$ would suffice. Therefore the witness basis, i.e. why-provenance of $C_1$ can be expressed as the set $\{\{A_1, B_1\}, \{A_1, B_2)\}\}$. For our goal to provide active lineage, we apply some of the same operations that are also necessary to compute why-provenance. But instead of simply stating the why-provenance, we evaluate it after each step and draw further conclusions regarding relevance of sources. This concept is used in the preservation of conditions (chapter 6.3.1).

### Where-Provenance

In contrast to why-provenance the where-provenance, also called *original source*, focuses on the concrete origin of data [GD07]. It respectively answers the question *where* exactly data items come from, regardless of the reasons why they are part of the result. This relates strongly to our definition of passive lineage. While we need to compute the while provenance as a part of active lineage, it is not sufficient. We basically evaluate the why-provenance for each item in the where-provenance and then remove non-relevant items from the result.

### How-Provenance

The concept of how-provenance is relatively new compared to the other two provenance types. It was introduced for the first time in 2007 by Green et al. [GKT07]. Following the nomenclature of why- and where-provenance, how-provenance answers the questions *how* the result was produced and *how* data items contribute. It is easily confused with why-provenance, but goes one step further.
In order to record why-provenance, most systems use a special notation. In order to keep the introduction short, we relinquish the in-depth explanation of such a notation as it can be found in [GKT07] and [CCT09]. But by using the given example from figure 3.2 and again applying query $Q_2$, we can also derive the (in this case) very simple how-provenance. The result is a combination of two tuples $A_1 \wedge B_1$, effectively preserving the computational expression that was used. By following such provenance information through the set of transformations, we get a picture not only where data items come from, why they are part of the result but also how exactly they were computed in each step. This is definitely the most advanced and thorough type of provenance. For our purposes however this yields no advantage over active provenance. It is not important for us how the data was computed, only the fact that the computation allows for its contribution to the result.

To summarize, our goal is ultimately to compute active lineage in contrast to other systems that process only one of the previously presented provenance categories. We do not only follow along the lineage path, but also use the gathered information to dynamically decide which paths and sources are effectively relevant for the lineage. Furthermore we want to provide the user with complete abstraction from the processing, which distinguishes our work from most of the other systems that require advanced user-interaction.

# 5. Approach

Now that we have outlined the task and how we want to approach it, we first need to introduce some of the tools that we have defined before we can start in-depth explanations of the developed algorithm in chapter 6. We will therefore address the definition of mappings as mentioned in chapter 2, as well as the grammar and the generic graph representations.

## 5.1. Mapping

The basic element that we use in this work is a mapping. While the meaning of "mapping" can differ depending on the context, we follow the definition used by Credit Suisse, which is also incorporated in existing works:

**Definition 2** *A* **Mapping** *is a set of components that are used to describe the creation of rows in a target table. They contain descriptions of rules and transformations that have to be applied in order to move data from one area to another one in the required form.*

It is basically used for three purposes:

1. To specify the elements that are necessary to create rows in a given database table (e.g. other tables, columns, values).
2. To specify how to manipulate and transform those elements under certain conditions.
3. To document the whole process.

A concrete example of such a mapping is given in figure 5.1. In accordance to the definition, it describes the mapping of columns from table *CUSTOMER_MASTER_DATA* to table *AGREEMENT* and provides instructions on how to transform elements under different circumstances.
Mappings are a very powerful tool to describe those transformations. They include not only data flows inside of single databases, such that pure SQL statements could cover. Mappings can specify how data is moved from database to database within the data-warehouse, or how to populate a table from flat files based on a given file structure. The mapping description as part of the metadata is not concerned with concrete implementations and therefore can express the complete data flow on a conceptual level.

There exist various mapping components that can be part of a mapping. They include for example elements such as *Entity*, *Attribute*, *Entity Attribute Population*, *Entity Attribute Population Condition Component* and many more. Every component is represented in the

**WHEN POPULATING** *Entity: AGREEMENT*
**FROM** *Entity: CUSTOMER_MASTER_DATA*

**POPULATE** *Attribute: AGREEMENT.ACCOUNT_CLOSE_DT*
**WITH** *CUSTOMER_MASTER_DATA.LAST_ACCOUNT_CLOSING_DATE* **IF**
    *CUSTOMER_MASTER_DATA.LAST_ACCOUNT_CLOSING_DATE != 01.01.0001*
**WITH** *"DEFAULT MAX"*

**POPULATE** *Attribute: AGREEMENT.ACCOUNT_FIRST_RELATIONSHIP_DT*
**WITH** *CUSTOMER_MASTER_DATA.FIRST_ACCOUNT_OPENING_DATE* **IF**
    *CUSTOMER_MASTER_DATA.FIRST_ACCOUNT_OPENING_DATE != 01.01.0001*
**WITH** *"DEFAULT MIN"*

**POPULATE** *Attribute: AGREEMENT.ACCOUNT_OPEN_DT*
**WITH** *CUSTOMER_MASTER_DATA.CIF_OPENING_DATE* **IF**
    *CUSTOMER_MASTER_DATA.CIF_OPENING_DATE != 01.01.0001*
**WITH** *"DEFAULT UNKNOWN"*

Figure 5.1.: Real world example of a mapping from *CUSTOMER_MASTER_DATA* to *AGREEMENT*.

conceptual mapping data model, designed by Credit Suisse experts. The **mapping data model** "describes the data required to document the population of Entities (Feeder Files, Tables, etc) and to provide a means for reporting data lineage" [Bud12]. This model includes every element that is needed to express any mapping within the given system. That means that every mapping can be constructed using only components from the model. A simplified excerpt is shown in figure 5.2 (see appendix A for the full model). It can be seen as a blueprint for mappings and therefore served as an important basis for our work.
The mapping data model describes data sources and targets of mappings as entities, which are in our case tables in the underlying database system. Similarly the entity's attributes are columns of the table. Each of those entities can potentially be both source and target. Therefore we call those entities generally *source* (of a mapping) and refer to them as *target* only in the explicit case for a specific transformation to avoid confusion. However, we use *entity* and *attribute* when describing more conceptual ideas.

## 5.2. Grammar

However, instead of working with the data model from section 5.1, we utilze the existing formal context-free grammar (see figure 5.3) that is based on the model. It describes the structure of any mapping in accordance to the mapping data model and includes all elements that were specified there. Consequently it is it sufficient to express every possible composition of mapping components in the given syntax (see appendix B for the full grammar). Mappings

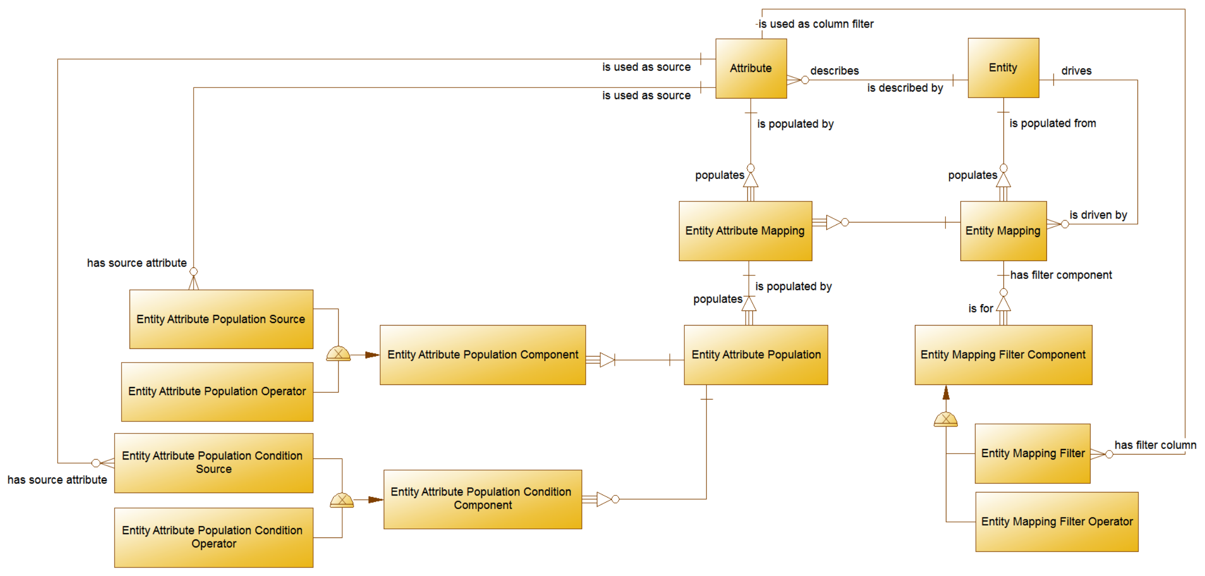

is used as column filter
is used as source
is used as source
Attribute
describes
is described by
Entity
drives
is populated by
is populated from
populates
populates
Entity Attribute Mapping
Entity Mapping
is driven by
has source attribute
is populated by
has filter component
populates
is for
Entity Attribute Population Source
Entity Attribute Population Component
Entity Attribute Population
Entity Mapping Filter Component
Entity Attribute Population Operator
Entity Attribute Population Condition Source
has source attribute
Entity Mapping Filter
has filter column
Entity Attribute Population Condition Component
Entity Attribute Population Condition Operator
Entity Mapping Filter Operator

Figure 5.2.: Excerpt from the conceptual data model for mappings

written in this form are meant to be human readable and therefore more verbose than technically necessary. A user should be able to understand the meaning of the mapping without further reference material. At the same time they can be processed by a computer, since they are structured according to the given grammar rules, unlike natural language. Figure 5.1 from the last section is written according to this syntax.
In order to use the grammar for our purposes, it was expedient to change the structure of some production rules so that we can represent these rules in a tree structure. This is far more convenient than a relational representation, as we will explain in section 5.3 in more detail. From this it follows that the grammar is not only used for its traditional purpose only, namely parsing input, since we plan to use already well-structured data. Mainly it is used as a basis for further processing of the mapping, independent from the technical implementation of any input.

| | | |
|---|---|---|
| mapping | ::= | population filter? navigation? |
| population | ::= | WHEN POPULATING targetEntity FROM drivingEntity populationClause |
| populationClause | ::= | (POPULATE populationComp)* |
| populationComp | ::= | attribute withClause+ |
| withClause | ::= | WITH expression (IF condition)? |
| expression | ::= | string \| computation \| concatenation |

Figure 5.3.: Excerpt from the used grammar

## 5.3. Grammar And Model In Comparison

To explain why we decided to use the grammar as a basis for our computations, we now compare the grammar with the model to show similarities and argue how the grammar is superior for our purposes.

First of all, the grammar is suited to give a human reader, who has no knowledge of technical details, insight on the content of mappings. It defines a syntax and allows a human reader to deduce the purpose of a component based on its written context. For example does the mapping always include exactly one main source and one target entity. The data model in figure 5.2 represents this with two relations between *Entity* and *Entity Mapping*. An *Entity Mapping* "is driven" by exactly one *Entity*, which makes that the declared main source for the mapping. Similarly exactly one *Entity* "is populated from" the *Entity Mapping*, which makes this *Entity* the target. But given only the relational tables, it is difficult for a human to derive which is a source or a target. It is necessary to follow the respective keys in the physical mapping table and therefore to have technical knowledge about the concept of databases. The grammar, however, attaches semantic meaning to each element with the syntax and provides needed context regardless of any technical implementation. Considering a mapping such as in figure 5.1, let us examine at the first lines:

**WHEN POPULATING** *Entity: AGREEMENT*
**FROM** *Entity: CUSTOMER_MASTER_DATA*

By comparing this to the rule "population" in the given grammar from figure 5.3, we can immediately identify the main source as *CUSTOMER_MASTER_DATA* and the target table as *AGREEMENT*.

Now one could argue that apart from readability both model and grammar are equal in power and coverage and therefore we could also use the model for our lineage computations. However, the latter is a far more suitable tool for our purposes. The reasoning for that is the following:
It is definitely possible to describe a mapping with the data model as well as with the grammar. Both define the structure of the mapping and determine how to compose the elements. However, the grammar is built hierarchically, whereas the model is flat, as it is meant to be implemented in a database system and can be directly converted into a relational table schema. This gives the grammar an advantage since we want to work on graph structures, as has been stated in chapter 2. A mapping can therefore be extracted directly from the underlying mapping graph in a graph representation, in contrast to a relational representation.
We gain then further advantages over a relational model when processing mappings in tree form. For example recursive rules, such as groupings or computations that are part of other computations can be traversed easily in a hierarchical representation, but on a relational schema this would lead to multiple self-joins of potentially very large tables.
Additionally, the grammar allows to easily validate given mappings. Parsing an error prone input is immediately noticed, whereas the population of tables may include many unknown

mistakes. Although this does concern the input of mappings more than the lineage computations, it played an important role in the decision to devise a grammar. Since therefore all mappings are in a form compatible to the grammar, we can perfectly use it to describe the general structure of mappings.

To conclude this argumentation, we can summarize that the model is perfect to represent data in a relational schema that is only accessed by computers. However, once human interaction is necessary, the grammar offers superior comprehensibility. Additionally, the hierarchical structure offers easier integration into the graph based system on which we are building our solution.

## 5.4. Utilization Of The Grammar

After we have shown that it is beneficial to use the grammar as a basis for our system, we now explain how we utilize the grammar.
For one, the grammar defines how concrete instances of mapping trees are structured. The important requirement we set for those tree representations is that no semantic information may be lost during tree construction, it must be possible to unambiguously recreate the mapping. For instance it is necessary to distinguish between source table and target table, not just to include two equal tables. Therefore we include *targetEntity* and *drivingEntity*, as figure 5.3 illustrates. We also need to preserve the mathematical computations according to the order of operation rules instead of order of appearance and similar information. Our grammar fulfills this requirement since it is completely compatible to the data model, as we explained in section 5.2. For a given mapping as in figure 5.4, the corresponding tree representation according to the grammar is shown in figure 5.5.

Consequently the grammar rules enable us to create a "map" of all components that may be part of a mapping. Looking at the grammar, we can immediately see where attributes occur or where computations may be used. Without this map, we would have to rely on actual instances of data to get a grasp of the structure on a per-mapping basis. However, we want to present an algorithm that works universally on any mapping. Therefore we can not rely on concrete data items, but have to work with the abstract mapping structure and then apply the resulting strategy to the concrete instances. Once this structure is known, we can determine all possible compositions of mappings beforehand and define algorithms universally for every mapping in the system.
The great advantage in this approach lies in the opportunity to use this to search not only for elements by occurrence but also by semantic meaning. We are for instance able to look specifically for the target table instead of any occurrence of a table, since we can exactly determine how and where it is specified in a mapping.

**WHEN POPULATING** *Entity: AGREEMENT*
**FROM** *Entity: CUSTOMER_MASTER_DATA*

**POPULATE** *Attribute: AGREEMENT.ACCOUNT_CLOSE_DT*
**WITH** *CUSTOMER_MASTER_DATA.LAST_ACCOUNT_CLOSING_DATE* **IF**
    *CUSTOMER_MASTER_DATA.LAST_ACCOUNT_CLOSING_DATE != 01.01.0001*
**WITH** *"DEFAULT MAX"*

Figure 5.4.: Simple example mapping

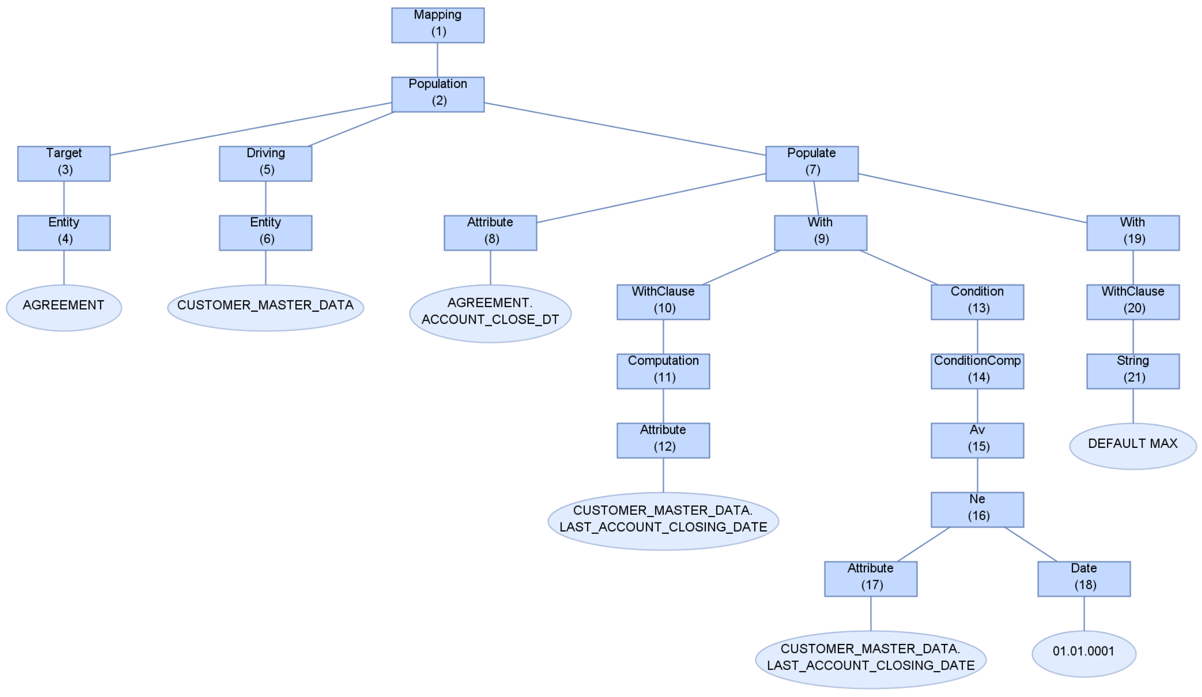


Figure 5.5.: Parse tree built from a simple mapping. Nodes are represented as squares, the content of leaf nodes is attached as an oval shape. Numbers in brackets denote the Id of each node for quicker reference.

# 6. Lineage Computation

Having explained the concepts that we use in our system, we can now commence with the in-depth explanation of the algorithm.
In order to answer a given question such as the request for the golden sources, we generally have to convert the input into a set of computations. For this we must address two basic aspects. Firstly it is necessary to follow the paths through the system and therefore to navigate from one intermediate source to the next by selecting all appropriate mappings, as we will describe in section 6.1. This is a high level process and can be expressed by a query over the set of all mappings. However, to identify those relevant mappings, we need to retrieve information from each mapping separately. Using those results is a more complex and requires different data depending on the posed request. This will be addressed in section 6.2. Secondly, to avoid false positives, it is necessary to preserve possibly stronger constraints in population conditions after each step. Section 6.3.1 will explain this aspect in more detail. It should be noted that by dividing the algorithm into several modular steps, we offer the basis for other investigations that were mentioned in chapter 2, such as impact analysis. Depending on the type of the request it is possible to retrieve different information from each mapping with a generic query. The affected parts of the algorithm can then be adapted for different situations, while the remaining implementation stays intact. However, we will focus purely on the data lineage aspects in this thesis, as it has been stated in the task description in chapter 3.

## 6.1. Moving Along The Graph

We already explained in section 3 that the main goal is to determine the lineage of a given data item all the way back through the graph of mappings. This eventually leads back to one or more golden sources. We therefore call this concatenation of mappings the lineage path in the mapping graph. Note that a source for one mapping can also be the target for another mapping.

### Mapping Graph

The **mapping graph** $MG(S, M)$ is composed of vertices $S := \{s_0, s_1, ..., s_i\}$ and directed edges $M := \{m_0, m_1, ..., m_j\}$, $i, j \in \mathbb{N}$. $S$ is the set of all sources and $M$ is the set of all mappings which are present in the system. where $|S| \geq 2$ and $|M| \geq 1$.
An edge $m_k = \langle s_m, s_n \rangle$ is considered to be directed from $s_m$ to $s_n$ for $s_m, s_n \in S$.
Using the previously explained mapping data model we can deduct the following property:
Each mapping always contains exactly one source and one target entity. Therefore we can define $s_m$ to be the **source** and $s_n$ to be the **target** of the mapping $m_k$.

### Lineage Path

The **lineage path** is a subset of mappings $L \subseteq M$ in the mapping graph. It describes the path that a data item took in the mapping graph from the golden source up to the user specified target, i.e. the set of mappings that were used during the computation of the result.

To find the lineage path, we need to navigate from the final result back in the mapping graph step by step and follow the edges in reverse order from source to source. We need to execute the same process recursively, until we reach the end of each path, i.e. the golden source. This algorithm can be roughly outlined as follows:

1. Find all mappings that use the current source as target
2. Extract the responsible sources from each of those mappings
3. Preserve the constraints for each source
4. Treat each result from step 2 as new target and continue with step 1 until no more mappings can be found

The first step can be expressed in one simple query over the set of all mappings and results in a set of zero or more mappings. Since we start with one single item, we select this item as starting point in the mapping graph and move on from that.
But it is not sufficient to follow the lineage paths on the entity level as one would assume intuitively. Each attribute can be populated from numerous sources. So we need to follow the lineage path explicitly for each attribute. Following that, the second step needs therefore to return a list of all source attributes that populate the corresponding target attributes. This can be achieved with a separate algorithm which is explained in the next chapter. After retrieving the source attributes, we then need to make sure that no false positives are included in the path. This is an important part of our approach and one of the key features. We therefore collect and process any constraints on the attribute level for every lineage path. Detailed explanation of this issue follows shortly.

## 6.2. Extracting Information From Mappings

As explained in 6.1 it is necessary to extract some information from each mapping according to the results from the last step. In this case we are generally interested in the same elements of a mapping in every iteration, but it is very likely that every mapping is structured differently than the one before. Therefore we cannot re-use the same static query over and over again, but have to make adaptations to the parameters and cater for any feasibly mapping structure. In order to formulate those queries, we have to follow several intermediate steps and consider several factors. In a simple implementation, the user might need to work some of this parts out on her own. However, in our scenario, the user cannot be expected to have extensive knowledge about both the mapping structure and database systems. Therefore we offer a high degree of abstraction. That leaves us in a position where we have to interpret the input and then generate the corresponding queries automatically from that. However, we do not want to constrain the user with limited power of the system. As a result, the now proposed algorithm is more than just a necessary tool to compute data lineage. It is a general solution to extract information from mappings while considering the semantic meaning of each element. This might prove very useful when addressing similar issues as later described in chapter 9, where we are interested in other components of the mapping.

The first challenge is to create some kind of processable mapping blueprint, that gives us an idea of the structure of a mapping. We can then find out which elements we are interested in and where to find them. This part of the process can be re-used in every iteration. The next step is the extraction of information from the specific mapping at hand where given parameters are used to find matching elements as output.
Before we can describe the algorithm in detail, it is necessary to explain some of the used constructs and definitions.

### 6.2.1. Meta Tree

As a basis for the process of data retrieval from mappings, we introduce the so-called meta tree. This is a simplified representation of the grammar rules in graph form, which describes what the instance graphs of mappings parsed with the grammar can possibly look like. To support the tree representation, the changes to the grammar that are mentioned in section 5.2 were made. We normalized the rules in such way that now every rule is either a composition of several other rules or a set of mutually exclusive alternatives. Therefore we can now distinguish between two cases: the compositions of rules and the list of alternative rules.

Rules in the form of

```
aggregation   ::=   'SUM('attribute')' | 'AVG('attribute')' | 'MIN('attribute')'
                    | 'MAX(' attribute')';
```

are called **alternative rules**. There can be only one alternative element selected at a time and each alternative element may consist of only one component. In contrast, rules in the form of

```
navigation   ::=   'NAVIGATE' fromEntity 'TO' toEntity 'USING' (usingClause)+;
```

are called **composition rules**. Every non-terminal symbol on the right hand side of a composition rule is element of the same alternative and the composition rule itself may contain only a single alternative.

We made two simplifications in the transformation of the grammar to the meta tree. This happens in order to construct a tree and not just a directed graph with possibly cycles. The first simplification therefore attacks recursive rules. Such a rule is for example *condition*:

| | | |
|---|---|---|
| condition | ::= | multiCondition (condComponent)* ; |
| condComponent | ::= | andOrOperator multiCondition; |

We reduce those rules by ignoring the recursive element. The *condition* rule is instead assumed to be equal to

condition ::= multiCondition;

The reason for this follows the purpose of the meta tree. We want to create a map of possible occurrences of certain elements. While a condition may be a sequence of other conditions that are linked with AND or OR statements, we are interested only in the fact that an attribute may be part of a condition, not where specifically in the recursive chain of condition it is located, i.e. on which level of the tree.
Second, compositions of equivalent non-terminal symbols such as

concatenation ::= attribute '||' attribute;

are considered to be equal to the reduced composition rule

concatenation ::= attribute;

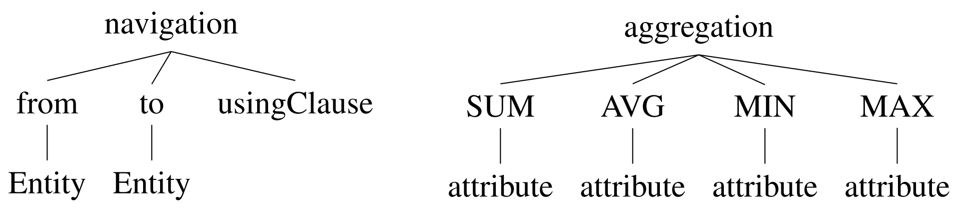
Again, we only need to know that an attribute may be part of a concatenation, not if it is the first or second element.

The tree representation according to this hierarchical structure of the grammar rules is the meta tree.

**Definition 3** *The* **meta tree** *is a graph MT(R,E) with a set of vertices* $R := \{r_0, r_1, ..., r_i\}$ *and directed edges* $E := \{e_0, e_1, \ldots, e_j\}$, $i, j \in \mathbb{N}$ *where* $|E| \geq 0$ *and* $|R| \geq 1$.

$R$ is considered to be the set of all rules in the grammar. Those can either be alternative rules or composition rules. $r_i$ denotes the i-th rule in the grammar and is therefore represented in the meta tree as node with Id i, i.e. $Id(r_i) = i$. $E$ is the set of all edges that represent the hierarchy of grammar rules. An edge $e_k = \langle r_m, r_n \rangle$ is considered to be directed from $r_m$ to $r_n$ for $r_m, r_n \in R$. Edges connect a grammar rule $r_m$ with each non-terminal symbol $r_n$ on the right hand side of the rule. The root of the tree is always $r_0$, i.e. the first grammar rule.

As an example let us have a look at the mentioned composition rule *navigation* and alternative rule *aggregation*. The tree representation of the *navigation* rule can be seen in figure 6.1a and similar figure 6.1b illustrates the *aggregation* rule. Figure 6.2 shows an larger excerpt of the full meta tree, where *aggregation* in incorporated.



(a) Composition rule (b) Alternative rule

Figure 6.1.: Example for meta tree construction

In a grammar, every production rule is eventually resolved to one or more terminal symbols. In the same way a parse tree contains only terminal symbols in leaf nodes. Since our meta tree does not visualize an actual input but merely the allowed structure, similar to the plain grammar, we include all non-terminal rules and not the terminals themselves. The nodes that form our leafs are therefore: entity, attribute, string, value and date. With this structure, we have now created a map to locate individual parameters for our blueprint query within their respective contexts.
If we want for example check every mapping that uses a certain value, we can immediately see in the meta tree where exactly a value may occur. In the example, this would be only one place, as part of a computation in the withClause. So without any further action, we immediately know exactly where we have to look for a value in mappings and its context.

The meta tree is not to be confused with the abstract syntax tree (AST) or the parse tree. While the AST and parse tree both display a concrete instance of the rules, we have created a generic presentation of the rules independent from any input. For example a computation in a parse tree can contain another computation and several values, while the meta tree only indicates the possible existence of a computation. The important information stored in the meta tree is the possibility of an occurrence of the computation at this position and therefore its context.

### 6.2.2. Affected Leafs

Once the meta tree is established, it allows us to select the parameters for the query. Possible examples for parameters are very specific requests, such as

"get all occurrences of '5' as value in a computation within the withClause" (I)
"get all occurrences of *AGREEMENT.ACCOUNT_CLOSE_DT* in a population" (II)

While (I) is a very specific parameter and can be directly resolved, the second request (II) is more general and needs some processing. By using the meta tree as it is displayed in 6.2, we can identify the relevant leaf nodes in both cases. Imagine the request is positioned at

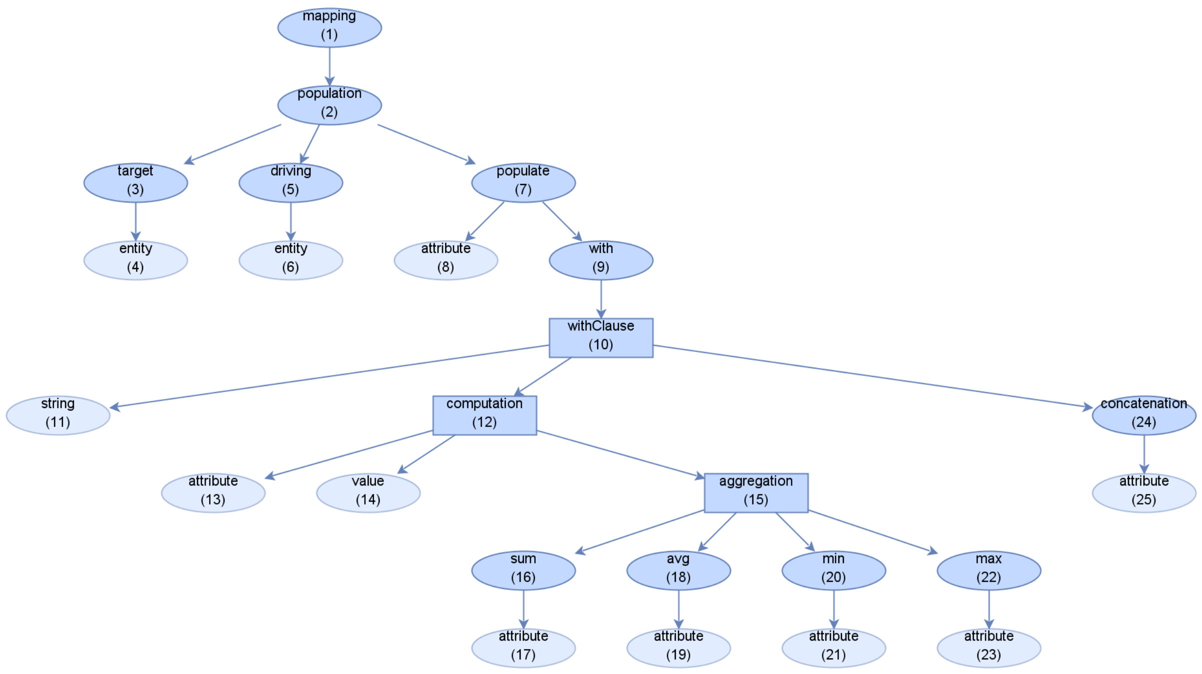


Figure 6.2.: Part of the meta tree built from the grammar rules. Oval nodes represent composition rules, square nodes alternative rules. Numbers in brackets denote the node id for more convenient references.

the given node in the tree, i.e. (I) at the corresponding 'value' node with Id 14, (II) at the 'population' node with Id 2. When traversing the subtree with the selected node as root, we visit all leaf nodes and add the appropriate leafs as well as their expected content to a list of matches. This has to be done for each parameter and results in one list of parameterized nodes. In the case of (I) this is easy, since we get a subtree with only a single node. Here 'value' (14) is already a leaf so we add it to the list and are done. For (II) this requires a more complex reasoning. Starting from the 'population' (2) node, we find 7 leafs of type 'attribute'. One directly as target of the populate (7) clause, 5 of them within a computation (12) and another one in concatenation (24). Thus we get a total of 9 parameterized nodes for the two example requests (I) and (II). An excerpt from the list of matching nodes after the process was executed is shown in Table 6.1. To simplify the description of requests, we propose the following short notation for the selection of parameters:

*context Id, node type, expected content*

For instance the two previous examples can be expressed as

*10, value, 5* (I)
*2, attribute, AGREEMENT.ACCOUNT_CLOSE_DT* (II)

| Node | Expected Content | Parent | Id |
|---|---|---|---|
| $value$ | 5 | $computation$ | 14 |
| $attribute$ | $AGREEMENT.ACCOUNT\_CLOSE\_DT$ | $population$ | 8 |
| $attribute$ | $AGREEMENT.ACCOUNT\_CLOSE\_DT$ | $computation$ | 13 |
| $attribute$ | $AGREEMENT.ACCOUNT\_CLOSE\_DT$ | $sum$ | 17 |
| ... | ... | ... | ... |

Table 6.1.: Parameterized Nodes

### 6.2.3. Computing The Context

To correctly identify all involved leaf nodes in the actual parse tree later on, we need to make sure that we can identify the semantic meaning of every node. For a given set of nodes in the parse tree $P = \{p_0, p_1, ..., p_i\}$, $i \in \mathbb{N}$ and nodes in the meta tree $R = \{r_0, r_1, ..., r_j\}$, $j \in \mathbb{N}$ we define

**Definition 4** *The* **semantic context** *($Context_s$) of a leaf node $p_n$, $n \in \mathbb{N}$ in the parse tree describes the exact position of its semantic representation $r_m$, $m \in \mathbb{N}$ in the meta tree. This may be expressed with the unique Id of the node $r_m$.*

This is equal to the n:1 mapping defined by $\forall p \in P\ \exists r \in R$: $Context_s(p) = Id(r)$ that maps each leaf in the parse tree to one leaf in the meta tree. In other words, the semantic context shows the semantic circumstances of the usage of a node $p_i$. For example the context of the target entity *AGREEMENT* in the parse tree in figure 5.5 is 4, as can be seen in the meta tree in figure 6.2. Therefore $Context_s(p_5) = 4$.
In order to compute the semantic context $Context_s(p_i)$, we need to utilize another tool, the set of identifying ancestors.

**Definition 5** *An* **identifying ancestor** *is a non-leaf node in the meta tree that necessarily has to exist exactly once in the set of ancestors for a given node in the meta tree.*

This includes every ancestor in the meta tree, where the corresponding grammar rule does not start and is not part of any recursion. Each identifying ancestor is used to unambiguously identify the semantic context of the succeeding nodes.
We consider *IA($r_n$) = {$r_i, ..., r_j$}*, $i, j \in \mathbb{N}$ to be the set of all identifying ancestors of $r_k$, $k \in \mathbb{N}$ in the meta tree. In contrast *allAncestors(x),* $x \in P \vee x \in R$ describes the set of all ancestors of node $x$, including those that are not identifying ancestors for $x \in R$. We write $isIdentifying(r_i)$,$i \in \mathbb{N}$ to express the fact that $r_i$ is an identifying ancestor.
To underline this definition, we look at a simple example in our meta tree in figure 6.2. The set of identifying ancestors IA($r_{14}$) = $\{r_{10}, r_9, r_7, r_2, r_1\}$. Node $r_{12}$ is not part of the set of identifying ancestors, since computations may occur recursively.

Furthermore we introduce the **computational context** ($Context_c$) of a leaf node $p$ as follows:
Let be $C = \{r_0, r_1, ...\}$, $C \subseteq R \wedge \forall r_i \in C : isIdentifying(r_i) = true$, further
let be $D = \{p_0, p_1, ...\} \wedge D \subseteq P$, $\forall p_j \in D\ : p_j \in allAncestors(p_n)$ and $r_j \in C$ for

$j = Context_s(p_j)$.
Then we can say $Context_c(p_n) = r_k \mid k = Context_s(p_l), p_l \in D, r_k \in R$.

From those definitions, we can make the following deductions:

- $Context_c(p_i)$ = IA($r_j$), $j$ = Id($Context_s(p_i)$)
- $Context_c(p_i)$ = $Context_c(p_j) \Leftrightarrow Context_s(p_i)$ = $Context_s(p_j)$

Using those, we can now compute the semantic context as

$$Context_s(p_x) = \text{Id}(r_y) \mid r_y \in R, Context_c(p_x) = IA(r_y), y = Context_s(p_x)$$

As an example, let us look at the parse tree and consider the node *Attribute* with Id 12 and content *CUSTOMER_MASTER_DATA.LAST_ACCOUNT_CLOSING_DATE*. According to the presented definitions, we can compute $Context_s(p_{12})$ = 13, since the corresponding node in the meta tree is $r_{13}$.

With those tools at hand, we can now generically compute and compare the context of each node in the parse tree. This is an advantage, since we do not need to take the actual structure of the current tree into consideration, but have defined a general solution that works for every possible parse tree and therefore every possible mapping.

### 6.2.4. Select Data

In this step we now load the actual data from a mapping according to the parameters selected in 6.2.2. To find matching data items in the parse tree, we need to consider and check for two properties of every data item:

1. Node type and content
2. Semantic context

This is reflected in our proposed notation from section 6.2.2. It is necessary to consider those properties, since not every occurrence of a given node content happens in the same type of node. For instance the content *AGREEMENT* may occur as an entity with this name or as a simple string without computational value. Depending on the situation we may be interested in only one of both manifestations, say the entity. Now an entity may be used in several different contexts, which we also have to distinguish. In the given sample from the meta tree in figure 6.2 entity may occur with the semantic context 4 and 6, i.e. as target or source. But if we are looking only for a target entity, the entity with semantic context 6 is not relevant.

We assume as a simple example that we are given the same mapping as in figure 5.4. Let us say we are now interested in the request

13, attribute, *CUSTOMER_MASTER_DATA.LAST_ACCOUNT_CLOSING_DATE*

That means, we want to see where attribute *LAST_ACCOUNT_CLOSING_DATE* of table *CUSTOMER_MASTER_DATA* is used as component of an population that in turn is used to populate another attribute. Figure 6.3 shows the matching attribute in bold.

WHEN POPULATING *Entity: AGREEMENT*
FROM *Entity: CUSTOMER_MASTER_DATA*
POPULATE *Attribute: AGREEMENT.ACCOUNT_CLOSE_DT*
WITH **CUSTOMER_MASTER_DATA.LAST_ACCOUNT_CLOSING_DATE** IF
        *CUSTOMER_MASTER_DATA.LAST_ACCOUNT_CLOSING_DATE != 01.01.0001*
WITH *"DEFAULT MAX"*

Figure 6.3.: Simple example mapping. The attribute of interest is written in bold.

However, as can be seen in both figure 6.3 and 5.5, this attribute occurs at two different locations, once in a with clause and once as part of a condition. Yet we are only interested in the first one.
As explained, we have to determine all relevant leaf-nodes in the parse tree first and then check the semantic context of those in order to eliminate the false positive. The first step is to query for any occurrence of *LAST_ACCOUNT_CLOSING_DATE*. The result is shown in table 6.2 and contains both of the mentioned nodes.

| Id | Content | Node Name | Parent | Semantic Context |
|---|---|---|---|---|
| 12 | CUSTOMER_MASTER_DATA. LAST_ACCOUNT_CLOSING_DATE | Attribute | 11 | 13 |
| 17 | CUSTOMER_MASTER_DATA. LAST_ACCOUNT_CLOSING_DATE | Attribute | 16 | 29 |

Table 6.2.: Result of the query and the respective semantic context of each node

The semantic context of each node was added manually for a better overview. The calculation of the semantic context for each element in the parse tree was outlined in 6.2.1 and won't be repeated here. It is now easy to see that both elements are used in a different semantic context. We can gather from the meta tree that the semantic context that we are interested in is 8, taken from the Id of the corresponding node. By comparing the semantic context of each node with the relevant semantic context, we can see that only one node remains. This is our final result in this step and shown in figure 6.3.

| Id | Content | Node Name | Parent |
|---|---|---|---|
| 12 | CUSTOMER_MASTER_DATA. LAST_ACCOUNT_CLOSING_DATE | Attribute | 11 |

Table 6.3.: Result from the data selection

### 6.2.5. Compose Data And Project Result

The previous section 6.2.4 showed how to extract elements from mappings. But since we are not only interested in the basic fact that a certain element exists in a mapping, but also how it relates to others, we offer a more complex query mechanism that can answer requests such as

"get all attributes that occur in the semantic context 10 and are in the same population as the attribute table0.attribute0 with semantic context 8 "

We translate this request into the query

*8, attribute, A : 10, attribute*

The first parts defines the look-up, as explained before: an attribute in the semantic context 8 or below in the meta tree, with the content A. The second part defines the projection from the mapping to all attributes with or below the semantic context 10, independent from their respective content. We assume implicitly that both should be *semantically related* in such way that they are both used in the same part of a mapping. In this case, we are looking for pairs of attributes that occur in the same population, not just in any population clause. One should have the given semantic context and content, the other is within a given semantic context and can have any content. To explain this relation formally, we need to introduce the following notations:

The **lowest common identifying ancestor** (LCIA) of two nodes $p, q \in R$, where R is the set of all nodes in the meta tree, is the lowest common ancestor of $p$ and $q$ that is identifying, i.e. $r = lca(p, q) \wedge isIdentifying(r) = true$, or short $r = lcia(p, q)$.

Two elements $p$, $q \in P$ where P is the set of all nodes in the parse tree are said to be **semantically related** in the lowest common identifying ancestor $r_{related}$, or shorter in the semantic context $Id(r_{related})$, if the following set of conditions hold:

1. $Context_c(p_i) \cap Context_c(p_j) \neq \emptyset$
2. $allAncestors(p_i) \cap allAncestors(p_j) \neq \emptyset$

Then we can say

1. $r_{lcia} = lcia(Context_s(p_i), Context_s(p_j))$
2. $A = allAncestors(p_i) \cap allAncestors(p_j)$

With that we can derive the relevant common ancestor to be

$$p_{common} \text{ for } p_{common} \in A \wedge Context_s(p_{common}) = Id(r_{lcia})$$

and from that define

$$r_{related} = r_n \mid n = Context_s(p_{common})$$

To apply this definition to an example, let us examine the mapping from figure 5.4, and pose the request

*8, attribute,* AGREEMENT.ACCOUNT_CLOSE_DT *: 10, attribute*

We can split this up into the two mentioned steps of executing the look-up and then the projection.

## Look-Up

By looking at the mapping tree in 5.5, we can see that *AGREEMENT.ACCOUNT_CLOSE_DT* occurs only once in the mapping as $p_8$. Its semantic context computes as $Context_S(p_8) = 8$, i.e. it is the attribute that gets populated by the population. We are looking exactly for attributes with context 8, therefore it is a match.

## Projection

Now we consider the projection of results for the items found during the look-up. This means we have to look up all remaining attributes in the table and then check the context for each. Querying the tree for this delivers a small result set:

| Id | Content | Node Name | Parent |
|---|---|---|---|
| 12 | CUSTOMER_MASTER_DATA. LAST_ACCOUNT_CLOSING_DATE | Attribute | 11 |
| 17 | CUSTOMER_MASTER_DATA. LAST_ACCOUNT_CLOSING_DATE | Attribute | 16 |

We are now able to compute both the computational and semantical context for all the attributes and find:

| $p_i$ | $Context_S(p_i)$ | $Context_C(p_i)$ |
|---|---|---|
| $p_{12}$ | 13 | 10, 9, 7, 2, 1 |
| $p_{17}$ | 29 | 15, 13, 9, 7, 2, 1 |
| $p_8$ | 8 | 7, 2, 1 |

Since we are interested only in attributes that are used in a sub-context of context 10, we can discard $p_{17}$ since context 10 is not part of its computational context. The lowest common identifying ancestor of the remaining attribute $p_{12}$ and the given $p_8$ can now be computed as

$$lcia(p_{12}, p_8) = 7$$

Expressed in natural language that means $p_{12}$ and $p_8$ are both part of the same populate clause, as stated in the initial request. This can be verified with a look at the mapping or parse tree.

In terms of lineage calculations, we use exactly this type of request recursively. More general

*8, attribute, X : 10, attribute*

where X is a placeholder for the currently investigated attribute. This request lists all attributes that were used to populate the given attribute X. If we were interested in further lineage calculations for our example, we would now proceed with $p_{12}$, find all mappings that populate CUSTOMER_MASTER_DATA and run the query with the new found attribute on those, i.e.

*8, attribute, CUSTOMER_MASTER_DATA.LAST_ACCOUNT_CLOSING_DATE : 10, attribute*

## 6.3. Condition Evaluation

What we offered so far was the passive lineage of attributes. But one of the advantages of our approach is the incorporation of the logic behind each mapping, that leads us to active lineage. We therefore examine the conditions under which different data items are used for the computations. This allows to reduce the number of lineage paths in some cases, as we indicated in section 3. While each mapping may allow for a broad band of data items to be processed, a sequence of mappings potentially removes some attributes from the result by applying excluding conditions on populations. This is not necessarily an error in the design of mappings. Intermediate sources may be starting point for several different computations with different purposes and therefore serve as sources for many mappings that require different data.
We therefore collect conditions along the lineage path and evaluate them after every step. However, the selection of paths may be further restricted if the user has additional knowledge about the composition of attributes or wants to constrain the lineage on given properties. In order to provide this functionality, we differentiate between preserved conditions, which are a collection of conditions along the lineage path and the initial conditions, which are user specified.

### 6.3.1. Preserved Conditions

While traversing the mapping graph, we dynamically record and update the conditional logic according to previous steps. The usefulness of maintaining and updating the combined conditions along the lineage can be shown in a quick example. Imagine a scenario with five intermediate results which are represented in the graph as five nodes $N_i$, $0 \leq i \leq 4$ where the data flow is directed from $N_{n+1}$ to $N_n$ for $0 \leq n \leq 2$. That means every attribute from $N_{n+1}$ has a corresponding source in $N_n$. Additionally we insert a mapping from $N_4$ to $N_3$ and one from $N_5$ to $N_3$, but with slightly different properties, which will be defined shortly. Figure 6.4 visualizes the corresponding mapping graph.

| Mapping | Target | Source | Source Condition |
|---|---|---|---|
| $M_0$ | $A_0$ | $A_2$ | $D_{Europe} \cup D_{Americas}$ |
| | $A_1$ | $A_3$ | $D_{Europe} \cup D_{Americas}$ |
| $M_1$ | $A_2$ | $A_4$ | $D_{Americas}$ |
| | $A_3$ | $A_5$ | $D_{Americas}$ |
| $M_2$ | $A_4$ | $A_6$ | $D_{Europe} \cup D_{Americas}$ |
| | $A_5$ | $A_7$ | $D_{Europe} \cup D_{Americas}$ |
| $M_3$ | $A_6$ | $A_8$ | $D_{Americas}$ |
| $M_4$ | $A_7$ | $A_9$ | $D_{Europe}$ |

Table 6.4.: Mapping components in a population

Every mapping $M_j$, $0 \leq j \leq 4$ contains a set of mapping rules, or more specifically rules about populations of certain attributes. Let us denote the attributes that are populated by those rules as $A_k$, $k = 0, 1, 2, \ldots$ which we use as an arbitrary numeration to distinguish individual attributes from several entities in this example. Table 6.5 shows a list of all exemplary nodes with their respective attributes.
In a mapping every $A_k$ is provided with a condition $C_k$ that adds a certain constraint which has to hold true in order to allow the population to be executed. Let those constraints for instance be set in a way that they select a subset $D_r$ of all data $D$ for which the population transformation is applied, with $r \in \{Europe, Americas\}$. This could be translated as a selection only consisting of data connected to a specific region. In Table 6.4 we can see the list of mapping elements to consider in this example. So in accordance with the grammar, the relevant part of the rules could be written as

$$\ldots\text{POPULATE } A_0 \text{ WITH } A_2 \text{ IF } A_2.REGION = Europe \text{ OR } A_2.REGION = Americas\ldots$$
$$\ldots\text{POPULATE } A_1 \text{ WITH } A_3 \text{ IF } A_3.REGION = Europe \text{ OR } A_3.REGION = Americas\ldots$$
$$\ldots$$

Now let us assume we are interested in the lineage of the two data items $A_0$ and $A_1$ in the last node, namely $N_0$. It is easy to see from the table that the source of $A_0$ is $A_2$ and the source of $A_1$ is $A_3$. On first glance it is sufficient to apply the initially given constraint about the region within every following iteration, which dictates that data from both Europe and Americas is considered in both attribute populations. But if we follow the next step back from $N_1$ to $N_2$ and then from $N_2$ to $N_3$ while keeping that initial constraint, we will now start to include false positives. $M_1$ only relays data from Americas. Yet when processing $M_2$, we again consider data from Europe in further lineage inquisitions, even though the data contained in $A_0$ and $A_1$ is solely associated with Americas. At the lineage path junction in $N_3$ we should therefore follow $M_3$ exclusively in further investigations and ignore $M_4$. The same issue arises if we instead apply only the current constraint of every population and not the initial one, as can be seen at the example of $M_2$ and $M_1$.
Two conclusions can be drawn from those considerations. Mainly we have shown that it is vital to dynamically update the constraints after querying each mapping, rather than using the

| Node | Attribute |
|---|---|
| $N_0$ | $A_0$ |
|  | $A_1$ |
| $N_1$ | $A_2$ |
|  | $A_3$ |
| $N_2$ | $A_4$ |
|  | $A_5$ |
| $N_3$ | $A_6$ |
|  | $A_7$ |
| $N_4$ | $A_8$ |
| $N_5$ | $A_9$ |

Table 6.5.: Node attributes

$$\begin{tikzcd}
 & & & N_4 \arrow[d, "M_3"] \\
N_0 & N_1 \arrow[l, "M_0"] & N_2 \arrow[l, "M_1"] & N_3 \arrow[l, "M_2"] \\
 & & & N_5 \arrow[u, "M_4"]
\end{tikzcd}$$

Figure 6.4.: Visualization of nodes and mappings

static conditions in every population. If the last constraint is stronger than the one before, we only apply the last one and vice versa. This can be achieved by detecting the strongest condition after every step and then evaluating the aggregated condition accordingly. Secondly it is obvious that we need to work on attribute level, since each attribute has to follow different constraints according to their individual population rules.
But how can we update the conditions properly? There are certain limitations, due to the semantic meaning of certain decisions. We include for example France as a region as part of Europe. A human user can intuitively understand that data from France might be included in a mapping for data from all of Europe, but not vice versa. To consider such a conclusion in the algorithm, we would need to formulate a potentially unnumbered amount of additional and very specific rules. What we can include however, is the purely logical aspect of conditions.

### 6.3.2. Basic Computation

For a given set of two population rules

$$\begin{gathered}\ldots\text{POPULATE } A_0 \text{ WITH } A_1 \text{ IF } A_1 = X \text{ OR } A_1 = Y\ldots \\ \ldots\text{POPULATE } A_1 \text{ WITH } A_2 \text{ IF } A_2 = X \text{ OR } A_3 = Z\ldots\end{gathered}$$

where we follow the active lineage of $A_0$, we now need to combine both conditions. Since the passive lineage leading to $A_0$ can be reduced to $A_2 \rightarrow A_1 \rightarrow A_0$, we substitute those three attributes with a general $A$:

$$\begin{gathered}A = X \text{ OR } A = Y \\ A = X \text{ OR } A_3 = Z\end{gathered}$$

Now we can deduce the strongest common condition along the transformation path as

$$A = X$$

meaning that only data is conveyed through the transformations for which the given attribute A equals X, where A is the original attribute in the golden source. Following the potential next step can be handled accordingly with the source of $A_2$ and its population rules. By applying this technique we can reduce the amount of followed lineage paths. If a combination of conditions results in a logical FALSE, for example $A > 0$ AND $A < 0$, we can stop the graph traversal at this point without further consideration, since no single data item can possibly fulfill this condition and therefore no data was relayed along this path. The same holds true if no common condition exists.
When we finally reach the golden source of the data transformations, we have accumulated a set of conditions that must hold for any data item within the source. It is now possible to run a query according to these conditions and select only the data items which are relevant for the final result that we examined in the beginning of the process.

### 6.3.3. Advanced Transformation

Since mapping transformations are not always formulated in such a simple form, we need to address the issue of combinations of several attributes, such as computations like

$$\begin{array}{ll}\ldots\text{POPULATE } A_0 & \text{WITH } A_1 + A_2 \text{ IF } A_1 < A_2 \\ & \text{WITH } A_1 - A_2 \text{ IF } A_1 > A_2\ldots \\ \ldots\text{POPULATE } A_1 & \text{WITH } A_3\ldots \\ \ldots\text{POPULATE } A_2 & \text{WITH } A_4\ldots\end{array}$$

To follow the active lineage of $A_0$ we have to examine two paths, one leads over $A_1$ to $A_3$, the other over $A_2$ to $A_4$. When investigating each path, the previously explained approach works for each path separately. We determine the first transformation to be $A_3 \rightarrow A_1 \rightarrow A_0$ and the second $A_4 \rightarrow A_2 \rightarrow A_0$ and substitute accordingly. The final result in terms of lineage is the union of data from each path.

### 6.3.4. Initial Conditions

In contrast to the preserved conditions, the initial conditions are given by the user and not part of a mapping. There are many reasons why the user might want to further constrain the lineage computation. For one, she might be interested only in a subset of data items from the result. More reasons to specify initial conditions are given in Chapter 7. It should be noted that in order to give meaningful constraints, the user has to be familiar with some the contents of tables and at least vaguely with the structure of the system.

It is obvious that tables often include data that is not relevant for every mapping that uses this table as a source. In the same way it may be a target for many different mappings. For example a table such as *PARTNER_CSID_MASTER* includes data about every partner of the company, may it be an individual or a business. A lineage computation for a given attribute such as *PARTNER_CSID_MASTER.BIRTH_OR_FOUNDATION_DATE* must therefore yield the lineage of both individual and business data. Now the user might be interested in only individuals. Therefore she has to specify that she wants to compute the lineage for only those mappings that process data from individuals. If the only indicator for those mappings is the partner type attribute, she has to add a constraint on the content of this attribute, e.g. "PARTNER_CSID_MASTER.PARTNER_TYPE = 2", where 2 is the index for individuals. It is then necessary to evaluate those constraints accordingly. We therefore consider them to be the first condition that is then updated in the first step according to the process in 6.3.1.

## 6.4. Complete Algorithm

Now that we have defined each step that is necessary in order to move along the path and extract the information that are needed, let us have a look at the complete algorithm to set the single steps into context. As explained, the approach can be divided into two parts. At first we prepare the static elements of the system, such as the meta tree. Then we run the algorithm recursively until all golden sources are determined. We are generally interested in the lineage of one attribute $A$, which we use as starting point.

1. Preparations:
   - Build the meta tree
   - User defines the starting point $A^*$ as $A$
   - User may define initial condition $C_0$
2. Find all relevant mappings $M$ which populate the selected attribute $A^*$:
   - Find all mappings that use the entity of attribute $A^*$ as target
   - If no mappings exist, then $A^*$ is stored in a golden source for $A$
3. Extract all used attributes $U$ from $M$ for given $A^*$:
   - Find occurrences of $A^*$ in $M$

- Compute the context of each $A^*$ in $M$
- Project the result $U$ accordingly from $M$

4. For each attribute $A'$ in $U$ do the following:
    - Record all conditions $C'$ for $A'$
    - Evaluate the new set of conditions together with those that were recorded for $A^*$
    - If they result in a definite *false* then stop here
    - Else start again at 2. with $A'$ as new $A^*$

# 7. Graph Traversal Patterns

In this section we are going to explain some of the most likely reoccurring patterns that emerge from the traversal of the lineage path and the conclusions we draw from them. In a complex real world application, like the metadata warehouse of Credit Suisse, there exist numerous layers and stages of mappings [JBM$^+$12]. This system will grow dynamically in the future and therefore will most likely get new inputs and outputs attached in various ways.
One very real issue in that context is that our system assumes a limited amount of resulting lineage paths. Although our main goal is to reduce the number of followed paths and we have shown an approach that tries to achieve this, in reality there may occur situations that our algorithm cannot cover. Most likely there will at some point emerge a chain of mappings where repeatedly only very weak conditions are specified and numerous inputs are used so that they produce a very large amount of paths to follow. Even with active lineage the complexity of the provenance may then exceed the feasible computational effort. There is also no gain in following dozens or even hundreds of highly branched paths, only to present the user with an overwhelming result that includes most likely many sources she is not interested in.
The reason for such an exploding number of results are often weak filter criteria. The whole concept of active lineage is based on the idea that paths can be excluded early in the traversal based on given filters, as we have shown in section 6.3. To apply this concept, we need to assure that those filter criteria are strong enough to adequately reduce the amount of paths. However, we have no influence on the quality and strength of the criteria in mappings, therefore we need additional information from the user in that situation. A possible indication of weak filters could be a crossed threshold on the number of followed paths. This threshold should be user specified and depends on the underlying system of mappings. Once this threshold is reached, the user then has to take responsibility and add stronger conditions, based on the expected result. She might for example specify a certain country code for relevant data items if mappings are organized by countries and she is interested only in data about certain areas or agree to continue the computation with the knowledge that it could yield a unnecessarily large result set.
Yet even with user specified initial conditions, the pure size and complexity of the system leads to other problems. Mapping paths stretch over several layers of the system that are structured differently. For example data that is organized according to country codes could very well originate from tables that are sorted by market segments and only then further diversified into countries. Filters that are accumulated over mappings that differentiate populations based on country codes naturally have no relevance for conditions based on market segments. Even with the active lineage algorithm and accumulated conditions we would then need to follow paths from every market segment. This could increase the number of results exponentially, even if we reduced the paths significantly in the steps before. Figure 7.1 illustrates this. Mappings that populate the two market segments Investment Banking and Private Banking filter

accordingly. But data that is structured in country tables is filtered differently. Therefore even after reducing the lineage path to *US* data only, we are then forced to follow every data flow out of both market segments.
Here again user interaction is needed. In the case that filters are not compatible anymore with the processed mappings, the user could make a manual decision which paths should be followed. Imagine for instance a multiple choice selection for the user, where she can select different conditions that are read from the mapping. The chosen conditions would then serve as new initial conditions for further lineage computation. In our example she might consider only Private Banking data as relevant, but did not know that the differentiation was relevant when starting the process based on country codes.

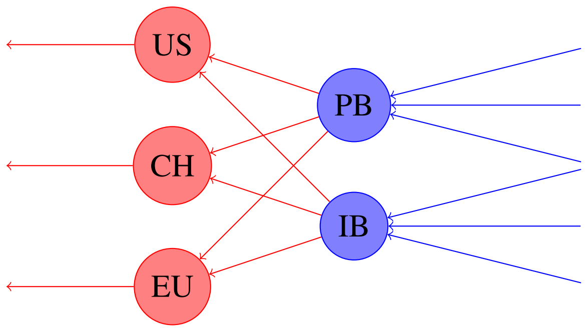


Figure 7.1.: Mappings based on locations (red) and market segments (blue)

A second issue that arises from the real world application concerns the selection of attributes. We follow lineage paths on an attribute level and assume a single attribute as a starting point. In reality a user is more likely interested in the lineage of a table record which includes multiple attributes. A record is often a result of aggregated data from different sources, where the structure of single rows is preserved or extended, but not recombined. For example a table like *PARTY* contains information about every party the company does business with. This includes employees, business partners and customers. The table has a column called *PARTY_END_DT* that specifies the date this party ceased being of interest to the financial institution, may it be an employee that leaves the company or a business that cancels the contract. *PARTY* also own an attribute *PARTY_ID* which is the unique identifier for an individual, organization or household that is of interest to the enterprise. As a result the mapping with target *PARTY* aggregates records from three tables, one for each type of party, but does not modify the respective rows. A record in $PARTY$ with the identifier of an employee can only include the end date of the corresponding employee contract and may never be paired with that of a business agreement.
Now obviously lineage investigations for *PARTY_END_DT* yield sources for all of employees, customers and partners, including every sources that may be relevant in different date computations during the sequence of mappings. In the same way another lineage investigation for *PARTY_ID* results in sources for all three types of party. But in reality, the record can contain only pairs of data grouped by the respective source. Figure 7.2 illustrates this problem by magnifying the attribute lineage for customer data in *PARTY*. While the resulting record in

*PARTY* is always created from data in $C$, mappings to an intermediate result may mistakenly introduce alternative lineage paths for *PARTY_END_DT*. A possible countermeasure for this problem is to compose the intersection of both lineage paths in order to find only common sources. In this scenario, the intersected lineage of the record would point only to $C$.

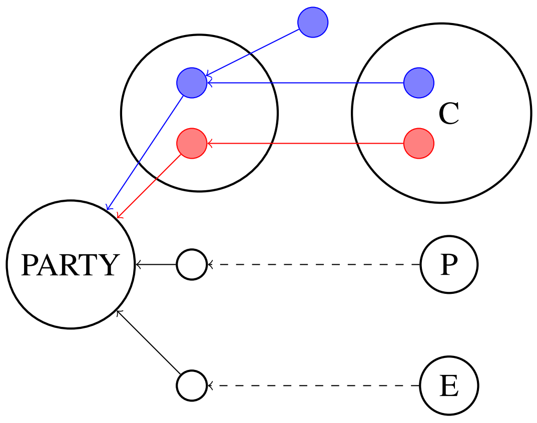


Figure 7.2.: Linage paths for record in *PARTY*, to customer, partner and employee data. Magnified on attributes in customer data: *PARTY_ID* (red) and *PARTY_END_DT* (blue).

The last aspect that we have to consider is our strong dependence on the quality and coverage of mappings that is needed to supply the desired precision of lineage. If the documented mapping path is disconnected by faulty mapping specifications or sources outside the available system, the lineage computation is interrupted. This is one of the issues that deters us from deploying recording or tracing methods, as they are used in other systems (cf. chapter 4). While there is no technical solution to this problem, it may ,however, help to detect errors in the mapping specification an could be used as indicator for inadequate data quality.

# 8. Prototype

Having explained the algorithm and its application in detail, we will now present a running prototype that was developed based on the theoretical work. It serves as a proof of concept and implements the algorithm that has been developed in this thesis. As we will explain further in section 8.2, we focused on the extraction algorithm and restricted the implementation on passive lineage. As programming language we chose Java.

## 8.1. Architecture

First we will outline the overall architecture and individual elements of our prototype. In order to keep the approach modular, we decided to split the implementation into five parts. This keeps the concrete implementation of different subtasks interchangeable. Each of those modules can be run individually and provides interfaces for structured input and output. Figure 8.1 shows the rough outline of the structure as well as the interaction between modules. In the

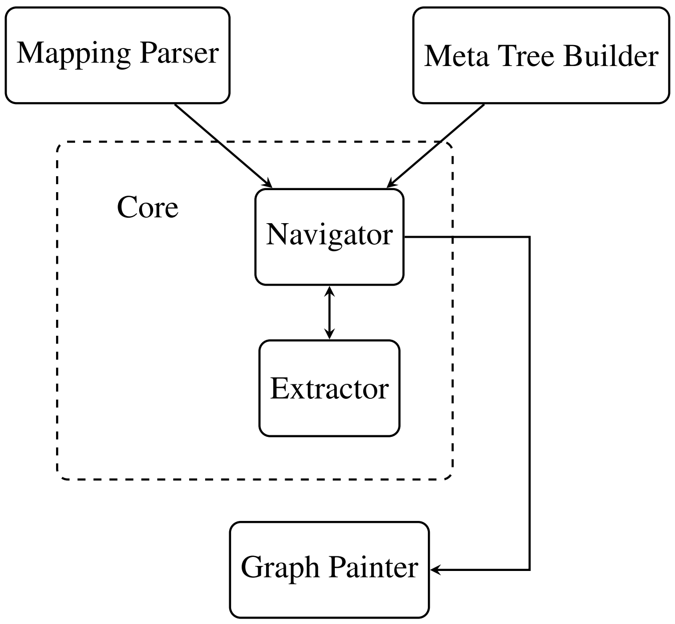


Figure 8.1.: Architecture of the Prototype

following sections we will explain the five individual modules, their respective task and how it is solved.

### Mapping Parser

The mapping parser serves as an interface to load mappings into our system. This module is necessary since the structure of currently available mappings is pure text only. Once data is available in RDF form, the parser can be interchanged with a module that offers database access and processes RDF data directly.
As a parser generator we used ANTLR [1] and created a suitable tree structure using the ANTLR specific rewrite rules to build a parse tree over the given grammar. This step allows for some optimizations, such as reducing the necessarily verbose input to only relevant tokens for the tree construction. For example we do not have to include the three tokens "WHEN", "POPULATING" and "FROM" into a parse tree, if we define that the following entity is the drivingEntity and insert a single node "driving" instead. Appendix B shows the grammar rules including the rewrite rules. The resulting parse tree is then utilized as outlined in as outlined in chapter 6.

### Meta Tree Builder

An important element in our approach is the meta tree. We describe its application in section 6.2.1. In the current implementation, the meta tree creation is not automated and does not use the grammar as direct input. The user is responsible to define nodes and their role in the tree manually. Those roles include *Alternative* (A), *Composition* (C), *Terminal* (T) and a list of nodes that are identifying ancestors in the meaning of the definition from section 6.2.3. The structure and roles have to be updated according to changes in the grammar by the user. We offer a text file to store those definitions and load the file at system start.
Once the tree is built, we index individual nodes in a depth first traversal with pre-order. Those indices are used for the context computations, as explained in section 6.2.3.

### Navigator

The navigator is part of the core implementation. It defines all steps that are necessary to compute data lineage along the mapping graph and therefore represents the implementation of section 6.1. It reads incoming mappings and poses corresponding requests to the extractor in order to compute the next steps in the mapping graph. This module can be interchanged in order to pursue different tasks, like impact analysis, instead of lineage investigations. Only alterations to the requests and conclusions that are drawn from the results have to be made. To visualize the results, it invokes the graph painter and renders the lineage path as a picture.

### Extractor

As can be taken from of chapter 6.2, the main challenge in this task was to provide a generic algorithm to extract information from mappings. The extractor implements this algorithm and can be run on any mapping that is structured according to the data model. It utilizes the meta tree that was created by the meta tree builder and processes mapping information that is

[1]http://www.antlr.org

provided by the mapping parser. Request to the extractor are posed by the navigator according to the current task and results are sent back accordingly.

### Graph Painter

As we need to represent most components in the form of a tree, the graph painter can be used to draw a graphic representation of any graph structure that is built in our system. This helps the user to visualize results or intermediate steps. Supported structures include parse tree, meta tree and accumulated lineage paths but can easily be extended. The painter is based on the open source framework JGraphX [2] and provides output as png or svg (Scalable Vector Graph) file.

## 8.2. Limitations Of The Prototype

The biggest limitation of the prototype is currently the computation of passive lineage only. It is possible to record all conditions along the lineage path, however, evaluating them requires a complex system to solve advanced logical expressions. We work with conditions such as comparisons between attribute and attribute, string and attribute, value and attribute, aggregations and combinations of all of those, all together linked with AND or OR statements as well as grouped in parenthesizes. A module that can minimize and check those expressions has to be very powerful in order to meet our requirements and would surpass the scope of this thesis. However, we have already introduced the principles of those mechanics in section 6.3.1.

Secondly, we read data locally from a a text file only and not from a database. While this has no impact on the functionality of the prototype, it limits the size of the evaluated set of mappings, since every mapping has to be included manually. This is mainly due to the limited availability of mappings in compatible format. However, an extension to connect to a database can be introduced in the future, as hinted in section 8.1.

[2] http://www.jgraph.com/

# 9. Future Extensions

After presenting both the theoretical approach and the implementations that serves as a proof of concept, we will now outline the conclusions that we can draw based on those results. We will address the possible extensions that might be addressed in the future, and give an idea on how to approach this.

## Checking Data Quality

We noted in section 7 that the quality of mappings in some areas is possibly below the needed level in order to provide accurate lineage information. This may be induced by a lack of documentation, a disruption of the data flow caused by the switch from one medium to another or too weak filter criteria on mappings. While this is troublesome in the aspect of lineage computations, it helps to find weaknesses in the existing data. Based on those finds, the responsible data owners can improve the quality of data accordingly. However, from this point of view, it is highly inefficient to rely on serendipity to spot affected mappings. Consequently, a possible extension would be a system that periodically checks on data quality. In other words, it could run automated lineage investigations on the whole system and report interrupted lineage paths that point to inadequate mapping specifications.

## Different Use Cases

A second feasible enhancement of the presented algorithm would be a modification that allows to investigate for other use cases that were mentioned in chapter 2:

- Impact Analysis
  This modification is fairly straight forward. The use case in chapter 2 already showed the possible benefit of this functionality. So instead of following the lineage path from a target back to all its sources, we could invert the process and list all the targets that are affected from a given source. Using the generic query expressions that are used to pose request to the system, as outlined in section6.2.5, it is also possible to search for attributes that are populated by given populations instead of vice versa.

- Optimization
  The proposed system to check for data quality is one facet of the general optimization of mappings. It is possible that several mappings are redundant in such a way that the are part of the mapping graph and passive lineage path, but not of any active lineage path. While this does not mean that there could be no future inputs or outputs that access those mappings, it might point to errors in mapping specification from a design perspective. This stands in contrast to the technical aspect like in the data quality considerations.

## Extending Grammar Rules

However, extending the algorithm to serve different use cases may lead to problems. It it possible that the simplifications we made when creating the meta tree in chapter 6.2.1 become relevant in another use case. However, this could be solved by increasing the granularity of the mapping rules. If we assume for instance that the order of concatenations of attributes is of importance for a given problem, we can adapt the grammar and therefore the meta tree accordingly. In this particular case, we would introduce two more rules, that represent the left hand side (lhs) and right hand side (rhs) of *concatenation*:

```
concatenation  ::=  lhs ’||’ rhs;
          lhs  ::=  attribute;
          rhs  ::=  attribute;
```

Now it is possible to specify parameters accordingly, whether an attribute is part of the *rhs*, or *lhs* respectively. The growth of the meta tree does not impact the algorithm.
Similarly more terminal symbols could be introduced, such as year, month and day as part of a date or for attributes the attribute name and the entity name. In general, it could be beneficial at some point to extend the grammar and therefore the meta tree in such way that every single element of a mapping can be queried.

## Assert Filter Relevance

Following the explanations in section 6.3.1 and the resulting patterns as shown in section 7, it becomes obvious that we need to evaluate the conditions of every mapping and assure that they are effective. Once the aggregated filter does not apply anymore, we basically loose the advantage of our system over a brute-force approach. Therefore a possible extension to the algorithm would be a process to check whether a filter applies to the next step or not. This could be started with user input, where it is currently only in the users responsibility to supply relevant filter conditions. In the case that a filter is too weak to filter and therefore reduce the next fork of lineage paths, the user should be informed. She then might want to apply stronger filter criteria, as suggested in chapter 7.

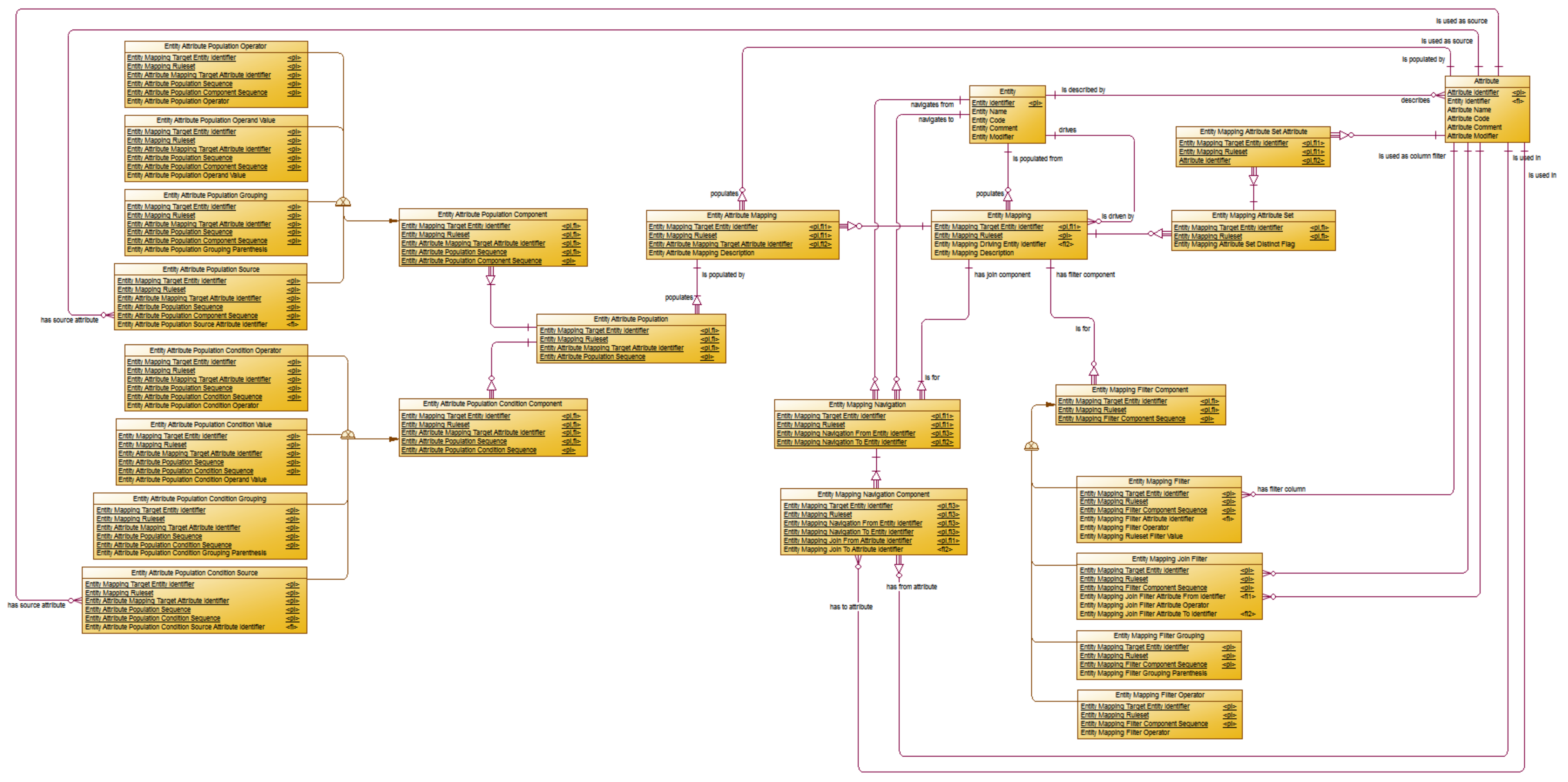
# 10. Conclusion

In this thesis we have presented a new concept to compute data lineage within the Credit Suisse warehouse environment. In contrast to other approaches, we do not access the concrete data in order to record executed transformations, instead we evaluate the conceptional mapping rules. In the financial industry, where data access is often restricted to users, this gives us a significant advantage. We are able to provide complete abstraction from sensitive data while still achieving full coverage on data lineage. To access the needed metadata, we built our system on existing infrastructure, namely the Credit Suisse metadata warehouse.
An essential component of our system is the algorithm that is used to gather information from mappings. This algorithm can extract mapping components based on their semantic meaning and recognizes the context in which components are used. By providing a generic interface for this algorithm, we built a basis that can be utilized to query for any element in the mapping structure. Consequently, this mechanism can be applied to answer many further inquiries besides data lineage.
The interface is completely abstracted from the implementation and uses the grammar syntax to offer intuitive interaction to users. Consequently, it can be used by any business user who has only minimal knowledge of mapping structures. Following that, our system fulfills the goal that no user interaction on a technical level should be necessary for the end user. Yet it provides precise and intuitive lineage information. In combination with the complete abstraction from actual data, this is a big step forward towards bridging the gap between business users and IT systems.
In conclusion, we believe that this thesis will provide valuable insight on data lineage, as well as general processing of mappings in the Credit Suisse metadata warehouse. While we focus on data lineage computations in this thesis, the developed algorithm might serve as starting point for many different applications in the future.

# Appendices

# A. Conceptional Model For Mappings

*see next page*

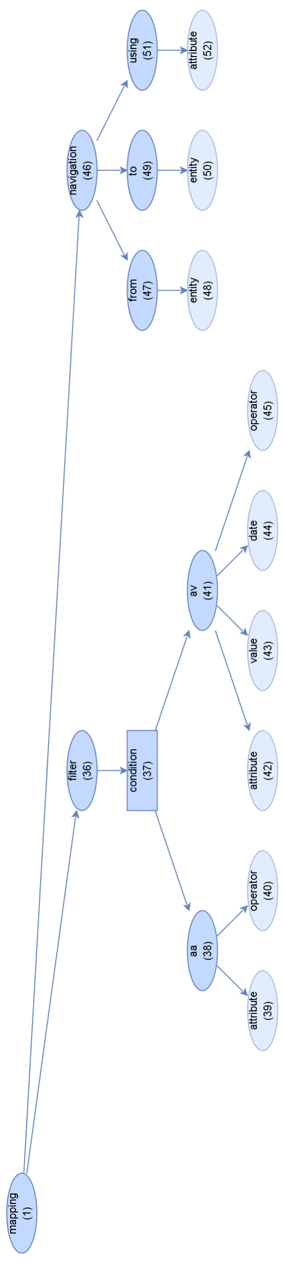

Attribute
Attribute Identifier <pi>
Entity Identifier <fi>
Attribute Name
Attribute Code
Attribute Comment
Attribute Modifier
Entity
Entity Identifier <pi>
Entity Name
Entity Code
Entity Comment
Entity Modifier
Entity Mapping Attribute Set Attribute
Entity Mapping Target Entity Identifier <pi,fi1>
Entity Mapping Ruleset <pi,fi1>
Attribute Identifier <pi,fi2>
Entity Mapping Attribute Set
Entity Mapping Target Entity Identifier <pi,fi>
Entity Mapping Ruleset <pi,fi>
Entity Mapping Attribute Set Distinct Flag
Entity Mapping
Entity Mapping Target Entity Identifier <pi,fi1>
Entity Mapping Ruleset <pi>
Entity Mapping Driving Entity Identifier <fi2>
Entity Mapping Description
Entity Attribute Mapping
Entity Mapping Target Entity Identifier <pi,fi1>
Entity Mapping Ruleset <pi,fi1>
Entity Attribute Mapping Target Attribute Identifier <pi,fi2>
Entity Attribute Mapping Description
Entity Attribute Population
Entity Mapping Target Entity Identifier <pi,fi>
Entity Mapping Ruleset <pi,fi>
Entity Attribute Mapping Target Attribute Identifier <pi,fi>
Entity Attribute Population Sequence <pi>
Entity Attribute Population Component
Entity Mapping Target Entity Identifier <pi,fi>
Entity Mapping Ruleset <pi,fi>
Entity Attribute Mapping Target Attribute Identifier <pi,fi>
Entity Attribute Population Sequence <pi,fi>
Entity Attribute Population Component Sequence <pi>
Entity Attribute Population Condition Component
Entity Mapping Target Entity Identifier <pi,fi>
Entity Mapping Ruleset <pi,fi>
Entity Attribute Mapping Target Attribute Identifier <pi,fi>
Entity Attribute Population Sequence <pi,fi>
Entity Attribute Population Condition Sequence <pi>
Entity Attribute Population Operator
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Attribute Mapping Target Attribute Identifier <pi>
Entity Attribute Population Sequence <pi>
Entity Attribute Population Component Sequence <pi>
Entity Attribute Population Operator
Entity Attribute Population Operand Value
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Attribute Mapping Target Attribute Identifier <pi>
Entity Attribute Population Sequence <pi>
Entity Attribute Population Component Sequence <pi>
Entity Attribute Population Operand Value
Entity Attribute Population Grouping
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Attribute Mapping Target Attribute Identifier <pi>
Entity Attribute Population Sequence <pi>
Entity Attribute Population Component Sequence <pi>
Entity Attribute Population Grouping Parenthesis
Entity Attribute Population Source
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Attribute Mapping Target Attribute Identifier <pi>
Entity Attribute Population Sequence <pi>
Entity Attribute Population Component Sequence <pi>
Entity Attribute Population Source Attribute Identifier <fi>
Entity Attribute Population Condition Operator
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Attribute Mapping Target Attribute Identifier <pi>
Entity Attribute Population Sequence <pi>
Entity Attribute Population Condition Sequence <pi>
Entity Attribute Population Condition Operator
Entity Attribute Population Condition Value
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Attribute Mapping Target Attribute Identifier <pi>
Entity Attribute Population Sequence <pi>
Entity Attribute Population Condition Sequence <pi>
Entity Attribute Population Condition Operand Value
Entity Attribute Population Condition Grouping
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Attribute Mapping Target Attribute Identifier <pi>
Entity Attribute Population Sequence <pi>
Entity Attribute Population Condition Sequence <pi>
Entity Attribute Population Condition Grouping Parenthesis
Entity Attribute Population Condition Source
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Attribute Mapping Target Attribute Identifier <pi>
Entity Attribute Population Sequence <pi>
Entity Attribute Population Condition Sequence <pi>
Entity Attribute Population Condition Source Attribute Identifier <fi>
Entity Mapping Navigation
Entity Mapping Target Entity Identifier <pi,fi1>
Entity Mapping Ruleset <pi,fi1>
Entity Mapping Navigation From Entity Identifier <pi,fi3>
Entity Mapping Navigation To Entity Identifier <pi,fi2>
Entity Mapping Navigation Component
Entity Mapping Target Entity Identifier <pi,fi3>
Entity Mapping Ruleset <pi,fi3>
Entity Mapping Navigation From Entity Identifier <pi,fi3>
Entity Mapping Navigation To Entity Identifier <pi,fi3>
Entity Mapping Join From Attribute Identifier <pi,fi1>
Entity Mapping Join To Attribute Identifier <fi2>
Entity Mapping Filter Component
Entity Mapping Target Entity Identifier <pi,fi>
Entity Mapping Ruleset <pi,fi>
Entity Mapping Filter Component Sequence <pi>
Entity Mapping Filter
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Mapping Filter Component Sequence <pi>
Entity Mapping Filter Attribute Identifier <fi>
Entity Mapping Filter Operator
Entity Mapping Ruleset Filter Value
Entity Mapping Join Filter
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Mapping Filter Component Sequence <pi>
Entity Mapping Join Filter Attribute From Identifier <fi1>
Entity Mapping Join Filter Attribute Operator
Entity Mapping Join Filter Attribute To Identifier <fi2>
Entity Mapping Filter Grouping
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Mapping Filter Component Sequence <pi>
Entity Mapping Filter Grouping Parenthesis
Entity Mapping Filter Operator
Entity Mapping Target Entity Identifier <pi>
Entity Mapping Ruleset <pi>
Entity Mapping Filter Component Sequence <pi>
Entity Mapping Filter Operator
is used as source
is used as source
is populated by
is described by
describes
navigates from
navigates to
drives
is populated from
is used as column filter
is used in
is used in
populates
populates
is driven by
is populated by
populates
has join component
has filter component
is for
is for
has filter column
has from attribute
has to attribute
has source attribute
has source attribute

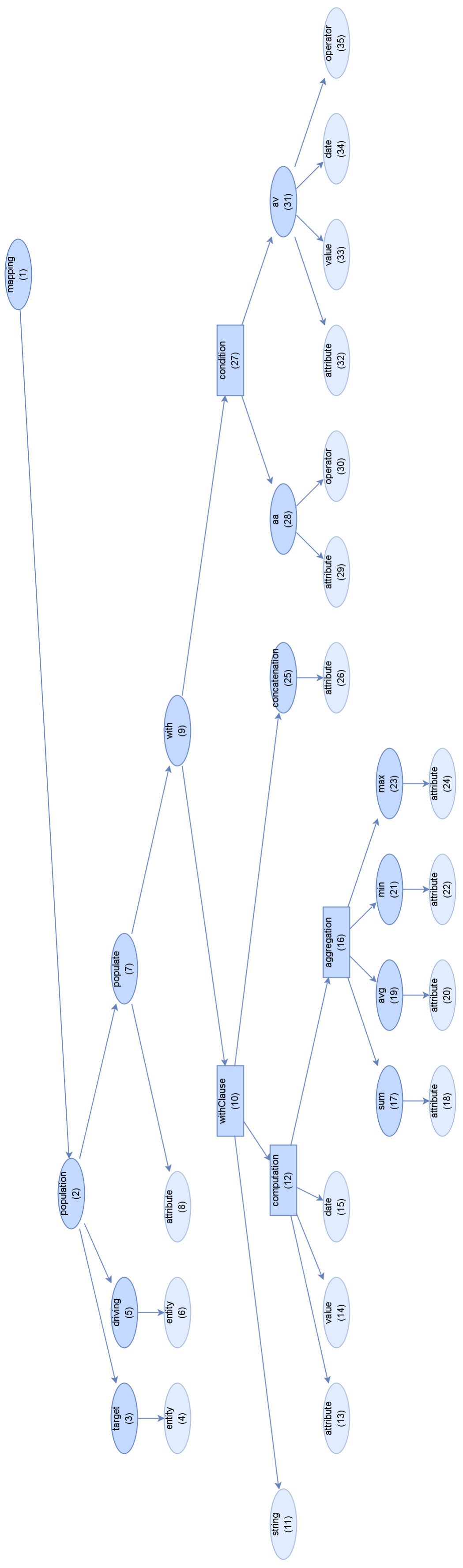
# B. Grammar Production Rules Including Rewrite

| | | |
|---|---|---|
| mapping | : | p=population (f=filter)? (n=navigation)? -> ˆ(Mapping $p $f? $n?); |
| population | : | ’WHEN POPULATING’ t=targetEntity FROM d=drivingEntity p=populationClause -> ˆ(Population $t $d $p); |
| populationClause | : | (’POPULATE’ populationComp)* -> ˆ(Populate populationComp)*; |
| populationComp | : | attribute withClause+; |
| withClause | : | (’WITH’ exp=expression (’IF’ cond=condition)?) -> ˆ(With ˆ(WithClause $exp) (ˆ(Condition $cond))?); |
| expression | : | stringExpression \| computation -> ˆ(Computation computation) \| concatenation -> ˆ(Concatenation concatenation); |
| concatenation | : | attribute ’\|\|’ attribute -> attribute attribute; |
| computation | : | (comp1 -> comp1) (’+’ c1=comp1 -> ˆ(Addition $computation $c1))*; |
| comp1 | : | (comp2 -> comp2) (’-’ c2=comp2 -> ˆ(Subtraction $comp1 $c2))*; |
| comp2 | : | (comp3 -> comp3) (’x’ c3=comp3 -> ˆ(Multiplication $comp2 $c3))*; |
| comp3 | : | (compElement -> compElement) (’/’ c4=compElement -> ˆ(Division $comp3 $c4))*; |
| compElement | : | attribute \| value \| aggregation \| LBRACK computation RBRACK -> computation; |
| filter | : | ’SELECT ROWS WHERE’ condition -> ˆ(Filter ˆ(Condition condition)); |
| navigation | : | ’NAVIGATE FROM’ e1=fromEntity ’TO’ e2=toEntity ’USING’ (u=usingClause)+ -> ˆ(Navigation $e1 $e2 ($u)+); |
| fromEntity | : | entity -> ˆ(From entity); |
| toEntity | : | entity -> ˆ(To entity); |
| usingClause | : | ((’,’? a3=attribute ’=’ a4=attribute) -> ˆ(Using $a3 $a4)); |
| condition | : | m=multiCond (c=condComp)* -> ˆ(ConditionComp $m $c*); |
| condComp | : | a=aoOperator m=multiCond -> ˆ($a $m); |
| multiCond | : | a1=attribute c1=condOperator a2=attribute -> ˆ(Aa ˆ($c1 $a1 $a2))\| (a1=attribute c1=condOperator v1=value) -> ˆ(Av ˆ($c1 $a1 $v1)) \| LBRACK condition RBRACK -> condition; |

| | | |
|---|---|---|
| targetEntity | : | entity -> ˆ(Target entity); |
| drivingEntity | : | entity -> ˆ(Driving entity); |
| entity | : | (’Entity:’? STRING) -> ˆ(Entity STRING); |
| attribute | : | (’Attribute:’? STRING) -> ˆ(Attribute STRING); |
| aggregation | : | ’SUM(’ attribute ’)’ -> ˆ(Sum attribute) \| ’AVG(’ attribute ’)’ -> ˆ(Avg attribute) \| ’MIN(’ attribute ’)’ -> ˆ(Min attribute) \| ’MAX(’ attribute ’)’-> ˆ(Max attribute); |
| compOperator | : | ’>’ -> ˆ(Gt) \| ’>=’ -> ˆ(Ge) \| ’<’ -> ˆ(Lt) \| ’<=’ -> ˆ(Le) \| ’=’ -> ˆ(Eq) \| ’!=’-> ˆ(Ne); |
| condOperator | : | compOperator \| string; |
| value | : | DATE -> ˆ(Date DATE)\| VALUE -> ˆ(Value VALUE); |
| aoOperator | : | ’AND’ -> And \| ’OR’ -> Or; |
| string | : | substring \| STRING; |
| stringExpression | : | ’"’ STRING* ’"’ -> ˆ(String STRING*) \| substring -> ˆ(String substring); |
| substring | : | ’substr’ LBRACK attribute position RBRACK -> ˆ(Substring position attribute); |
| position | : | ’,’ v1=VALUE ’,’ v2=VALUE -> ˆ(From $v1) ˆ(To $v2); |
| STRING | : | (’a’..’z’ \| ’A’..’Z’) (DIGIT \|’a’..’z’ \| ’A’..’Z’ \| ’_’)*; |
| ATTRIBUTE | : | STRING ’.’ STRING; |
| VALUE | : | DIGIT+; |
| DATE | : | DIGIT DIGIT ’.’ DIGIT DIGIT ’.’ DIGIT DIGIT DIGIT DIGIT; |
| fragment DIGIT | : | (’0’..’9’); |

# C. Full Meta Tree (split)




mapping (1)
population (2)
target (3)
entity (4)
driving (5)
entity (6)
populate (7)
attribute (8)
with (9)
withClause (10)
string (11)
computation (12)
attribute (13)
value (14)
date (15)
aggregation (16)
sum (17)
attribute (18)
avg (19)
attribute (20)
min (21)
attribute (22)
max (23)
attribute (24)
concatenation (25)
attribute (26)
condition (27)
aa (28)
attribute (29)
operator (30)
av (31)
attribute (32)
value (33)
date (34)
operator (35)

# Bibliography

[BCTV08] Peter Buneman, James Cheney, Wang-Chiew Tan, and Stijn Vansummeren. Curated databases. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '08, pages 1–12, New York, USA, 2008.

[BKWC01] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory - ICDT 2001*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, Berlin / Heidelberg, 2001.

[BT07] Peter Buneman and Wang-Chiew Tan. Provenance in databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1171–1173, New York, USA, 2007.

[Bud12] John Budd. Rene mapping. CS Internal, 2012.

[CCT09] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Found. Trends databases*, 1(4):379–474, April 2009.

[CW03] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1):41–58, May 2003.

[EPV10] Kai Eckert, Magnus Pfeffer, and Johanna Völker. Towards Interoperable Metadata Provenance. In *Proceedings of the ISWC workshop on Semantic Web for Provenance Management (SWPM)*, 2010.

[GA09] Boris Glavic and Gustavo Alonso. The perm provenance management system in action. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 1055–1058, New York, USA, 2009.

[GD07] Boris Glavic and Klaus R. Dittrich. Data provenance: A categorization of existing approaches. In *BTW*, pages 227–241, Bonn, 2007. Gesellschaft für Informatik.

[GKT07] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 31–40, New York, USA, 2007.

[GM11] Boris Glavic and Renée J. Miller. Reexamining some holy grails of data provenance. In *Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2011.

[JBM$^+$12] Claudio Jossen, Lukas Blunschi, Magdalini Mori, Donald Kossmann, and Kurt Stockinger. The credit suisse meta-data warehouse. In *ICDE*, pages 1382–1393, 2012.

[KIT10] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 951–962, New York, NY, USA, 2010. ACM.

[Lle96] David T Llewellyn. Banking in the 21st century: The transformation of an industry. In Malcom Edey, editor, *The Future of the Financial System*, RBA Annual Conference Volume. May 1996.

[MM12] Arunlal Kalyanasundaram Murali Mani, Mohamad Alawa. Algebraic constructs for querying provenance. In *Advances In Databases, Knowledge, And Data Applications. International Conference. 4th 2012. (DBKDA 2012)*, pages 187–194. Curran Associates, Inc., April 2012.

[NB04] H.R. Nemati and C.D. Barko. *Organizational Data Mining: Leveraging Enterprise Data Resources for Optimal Performance*. Idea Group Pub., 2004.

[Tan07] Wang Chiew Tan. Provenance in databases: Past, current, and future. *IEEE Data Eng. Bull.*, 30(4):3–12, 2007.

[Win01] R. Winter. A current and future role of data warehousing in corporate application architecture. *Hawaii International Conference on System Sciences*, 8:8014, 2001.