A model-driven data-analysis architecture enabling reuse and insight in open data

Robin Hoogervorst
July 2018

Master's Thesis
Master of Computer Science
Specialization Software Technology

University of Twente
Faculty of Electrical Engineering, Mathematics
and Computer Science

Supervising committee:
dr. Luis Ferreira Pires
dr.ir. Maurice van Keulen
prof.dr.ir. Arend Rensink

UNIVERSITY OF TWENTE.

Abstract

The last years have shown an increase in publicly available data, named open data. Organisations can use open data to enhance data analysis, but traditional data solutions are not suitable for data sources not controlled by the organisation. Hence, each external source needs a specific solution to solve accessing its data, interpreting it and provide possibilities verification. Lack of proper standards and tooling prohibits generalization of these solutions.

Structuring metadata allows structure and semantics of these datasets to be described. When this structure is properly designed, these metadata can be used to specify queries in an abstract manner, and translated these to dataset its storage platform.

This work uses Model-Driven Engineering to design a metamodel able to represent the structure different open data sets as metadata. In addition, a function metamodel is designed and used to define operations in terms of these metadata. Transformations are defined using these functions to generate executable code, able to execute the required data operations. Other transformations apply the same operations to the metadata model, allowing parallel transformation of metadata and data, keeping them synchronized.

The definition of these metamodels, as well as their transformations are used to develop a prototype application framework able to load external datasets and apply operations to the data and metadata simultaneously. Validation is performed by considering a real-life case study and using the framework to execute the complete data analysis.

The framework and its structure proved to be suitable. The transformation structure allows for traceability of the data, as well as automatic documentation of its context. The framework structure and lessons from the prototype show many possible improvements for the different metamodels. These provide more expressiveness for defining models, while maintaining the interoperability between different datasets.

Contents

Chapter 1

Introduction

Data can be used as a foundation for decisions within organisations. Decreased data storage costs and faster internet speeds have enabled an increase in data availability. Organisations often collect data they deem valuable and have software applications like a CRM or ERP that store data about their customers and operations.

These data hold valuable information, but extracting this information requires analysis and interpretation. This analysis is costly and requires technical expertise. Apart from the technical knowledge, domain knowledge about the information as well as context is needed to properly interpret the analysis, requiring people with a combination of technical and domain expertise on the subject of analysis. This provides barriers for effective use of many different data sources within organisations.

Internal data sources are often well structured and tooling within the organisation is implemented for this specific structure, lowering the barrier for use. To enhance this information, external data sources can be used, but these sources are not under control of the organisation and thus cannot be used easily. Because more data is becoming publicly available, there is an increasing need for a solution to lower the barrier for using external data.

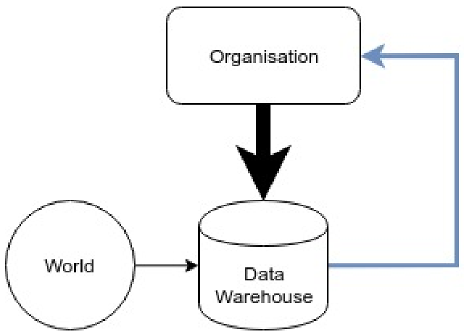
By generalizing data structure and metadata, it is possible to decrease this barrier and use data sources to find knowledge, which can be used to improve business processes and decisions. The goal of this project is to provide a solution that eases data analysis on external sources, while being re-usable and compatible with internal sources.

1.1 The impact of open data

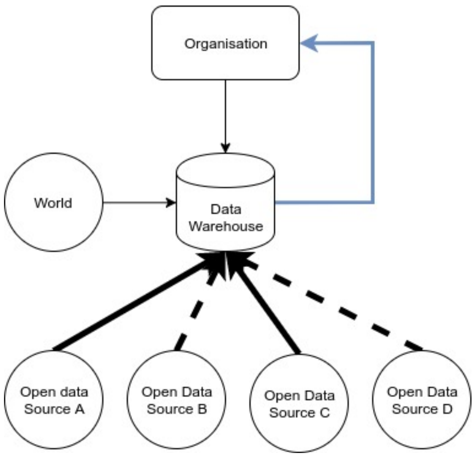
Based on the trend of rising data availability and a vision on “Smart growth”, the European Union has the vision to make its documents and data as transparent as possible. Based on this directive, the Netherlands implemented a law that makes re-use of governmental data possible [11], as of June 2015. This law caused governmental organisations to publish more and more data classified as 'open data'[19].

Open data is a collective name for publicly available data. It is based on the philosophy that these data should be available for everyone and be used freely. Because the scope of the law applies to all governmental organisations, the scope of new available data sources is very large.

These extra data change the way that organisations can use data sources, as shown in figure 1.1. Traditionally, organisations use the data generated by themselves, in addition to some data that is gathered from the world around them (1.1a). These data are structured according to the needs of the organisation and they have influence on how this is designed. Because the amount of external data is small, the benefits of such an implementation outweigh the costs and thus effort is made to import these data into its internal data sources.



(a) Traditional data flow for an organisation. Some data is gathered from the world and data is generated from applications within the organisation.



(b) Changed situation including open data. Many different sources can provide data to the organisation, but not all are relevant.

Figure 1.1: Overview of changing data flows for an organisation due to the rise of open data

In the open data situation (1.1b), most data is gathered outside. The amount of data coming originating the organisation is relatively small compared to the complete set.

Organisations do not have influence on how this data is gathered, processed and published. This means that every different data source has a different way of publishing, can have a different level of trust and has different areas of expertise. It becomes a challenge to incorporate these data, because it is expensive and time-consuming to process the data from all these different sources by hand. This challenge often means the data is not incorporated at all, neglecting the opportunities these data can provide.

To enable effective use, several challenges need to be resolved. First of all, there are technical challenges. These include different data structures, different formats, difficult accessibility, etc. Usually, these problems can be resolved when the data is loaded into a data analysis tool and scripts can be created that load the cleaned data into the tool. This process often forms a big part of time spent by data analysts, because it can become very complex. Because this process takes place before the actual tooling is used, insight in this process is lost and the transformations (and possible errors during it) become invisible.

Another challenge concerns the context of the data. Values in themselves lack any meaning. Their meaning is defined by the context that they are put into. The number 42 in itself does not mean anything, but when it is stated that the number represents "the percentage of males", suddenly it has meaning. This still is not a complete picture, as asking the question "The percentage of males in what?". The context could be further enhanced by stating it represents the percentage of males within the Netherlands. There are many questions that can be asked on what the data actually represents.

Then again, even when its exact meaning is known, the context is not complete. There is no information on, for example, when this measurement is taken or how it is taken (or calculated). This measurement might be taken only within a small group and not be representative. The measurement might be performed by a 4-year old, decreasing the trust in this certain measurement. Or someone might have calculated this number based on personal records from ten years ago.

More concretely, we state that these open data sources cannot be directly used within organisations, because:

- Open Data is published in many different formats, e.g. CSV, XML, (Geo)Json or OData API. These data need to be transformed before they can be used for analysis and visualisation.

- The context of the data (what is measured, how it is measured) is not present directly in the data itself and harder to interpret because the source is not directly from the organisation itself.
- The source may be of low quality. This includes missing values, wrong values, slightly different values that can't be compared easily or different keys to identify different meaning.

1.2 Project goal

We argue that extensive use and structuring of metadata enables the use of this context during data analysis and generalise analysis methods based on these metadata structures.

Metadata are used during data analysis to provide meaning. A trivial example is data stored in a database, where the table in the database is predefined which defines the columns (often including types), and thus structure. This table provides the context in which data can be retrieved. Usually this use of metadata is very limited and much information about the analysis result itself is kept inside the data analysts mind.

By enriching this metadata and creating a structure for it, more extensive documentation of the context of data retrieval is possible, as well as documenting data results within this enriched context.

This research aims to provide structure for users to be able to specify and use these metadata, generalized for different sources. Using modeling techniques allows us to structure metadata properly and take advantages of these structures during the data analysis. Applying this to the situation shown in figure 1.1, changes that situation to the new one shown in figure 1.2. The models should be expressive enough such that users only need this model to provide all information required. This information includes how data can be accessed, where data is stored and what the retrieved data actually means.

If well designed, these model abstractions provide possibilities for generalizing queries across different external datasets, without the need to gather all data into a data warehouse. Only an abstraction of data source is not sufficient to effectively perform these operations. Hence, a second step is to design an abstraction for these queries, ensuring compatibility with the data source models. The combination of these provides all information needed to execute data retrieval and operations on external sources.

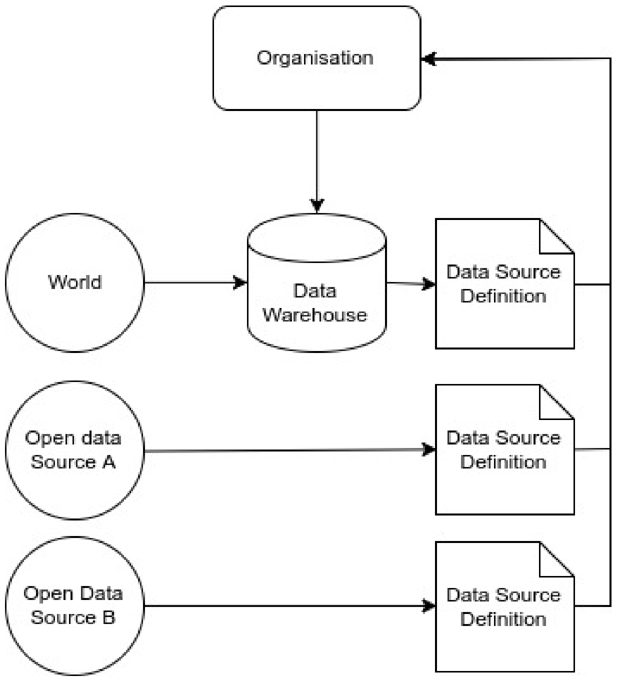


Figure 1.2: A schematic overview of data flows for an organisation using data source models

To be able to properly design these metamodels, we pose the following research questions.

RQ 1. What elements are necessary to create a metamodel able to represent existing datasets?

RQ 2. How can we create models for existing datasets efficiently?

RQ 3. What is the best method to define a generalized query in terms of this data model?

RQ 4. How can the generalized queries be transformed to executables able to retrieve data?

RQ 5. How can the context of the data be represented and propagated in the result?

This goal of this project is to define the proper abstractions, and provide an environment of transformation definitions that make these models usable. The complete package of metamodels and transformations created during this research will be referred to as *framework*. The framework is considered to be useful, when it

1. is able to load open data in a raw form,
2. allows users to put these data into context,
3. eases re-use of analysis methods on datasets
4. enables analysis methods on these data that maintains this context,
5. and allows for easy publishing of results of this analysis to the end user.

1.3 Project approach

The project goals require a method to structure abstractions and properly define these, which is why we deem Model-Driven Engineering (MDE) to be a suitable approach for solving this problem. MDE allows us to explicitly define the structure of required models as meta models. Functionality is defined in terms of these meta models. This allows us to define functionality for all datasets that have a model defined within the constraints of the meta model.

With the use of MDE comes the definition of a transformation toolchain, consisting of metamodel definitions and transformations between them. Transformations are defined in terms of the metamodel, but executed on the models. These transformations describe the functionality of the framework. This toolchain defines the inputs, outputs and steps required to generate the outputs from the inputs.

To provide an overview to the reader, the transformation toolchain used in the remainder of this report is introduced now. Figure 1.3 shows this chain. The most important models and metamodels are shown, as well as their relation between them.

The top layer represents the metamodel layer and contains metamodels for the dataset, function and raw data. The middle layer, called model layer, contains instances of these metamodels and represent actual datasets, functions and raw data sources. The bottom layer represents the physical layer. Only here is data transformed, executed and modified and upper layers only store information about the metadata.

The metamodel layer provides the definitions for the model layer, while the model layer provides the definitions that provide the base for the lower level functionality. In this transformation chain, a combination of a dataset model and function model is converted into executable code on the data.

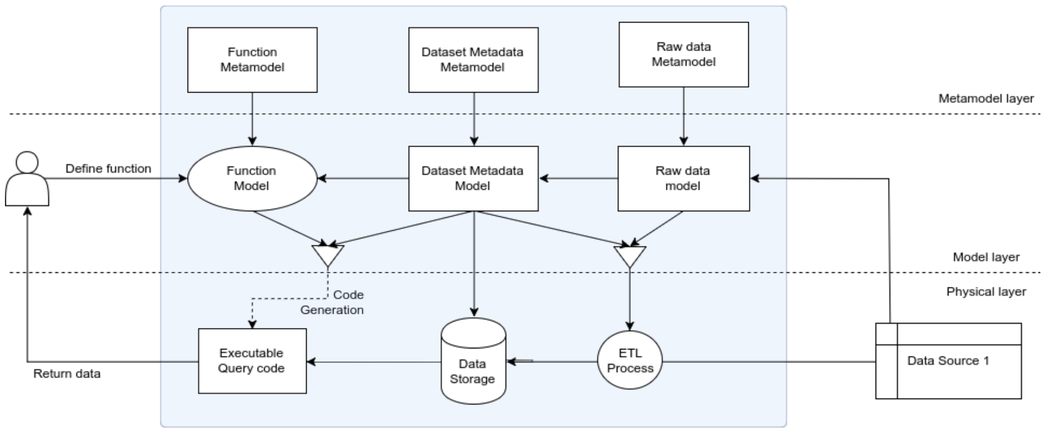


Figure 1.3: A high level overview of the steps of the envisioned solution. The dataset model forms the center, functions are defined in terms of this model and a method of converting the data to this model is needed as well. The bottom layer represents the data-flow that is needed to perform the actual analysis.

This executable retrieves the results from the data as specified by the function model.

We defined the metamodels for the dataset and function based on research on existing data analysis methods and metadata modeling techniques. Then, transformations based on these metamodels are defined that allows a user to transform these models into executable code. This executable code retrieves the data from the desired data source and provides the user with the desired result.

A prototype is implemented based on the definition of the metamodels and transformations and present how these cases can be solved in terms using the prototype. We focus on the metamodels that define the metadata and operations and deem model-mining of existing datasets out of scope for this project.

After the research and implementation, we validate usefulness of the framework, based on two cases representative for a policy-driven data analysis. These cases present challenges that arise during data analysis for policy questions, which is one of the most important use cases of open data. This validation shows a complete walk-through of how a user could use this framework.

1.4 Structure of the report

The rest of this report is structured as follows. Chapter 2 presents background information on the use cases for data analysis, as well as modern data analysis and metadata modeling solutions. This provides the foundation for decisions made in the modeling process. Chapter 3 presents the cases used for validation of the framework. Chapters 4 and 5 present the design of the metamodels and resulting DSLs for the dataset and function respectively. Chapter 6 provides an overview of the transformations in the framework, while chapter 7 provides more specific details about the prototype implementation. When the framework descriptions have been presented, Chapter 8 shows an example case using the framework and uses this as a method of validation. To conclude, Chapter 9 presents our conclusions and Chapter 10 discusses ideas for improvements.

Chapter 2

Background

Data analysis and the use of its results is already often used in businesses. They use techniques to analyse these data and use them to make better decisions. This history brought techniques to perform data analysis and strategies to apply these to policy decisions. These policy strategies are investigated in this chapter to provide a better view on the requirements of data analysis.

Similarly, solutions to perform data analysis are investigated. These include storage solutions like databases, as well as libraries to directly transform data. The last element required as background is the effort others put into describing metadata of datasets.

2.1 Data-driven decision making

The huge amounts of data available today enables opportunities for analysis and extraction of knowledge from data. Using data as a foundation to build decisions upon is referred to as data-driven decision making. The goal is to analyse the data in such a way that it provides the right information for the people that need to make the actual decision. As described in the introduction, this process changes when open data is added as an additional source. To support the modeling process, this chapter explores the different opportunities for using open data within this process.

Because we are creating an abstraction on the queries and data sources, we need to know the context and possible use cases in which we want to execute queries. This also puts the cases presented in chapter 3 in perspective. We use the model of planning and control cycle. There are many different methods and models, and because it is only used to provide context for the operations and analysis, we choose a popular one, which is the *Lean Six Sigma* model.

Six Sigma consists of an iterative sequence of five steps: Define, Measure, Analyze, Improve and Control, as shown in figure 2.1.

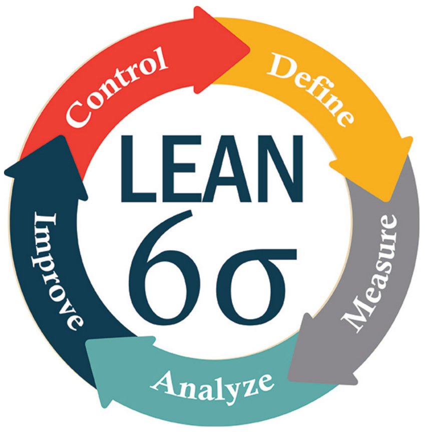


Figure 2.1: A schematic overview of the cycle of the six sigma approach

Define Based on an exploratory search through data, problems can be identified or new problems can be discovered based on new insights provided by the data.

Measure When a problem has been defined, data can aid in measuring the scope and impact of the problem, indicating its importance and priority.

Analyse Analysis on relations between different problems and indicators, enabling insight on the cause of the problem or methods to solve it.

Improve Using prediction modeling, different solutions can be modeled and their impacts visualised.

Control Data can provide reporting capabilities to validate actual improvements .

The data used for enhancing these steps usually originate from within the company. A very basic example can be an observations that sales of the company have dropped significantly. The following steps will investigate: how much the sales have dropped, what the cause is, and how it could be improved. At that point, a decision is made to change something within the company, e.g. perform more advertising. The time period after that decision, the control step is in progress to check whether the decision actually improved the sales again. Once this observation has been made, the cycle starts again.

Open data can improve these steps by providing additional information that is traditionally outside the data collection scope of the company.

Define Open data can show additional problems that the company did not consider, because there was no insight. They can also provide information on topics that the company needs information for, but has not got the resources to collect these data.

Measure External sources give unbiased information and can be used to validate observations made by the organisation.

Analyse The wide scope of open data makes it possible to more extensively investigate relationships and compare different areas of interest. For example, a observation is made that sales decreased significantly. Analysis shows that the market for the sector as a whole dropped, which may indicate the problem is external rather than internal and changes the view on the decision to be made.

Improve External prediction numbers can be used to either foresee future challenges, or incorporate these numbers in models from the company to improve these.

Control Use the additional information to gain extra measurements on the metrics that are important.

The additional value of open data is expected to be generally in the define, measure and analyse steps. Improve and control indications are very specific to the company itself, and therefore usually measured by the company itself. The define, measure and analyse steps are also targeted at gaining information from outside of the company, which is an area that open data holds information about. Cases in chapter 3 will show concrete examples of different business questions that can be answered within separate steps.

[20] takes another approach and divides qualitative data analysis applied policy questions into four distinct categories:

Contextual identifying the form and nature of what exists

Diagnostic examining the reasons for, or causes of, what exists

Evaluative appraising the effectiveness of what exists

Strategic identifying new theories, policies, plans or actions

These four categories divide the different policy questions arising from the different steps from the business improvement models.

Insight on a business level is best obtained when insights are visualised well using the appropriate graph type, like a scatter plot or bar chart. These visuals directly show the numbers and give insight in the different questions asked. It is very important to choose the right type of visualisation, because this choice has impact on how easy it is to draw insight from it. This choice is based on what needs to be shown, and the type of data. [14] identifies the following types of visualisation, based on need:

Comparison How do three organisations compare to each other?

Composition What is the age composition of people within Amsterdam?

Distribution How are people within the age range of 20-30 distributed across the Netherlands?

Relationship Is there a relation between age distribution and amount of children?

The risk is that the most insightful graphs hide data to avoid clutter. While this allows the visualisation to convey meaning, it can be misleading as well. It may be not clear how much of the data is neglected, if there were any problems during aggregation, if there is missing data, how the data is collected, etc.

Important to note is that the questions for qualitative data analysis do not directly correspond to the different graph types. Policy questions are generally too complex to grasp within a single graph. [18] defines a iterative visual analytics process that shows the interaction between data visualisation, exploration and decisions made. They argue that a feedback loop is necessary, because the visualisations made provide knowledge, which in its turn can be used to enhance the visualisations made and models underlying them. This improves the decisions.

This cycle inherently means that questions arise from visualisations, which can be answered again. When data analysis takes relatively long, this prohibits the lean approach for this data visualisation, because the people performing the data analysis usually do not have the domain expertise to generate new insights and questions from the analysis. Lowering this barrier toward non-technical people therefore greatly enhances the decision making processes.

2.2 Sources for data-driven desicion making

Data is the key component for proper data-driven decision making. Organisations often use internal data that they collect based on the metrics they aim to analyse. These data may be too limited to be able to base conclusions on, or the use of additional sources might lead to more insights that internal data alone would be able to.

To increase the amount of data used for the decision, open data can be freely used to enable new insights. Open data are generally data published by the government and governmental organisations and are published to increase transparency within the goverment, and allow other organisations to provide additional value to society by the use of these data. The main guidelines for open data are the FAIR principles [3]:

Findable which indicates that there are metadata associated with the data to make them findable

Accessible in the sense that the data are available in a standardized, open communications protocol and that the metadata still keeps available, even if the data are not available anymore

Interoperable data uses a formal, accessible and open format and complies with the open data ecosystem.

Re-usable which ensures that data are accurate as a clear and accessible usage license.

These guidelines aim for easiest re-use of data. The vision of the government to actively engage in publishing these data is relatively new, and publishing organizations themselves are still searching for the right approach to publish these data. This creates a diversified landscape and makes it harder to use these data. Although the publishing organisations try to adhere to the FAIR principles, the diversified landscape and the barriers it provides leave many opportunities for the use of these data not used.

Strictly speaking, open data could be open data if it is just a plaintext file somewhere on a server. This, however, scores very low in every aspect of the FAIR guidelines. Just publishing some files is generally not enough to let people reuse the data in an efficient manner. It is important to know more about the dataset.

Metadata is an essential element when publishing data for reuse. Users that download the data must be able to know the meaning of the numbers, they must know who published the data, they must know the structure of the data or possible codes that are used. The knowledge of these elements must be published alongside the data for it to be actually useful. To facilitate an open data platform, models have been developed that model these metadata and generalise it for the users.

Other than just the meaning of data, having multiple data sources brings additional problems for the data analysis. Different formats, structures and meaning is difficult to understand. To be able to use the data, data analysts have two choices. Either they convert all needed data into a format they are comfortable with, or they use tooling that is able to use these multiple data sources.

2.3 Data analysis solutions

We investigate data analysis solutions that are widely used nowadays. These provide the foundation and inspiration for the analysis models that we provide. In general, there are three distinct problems that a data analysis solution needs to solve:

Data structure A definition of the data structure is needed to know how the data is stored and how to access this. This can be very simple, like an array, or very complex like a full database solution. A consistent structure allows standardisation of operations, while different structures may be appropriate for different data.

Data operations The analysis consists of a set of operations that are executed on the data. Because operations need to access the data, these can only be defined in terms of the structure of the data itself.

Data loading Data needs to be loaded in the desired sturcture, which is mostly not the case. To be able to load the data appropriately in the new structure, it might be neccessary to define operations that transform the data into a suitable form.

One approach is to organise all the data required into a single *data warehouse*, which is a huge data storage in a specified format. In such a solution much effort is spend to define a suitable storage mechanism that stores the data for easy analysis. This approach is usually taken when questions about the data are known beforehand, because the storage solution is generally optimised for analysis, rather than the original data.

Data warehouses are designed for a specific use case and usually internal business information. A classical example is the storage of orders and customers to allow for analytics of orders per customer, orders per time and other metrics that indicate how well the business is performing. These data originate from internal systems with underlying data storage. Sales might originate from the payments system, a complex webapplication that running a webshop, or an internal software application for customer relations.

By defining an *ETL* pipeline, data warehouses automatically load external data into the warehouse. Just like the data storage, this ETL pipeline definition can be very complex, and is always specific for the data warehouse it is designed for. This means that the data operations used are not reusable and it is hard for users to trace the origin of the data. Reuse and interpretation of these data is highly dependent on the level of documentation that is provided.

When we step down to a lower level, we can investigate different techniques. We choose these techniques based on their widespread usage and different use case scenarios. The data analysis solutions we investigate are:

SQL or *Structured Query Language* is the best known and most used query language, and is a standardized language for querying databases. Many dialects exist for specific database implementations and their features, but all dialects share the same core. It acts on 2-dimensional data structures, referred to as tables. Usage of SQL requires the definition of these tables in the form of typed columns [1].

OLAP or *Online Analytical Processing* has specific operations to change cube structures. Its data structure differs compared to standard SQL in the sense that it uses multiple dimensions. Languages that allow querying over OLAP structures define operations that deal with this additional data structure.

Wrangling language, by Trifacta is a language that backs the graphical application that Trifacta makes, which lets users interact with their data to clean it. The steps the user performs are captured in a script in this specific language, which can be executed on the data.

Pandas library is a data analysis library for python that allows users to perform data operations on a loaded DataFrame, which is a 2-dimensional data structure.

All data processing libraries have their own vision on how data analysis should be performed ideally and differ in expressiveness, usability and

[1] although typed columns is the most used implementation, there are implementations like SQLite that do not require this

method of usage. SQL and OLAP are complete data storage and query possibilities and targeted to provide more data analytics, rather than extensive data science operations. The trifacta language is more extensive than SQL with regard to operations and transformations on the data and aims at providing a solution to be able to interactively clean your data. The pandas library provides data analysis capabilities to python. This aims partially at solving the same problems as the trifacta language, but provides an API as first-class citizen, rather than providing a language for a user interface.

2.3.1 Database storage

Database storage is a method to storage data in a permanent manner. Because they store data, they provide structure over the data they store and have methods to make this data accessible. All database solutions have methods to extract data from it.

SQL based databases are one of the oldest and stable storage solution available. Database packages like *PostgreSql*, *MySQL* or *Oracle* provide databases that can be queried using SQL. Queries are executed on a tables, which are defined in a database schema.

Such a schema describes a set of tables and which columns exist in which table. Based on such a schema, users can insert rows into the database, retrieve rows, and delete rows.

These databases provide 2-dimensional storage.

2.3.2 Pandas

Yet another option to define data transformations is the pandas library for python, which is often used in the data science community. Pandas works with the data structures *Series*, which essentially is a list of values and *DataFrames* which is a 2-dimensional data structure. The expressiveness of python allows users to define data operations in the form of simple equations that can quickly become more and more complex.

By the nature of being a python library, pandas makes it possible to use pre-defined functions on these data structures and by creating a very extensive set of functions it allows users to be very expressive with their data analysis. This allows users to define complex analysis methods and perform operations on the level of expressiveness that python provides, rather than the sometimes limited data operation functions in SQL.

This expressiveness is most notable when performing operations on a row-by-row basis. SQL defines these operations as a function in the select clause, but the user is limited to the built-in functions its specific database engine supports. Pandas allows users to define an arbitrary function based on a Series data structure and a new Series can be created that draws the value based on this function. These functions are limited to the possibilities of a python function definition, which essentially comes down to no limits.

2.3.3 OLAP

The 2-dimensional data structure of SQL has its limits, especially for aggregated data sources. The On-Line Analytical Processing cube, or OLAP for short is a storage and query method targeted at multi-dimensional data. This is one of the reasons this technology is mostly used in data warehouse.

Just like SQL databases, OLAP databases require a pre-defined schema to be able to load the data into. But because it uses a different storage mechanism, it can provide additional operations that operate specifically on the dimensions as specified in the schema.

The OLAP concept has been described by different researchers and many derivatives have been defined that differ slightly in form of definition or operations that have been defined. We will make a simple definition based on these concepts and make an illustration of the operations that have been defined. Its goal is to define a base when OLAP is referenced elsewhere in this report.

We describe the OLAP cube on basis of figure 2.2. A single OLAP cube has multiple dimensions with distinct values. For each combination of dimension values, there is a box that has a value for every metric defined. Figure 2.2 shows a cube with three dimensions, which is the largest dimension count that can be visualised easily. The cube itself, however, is not limited to three dimensions and can have many more dimensions.

While this visualisation is very simple, in practice these cubes can become very complex. Often, dimensions inherit hierarchies that aid the user in quickly collecting data. For example, there is a time hierarchy that gives an overview of sales per day. It is often desirable to view these numbers also per week, per month or per year. In OLAP cubes, these values are also calculated and stored in the cube. These data can then be queried easily, without the need for many calculations.

These dimension hierarchies are not trivial. For example, within the time hierarchy, we can aggregate days by using hourly values, and weeks by using daily values, but months cannot be calculated by using weekly values. Therefore, to aggregate months, daily values are needed. This complexity

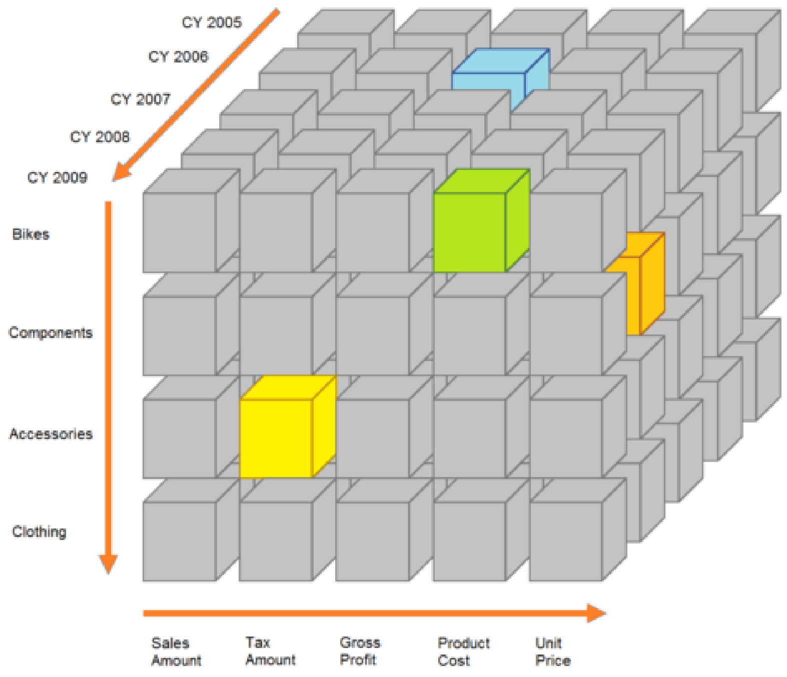


Figure 2.2: Schematic representation of an OLAP cube with three dimensions. Every smaller cube in the center represents a set of metrics and its values.

requires a definition that is able to capture this complexity.

We will use the definition of a direct acyclic graph. This allows for arbitrary parent-children relationships, while removing the complexity arising from cyclic definitions.

OLAP cubes are designed to be able to provide insight to the user interacting with it. Operations can be defined that allow a user to query the data within the cube, and extract the desired results. Although these operations may differ from implementation to implementation, the essential operations are:

Selection Select a subset of metrics. In the visualisation, this corresponds to taking a smaller section of each box.

Slice and dice Selecting a subset of dimensions. This is called slicing when a selection across a single dimension is made (because it cuts along a single axis) and called dicing when a selection across multiple dimensions is made and can be seen as selecting a sub-cube from the bigger cube. All small boxes containing the metrics are untouched, as only a sub selection of these boxes is made.

Roll-up Because the dimensions contain hierarchies, conversions between the levels of these hierarchies can be made. A roll-up is navigation to a higher level in the hierarchy. The limit is when all values for a dimension are summed up to a total value.

Drill-down Drill-down is the opposite of roll-up. It steps down a level in a dimension hierarchy. The limit of drilling down is defined by the level on which data is available.

Operations on multiple cubes are more complex, because the complexity of the dimensions needs to be taken into account. A Cartesian product, for example, creates a values for every possible combinations across the dimensions. This also means that metric values are duplicated across the cube and that the dimensions do not apply to every metric contained in the box. This makes the cube much harder to define and interpret.

2.4 Metadata modeling

Apart from the solutions for applying data transformations, metadata definitions are an essential element as well. Metadata is data describing data. Definitions of metadata are even more broad than data itself and can consist of a wide variety of properties. While the description of data structure is the most essential for data processing, other elements are necessary to provide meaning to the data.

A proper definition to describe these metadata is hard, because there are many elements to consider. Hence we start with an identification of the elements and different use cases that users require for these metadata.

Metadata is used by users to let them understand the data it represents. Essentially, the metadata should provide answers about the data that the users can ask to their selves. Questions like: Where do these data come from? or What are the quality of these data?.

A taxonomy for end-user metadata is given by [16], and presented in table 2.1. Based on an end-user's perspective (the user that is using the data after analysis), they define four categories containing information aiding the user in interpreting the data: definitional, data quality, navigational and lineage. Unfortunately, a well-defined model on what these different categories contain is missing and each of these categories is still very broad.

Category	**Definition**
Definitional	Convey the meaning of data: *What does this data mean, from a business perspective?*
Data Quality	Freshness, accuracy, validity or completeness: *Does this data possess sufficient quality for me to use it for a specific purpose?*
Navigational	Navigational metadata provides data to let the user search for the right data and find relationships between the data
Lineage	Lineage information tells the user about the original source of the data: *Where did this data originate, and what's been done to it?*

Table 2.1: An end-user metadata taxonomy defined by [16]

Different efforts have been made to standardize these metadata [7]. DCC provides a set of metadata standards that aim to describe different methods of describing metadata. We observe that many of these specifications are based on a single domain, and describe only the meaning of the data (definitional).

One of the standards described is *DCAT* [15], as developed by Fadi Maali and John Erickson. The goal of DCAT is to promote interoperability between different data catalogs, such that datasets can be indexed across different platforms without the need of duplicating the complete dataset. A dataset here is defined to be a file or set of files. DCAT standardizes properties that describe the file and properties of the data as a whole, like title, description, date of modification, license, etc. These attributes can be used during the implementation of a data catalog application that can then easily share its definitions with another catalog built on top of DCAT.

Because it is aimed towards data catalog interoperability, it does not provide information on the data itself. Within terms of the above taxonomy: it provides information in definitional and navigational context on a high level. To some extent information about lineage, because it can be queried for the source, but it is not guaranteed that this source is the original source and does not provide information about transformations applied to the data.

The PROV Family of documents [17] provides effort into generalizing the provenance of data. It *"defines a model, corresponding serializations and other supporting definitions to enable the inter-operable interchange of* *provenance information in heterogeneous environments such as the Web.”*. Based on the research for PROV, eight recommendations are provided to support data provenance on the web [8]. There recommendations focus on how to incorporate this provenance into a framework, like *“Recommendation #1: There should be a standard wary to represent at minimum three basic provenance entities: 1. a handle (URI) to refer to an object (resource), 2. a person/entity that the object is attributed to and 3. a processing step done by a person/entity to an object to create a new object.”*. Incorporating these recommendations allows for a more complete and transparent provenance framework.

While these solutions provide a standardized solution to provide metadata about the dataset as a whole, this still misses much of the context of the data itself. The data analysis solutions described above provide this information on a lower granularity to some degree, but still miss much information. SQL databases, for example, provide information the table information and some types, but lacks further description. OLAP cubes provide some additional information, but still lack much information.

Even using all these solutions does not provide much information on data quality. This quality metadata is broad, because there are many different quality issue sources that occur on different levels. On the lowest level, someone might have entered a wrong number in the database and this single value is wrong. A level higher, there could be a systematical error in a single column (e.g. leading or trailing white space), or the complete dataset could be having issues.

All in all, we observe that there is no unified method to describe metadata. The methods and models described above are used as an inspiration for our contributions. The contribution of this research for metadata is focused on how we can propagate metadata during data analysis, rather than providing a detailed model for the description of metadata. We will introduce a basic method to describe some metadata, and investigate the effect of propagating this information.

Chapter 3

Case Studies

To illustrate concepts, we will introduce two case studies. These cases originate from a discussion group consisting of 11 board members of public organisations throughout the Netherlands. This group discusses open data, use thereof inside their organisations and the impact it will have on their decision making processes. These cases thus arose from practical policy decisions that proved to be difficult because there was not enough insight for a substantiated choice.

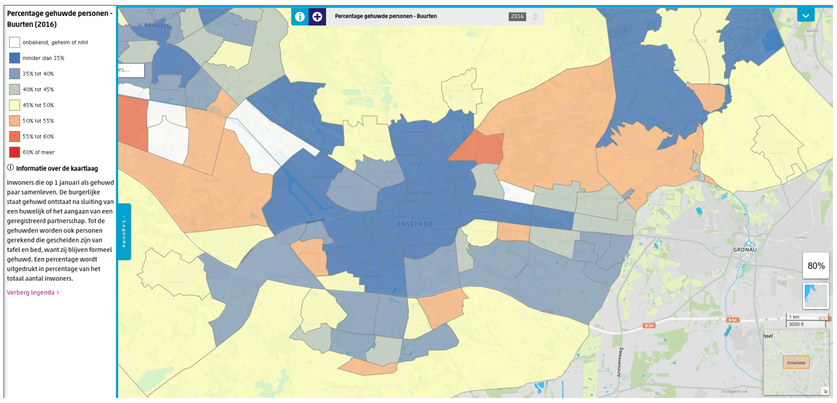
This chapter indicates the insights required for policy decisions and subsequently gives an indication of the technical elements required to generate these insights.

3.1 Case 1: Supply and demand childcare

The first case involves supply and demand for childcare services. A large childcare services organisation with multiple locations wants to open a new location. The success of a childcare location is highly dependent on the demand close to it. Without children, there are no customers and the location set up for failure.

Before making this decision, it is essential to have an indication of the demand within possible locations. This could be done based on feeling and knowledge of the decision maker, but this relies heavily on his or her knowledge and is subject to biases from this person.

By performing an analysis comparing demographic information across neighborhoods, an indication for the best possible location can be given. This can be visualised using a chloropleth map, where each neighborhood is colored with the expected demand in that neighborhood, similar to the visualisation shown in figure 3.1.


Percentage gehuwde personen - Buurten (2016)
onbekend, geheim of nihil
minder dan 35%
35% tot 40%
40% tot 45%
45% tot 50%
50% tot 55%
55% tot 60%
60% of meer
Informatie over de kaartlaag
Verberg legenda ›
Legenda
Percentage gehuwde personen - Buurten
2016
ENSCHEDE
GRONAU
80%

Inwoners die op 1 januari als gehuwd paar samenleven. De burgerlijke staat gehuwd ontstaat na sluiting van een huwelijk of het aangaan van een geregistreerd partnerschap. Tot de gehuwden worden ook personen gerekend die gescheiden zijn van tafel en bed, want zij blijven formeel gehuwd. Een percentage wordt uitgedrukt in percentage van het totaal aantal inwoners.

Figure 3.1: A screenshot of cbsinuwbuurt.nl, with a chloropleth map visualisation of the amount of married people per neighborhood

Such a visualisation requires us to reduce data sources to a single value per neighborhood, which can be mapped to a color to represent that neighborhood on the map. This number could, for example, be calculated using a model for supply and demand, which requires us to look at supply and demand separately.

The supply is defined as the amount of places available for the childcare services. This can be estimated with good accuracy, because the register with all certified locations is published as open data. The National Registry for Childcare [4] ('Landelijk Register Kinderopvang en Peuterspeelzalen' in Dutch) registers every location and amount of places available per location, which directly gives a good indication of on what locations there is a lot, of little supply. The integral dataset can be downloaded as an CSV file through the data portal of the Dutch government [6].

Based on this information, one of the analysis methods that could be performed is to plot the numbers of the amount of places for each location on a map. This quickly gives a visual overview of where there are places available. Even though this requires some technical knowledge, there are many tools available online that allows a novice user to perform this operation and view the map.

Figure 3.2 shows this process from a more low-level perspective. If the data from the register needs to be plotted on a chloropleth map with a value per neighborhood, the data from the register needs to be aggregated to this level. The sum of all places available can be counted per neighborhood. However, the register does not include the neighborhood information. We need to extend the register data with this neighborhood information by integrating it with another dataset.

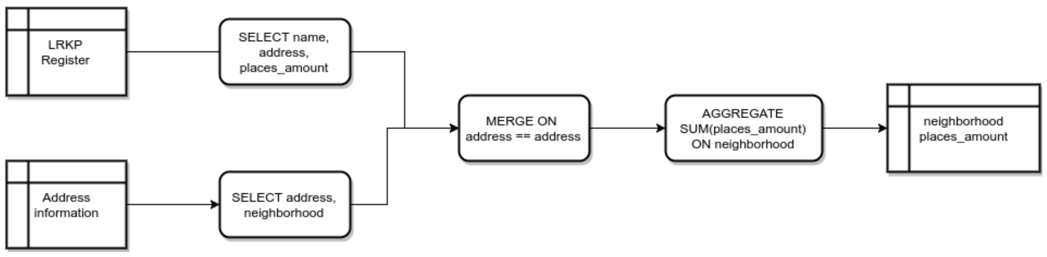

LRKP Register
SELECT name, address, places_amount
Address information
SELECT address, neighborhood
MERGE ON address == address
AGGREGATE SUM(places_amount) ON neighborhood
neighborhood places_amount

Figure 3.2: The data flow for the data to be retrieved from the register

All in all, even this simple question results in a data processing pipeline that requires us to integrate multiple datasets. When only presented with the end result, critical policy makers immediately will ask questions like: "How are these results calculated?", "What is the source?", "How trustworthy is the source?. This is because all assumptions made can highly influence the final results, and interpretation for these people is critical.

The resulting map gives insight on the places with high supply, but does not provide enough information. A location where few places are available might be a tempting, but is not relevant if no one lives in the neighborhood.
Data indicating the supply is just as important. Since no exact numbers available, an indirect approach will be used. Various demographic properties of neighborhoods can be used to provide an indication. While this does not provide us with exact numbers, it can provide the right insights. Because this report focuses on the technical results, rather than the policy information, we will use simplified model that only uses information available already as open data. This model uses the following properties as an indicator for high demand of childcare services:

- Number of inhabitants
- % of inhabitants between 25 and 45 years
- % of people married
- % of households with children

These numbers are available on a low level and provided by the CBS. While this model may not be 100% accurate, this is not essential for the analysis point that is made. The goal for this analysis is that we can easily use different indicators, incorporate these into a model and provide visualisations of these analysis.

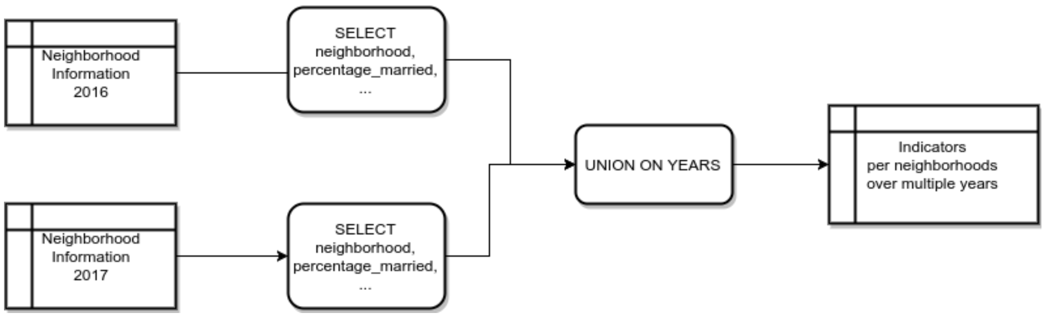

Neighborhood Information 2016
SELECT neighborhood, percentage_married, ...
Neighborhood Information 2017
SELECT neighborhood, percentage_married, ...
UNION ON YEARS
Indicators per neighborhoods over multiple years

Figure 3.3: The data flow for the data to be retrieved from the CBS

3.2 Case 2: Impact of company investments

The second cases concerns insights in the investments made by Oost NL. Oost NL is an investment company with the goal to stimulate employment within provinces Overijssel and Gelderland. The insight they require is twofold. On the one hand, they require insight in the impact on their investments. Since investments are not targeted for profit, but for economic growth, it is hard to measure. Insight in what investments do have an impact and what investments do not can aid them in better guiding their investments.

Another insight that they require is what companies are suitable for an investment. Generally, investments target companies and startups who innovate. When Oost NL wants to invest in a certain business sector, they look for companies within the regions they think it is possible to find a suitable investment. Where they look is mainly based on assumptions, which may or may not be completely off.

Much open data is available based on registers of companies. One of the registers in the Netherlands is LISA [5]. LISA gathers data from a national questionnaire send to companies and publishes open data based on aggregations of this questionnaire. Table 3.1 shows one of the open data sets that can be generated from their site. It shows the amount of business locations in that specific region, per year and per business sector, including the amount of employees summed up in that region.

These data can be used to investigate trends of growth per sector, per region and create a baseline for growth.

Another similar dataset that can be used for this purpose is the dataset "Vestigingen van bedrijven; bedrijfstak, regio"[1], as provided by the CBS. This dataset provides similar metrics, but uses the more elaborate sector classification. Another difference is that this dataset is accessible trough an

Table 3.1: An excerpt of open data provided by the LISA register

Corop	Sector	Jaar	vestigingen totaal	banen 0 t/m 9	banen 10 t/
Achterhoek	Landbouw en Visserij	2013	3830	7660	1070
Achterhoek	Landbouw en Visserij	2014	3800	7370	930
Achterhoek	Landbouw en Visserij	2015	3790	7230	1010
Achterhoek	Landbouw en Visserij	2016	3810	7200	1020
Achterhoek	Landbouw en Visserij	2017	3730	7060	1110
Achterhoek	Industrie	2013	1780	3020	11770
Achterhoek	Industrie	2014	1750	3020	11790
Agglomeratie ’s-Gravenhage	Landbouw en Visserij	2013	2070	3060	910
Agglomeratie ’s-Gravenhage	Landbouw en Visserij	2014	2050	2890	930
Agglomeratie ’s-Gravenhage	Landbouw en Visserij	2015	2090	2900	960
Agglomeratie ’s-Gravenhage	Landbouw en Visserij	2016	2140	3020	920
Agglomeratie ’s-Gravenhage	Landbouw en Visserij	2017	2160	2950	960
Agglomeratie ’s-Gravenhage	Industrie	2013	1470	2610	4040
Agglomeratie ’s-Gravenhage	Industrie	2014	1550	2650	3910
Agglomeratie ’s-Gravenhage	Industrie	2015	1640	2760	3610

open API, and thus more easily accessible without using your own data storage solution.

Investments of Oost NL are often long-term and there are many factors having influence on the performance of the companies they invest in. This makes it hard to generate visualisations that undoubtedly show the impact of their investments.

From a research perspective, it is necessary to compare the growth of the companies invested in to growth of companies that did not receive this investment. One method can be to create a visualisation in which the growth of a company is compared to the growth of its sector and region, or compare its growth with similar companies throughout the Netherlands as a whole.

Measuring growth in such a manner will never become an exact science, but can provide valuable insights. These insights can best be obtained when growth is measured across as many relevant measurement scales as possible, i.e. compare it in as many relevant ways as possible. Which comparisons are relevant and which are not are to be determined by the business experts.

Oost NL uses a topsector classification for their companies, while registers (and thus resulting data) in the Netherlands usually categorize the companies using the SBI [2] (Standardized Company division). This SBI categorisation is, however, not insightful for Oost NL because this categorisation does not align with their investment portfolio and targets.

The SBI structure is a classical example of a dimension specified as an hierarchical tree structure. The root is “Total”. The second layer represents the most high level categorisation. Then, every category is categorised in smaller sub-categories.

The topsector classification Oost NL uses is a simpler subdivision across 8 different categories. These 8 categories represent the important sectors for their innovative investments, and companies that do not fall into one of these essential categories are classified as “Other”. This allows Oost NL to focus on the companies that are important for them.

To be able to use datasets with the SBI classification for comparison, we need to be able to convert this tree structure into the more simple topsector classification. Because the SBI categorisation is more explicit and contains more information, it is impossible to accurately map the sectors to this SBI code dimension.

Oost NL has provided a mapping from SBI code to sector, for each SBI code on all levels of detail. Since the SBI and topsector classifications cannot be mapped directly, It does, however, give a appropriate indication of companies to sector.

Chapter 4

Dataset modeling

This chapter introduces the first element of the proposed solution, being a dataset metamodel. This metamodel describes the structure for a model that describes the metadata of a dataset. A DSL is generated off of this metamodel that can be used to create model files for different open data sets. Such a model then directly represents the metadata of the dataset.

4.1 Metadata

The metadata should aid users, as well as machines, to be able to read and interpret the data. While users mainly use it to understand the data, machines process the data and need to understand it in their own manner. A proper metamodel is able to fulfill these tasks for a wide variety of data.

Chapter "background" did show techniques to describe metadata. There is, however, no existing metamodel suitable for our goals, requiring the definition of a custom one.

The elements required in this metamodel depend on the definition of "to understand" within the context of the data. To identify and classify what elements belong to this, we take a pragmatic, bottom-up approach, based off of questions that the metadata should be able to answer. We noted these and classified them into the following 6 categories.

Origin What is the original source? Who created or modified it? How old is it? Who modified the data? How is the data modified?

Distribution How can it be accessed? How can I filter or modify data?

Quality Can I trust the data provided in this dataset? Are all measurements complete? Are all measurements done at the same type/in the same manner? Can I use the data without validation?

Scope What region/time/values does it cover?

Structure What is the size of this dataset? What are relations between different elements?

Interpretation How should I interpret it?

More formally, to make the data understandable the metamodel should provide an abstract interface that allows processing steps to be defined and applied, independent on the data its concrete representation. Additionally, it should capture additional information about the context of the data. The combination of these two elements creates a structure allowing interaction and processing of the dataset while containing the information about its context.

4.2 Data structures

As an starting point, the two different datasets needed by case 1 will be analysed using these questions. Case 1 concerns supply and demand for childcare services and mainly uses two different data sources. The first source is the register of childcare services and the second source is the regional information from the CBS. These sources are representative of many open data sources, as we will discuss later.

The childcare service register provides an overview of every location in the Netherlands where children can be taken in. An excerpt of this dataset is shown in table 4.1. These data are published in CSV format and is structured such that every row represents a single childcare service. For each service, it provides the type of service it offers, its name, its location, amount of places and responsible municipality.

This source is representative of different registers of locations, companies, buildings or organisations that may have data that is structured in a similar format. Such a source contains a row for each instance and has columns for each of its properties.

Table 4.1: An excerpt of the register childcare services with the headers and 5 rows of values representing childcare services in Enschede

type_oko	actuele_naam_oko	aantal _kindplaatsen	opvanglocatie_adres	opvanglocatie _postcode	opvanglocatie _woonplaats	cbs_code	verantwoordelijke_gemeente
VGO	Hoekema	4	Etudestraat 45	7534EP	Enschede	153	Enschede
KDV	Peuteropvang Beertje Boekeloen	14	Boekelose Stoomblekerij 27	7548ED	Enschede	153	Enschede
VGO	Zwart	4	Bentelobrink 128	7544CR	Enschede	153	Enschede
VGO	Ramjiawan Mangal	6	Padangstraat 68	7535AE	Enschede	153	Enschede
VGO	Reve-Kompagne	4	Kruiseltlanden 7	7542HC	Enschede	153	Enschede

The other data source for the first case originates from the CBS, and is accessible through the CBS' OData API. In addition to providing the raw data, it has capabilities to filter the data, perform simple operations or retrieve additional metadata. Listing 4.1 shows an excerpt of the raw data response. Because the dataset itself is too large to show in this report (62 properties), only a single metric and the two dimensions are selected.

Listing 4.1: An excerpt of how the response looks like from the OData API from the CBS

```
{
  "odata.metadata":"http://opendata.cbs.nl/ODataApi/OData/70072ned
      /$metadata#Cbs.OData.WebAPI.TypedDataSet
      &$select=Gehuwd_26,RegioS,Perioden",
  "value":[
    { "Gehuwd_26":11895.0,"RegioS":"GM1680","Perioden":"2017JJ00" },
    { "Gehuwd_26":6227.0,"RegioS":"GM0738","Perioden":"2017JJ00" },
    { "Gehuwd_26":13181.0,"RegioS":"GM0358","Perioden":"2017JJ00" },
    { "Gehuwd_26":11919.0,"RegioS":"GM0197","Perioden":"2017JJ00" },
    { "Gehuwd_26":null,"RegioS":"GM0480","Perioden":"2017JJ00" },
    { "Gehuwd_26":null,"RegioS":"GM0739","Perioden":"2017JJ00" },
    { "Gehuwd_26":null,"RegioS":"GM0305","Perioden":"2017JJ00" },
    { "Gehuwd_26":11967.0,"RegioS":"GM0059","Perioden":"2017JJ00" },
    { "Gehuwd_26":null,"RegioS":"GM0360","Perioden":"2017JJ00" },
    { "Gehuwd_26":9118.0,"RegioS":"GM0482","Perioden":"2017JJ00" },
    { "Gehuwd_26":10960.0,"RegioS":"GM0613","Perioden":"2017JJ00" },
    { "Gehuwd_26":null,"RegioS":"GM0483","Perioden":"2017JJ00" }
  ]
}
```

The structure between these two data sources may seem disparate, but they are actually very similar. Both are a list of grouped values. The CSV file groups the values by row, and identifies values by the header on the first row. The OData result explicitly groups these values as sets of key-value pairs. When the keys for each set are the same, these data structures are identical albeit in a different representation.

The CSV format could be easily converted to the OData result by generating key-value pairs based on the column header and the value in its column. Every row then represents an entry in the set, and the value for each column forms a key-value pair within this set. The OData repsonse can be rendered to CSV by extracting the keys to headers, and placing the values in the corresponding columns.

This two-dimensional structure is often seen in exported data and especially open data sources. It is convenient for exports, because it is very simple. More complex structures tend to be hard to distribute and interpret further.

Because this structure is so common, we limit the supported datasets by only supporting data that can be represented as a list of sets. The implication for our metamodel is that it should accurately describe the properties of each set. Structure can be generalized, with the condition that a method is need to identify the representation.

Another important aspect is the interpretation. This can be split up into interpretation of each individual key, each individual value and the set as a whole.

The key does not provide much information. The key "Gehuwd_26" in the CBS data leaves the user in the dark with its exact representation. It could be guessed that it represents the amount of married people, but this is still not enough. Questions arise such as: Which people are taken into account? How are these people counted? What does the number "26" mean within the key?

In addition to the information the key should provide, the value provides information itself as well. Each value says something about its group, but not all values are equal. Isolating the column amount of places ('aantal_kindplaatsen') yields the values: $4, 14, 4, 6, 4$. The values are a pretty sequence, but do not convey any meaning. Isolating the names column provides the sequence: "Hoekema", "Peuteropvang Beertje Boekeloen", etc. These names are not valuable on their own as well, but they do provide a means to identify a single instance, and thus the topic of the group.

Combining these two sequences yields key-value pairs that match the amount of places to the name. Adding the other information from the dataset like location and type adds even more and more information about the instance.

The difference between the different types of values lies in the fact that one column can be removed without loss of meaning, while the other one cannot. When the column with amount of places is removed, the location of each instance is still known and instances can be identified. In contrast to the name or location, as information about the context is lost upon removal.

The CBS data source can be analysed in the same manner. In this dataset, the column "Gehuwd_26" can be removed without loss of context for the other variables present in the dataset, but columns "RegioS" and "Perioden" cannot. The childcare dataset has one column that can identify the childcare service and each row represents the information of a single instance. In the CBS data source, the combination of both "RegioS" and "Perioden" is needed.

We identify the columns that cannot be removed without loss of meaning to be the identifying columns, similar to a primary key in SQL databases. When the names of the childcare instances is used as the identifying column, this column cannot be removed without loss of information. These identifying columns play a key role in determining the scope of a dataset, its topics and in combining multiple datasets.

4.2.1 Dimension and metrics

The identifying columns are an essential element of the context of the dataset, and thus essential metadata. A method to fundamentally capture these properties into the dataset is by classifying a column to be either a dimension or metric. Dimensions and metrics form the foundation of OLAP databases (section 2.3.3). Yet, definitions and interpretations of dimensions in datasets differ in academia. Based on the observation about columns that can or cannot be missed, we consider a data feature to be a dimension if *its row value is necessary to define the context of the value of the metrics in the same row.*

When classifying a column to be dimension or metric, its role in identifying the subject of the row is the deciding factor. If the value describes a property of the subject, it is considered to be a metric, if it puts the subject into perspective or gives an indication of partitioning of a value, it is a dimension.

Our definition provides some useful properties. First of all, the complete scope of the dataset can be identified by just inspecting the dimensions. Because these dimensions describe what the data is about, their values describe the complete scope.

This dimensional information can be further enhanced. Different keys can describe different types of a dimension. One example is the use of a dimension that describes time-related information. By adding such a type to the dimension, the scope of dimension, regional and topics quickly become apparent.

Sometimes, the structure of the data obstructs the actual use of dimensional information. Table 4.2 shows an example which provides information on the amount of people in a specific region (specified by a code). In this case, the region code is the subject, as all values describe a property of that region.

Based on our earlier definitions, we should interpret the dataset as having a single dimension and three metrics: “Total people”, “Male” and “Female”.

Region	Total people	Male	Female
GM0153	1500	800	700

Table 4.2: An example dataset

The dataset could be just as well be represented using a single metric with "Amount of people" and two dimensions for region and gender, which would transform the dataset into the form in table 4.3. Because it now is a dimension, it directly identifies in the metadata that information about the differences in gender is present, and that the scope is the complete set of people.

Region	Gender	Amount of people
GM0153	Total	1500
GM0153	Male	800
GM0153	Female	700

Table 4.3: The example dataset in a pivoted form, using an additional dimension

The method of converting this information is known in OLAP terminology as a *pivot* operation. One solution is to create an extraction process that extracts this information when importing such a dataset, but doing so would require a high level of complexity in querying and function definition. Besides that, a problem occurs when the dataset has additional metrics that are not represented in that dimension. The example shown in table 4.4 has two additional columns that are not related to the gender dimension.

Region	Total people	Male	Female	Native inhabitants	Immigrants
GM0153	1500	800	700	1350	150

Table 4.4: The second example dataset

Yet, doing such operations is useful to change the structure in a manner that allows the definition of more context. Both methods of representation have their advantages and disadvantages. Ideally, it should be possible to represent the dataset in both methods and be able to choose the appropriate model when starting an analysis.

4.2.2 Aggregated vs. non-aggregated data

The difference between the two data sources is the level of aggregation. Dimensional data structures are generally used to describe aggregated data, and the aggregation means are used as the dimensions.

For data sources that represent measurements on single entities, an specific dimension type is used called "Identity". Its values identify a single entity and requires the data source to have a feature that can be used as an identity.

4.2.3 Origin and quality

Elements important for traceability of the dataset are its origin and quality. These two properties are not essential for data analysis, and often not considered in database models. But, traceability is a key element of the dataset once important decisions are based off of these data, which is part of the interpretation of the data.

The origin of the data can be split up into two categories, being the original source and the operations that have been performed on the data. These metadata should provide insight into the complete path from data aquisition to presentation of the data.

This includes information about the author, organisation that the author belongs to, method of publishing people who modified it. The credibility of the author or its organisation can have a big influence on the trustwhortyness of the data.

Another closely related metadata caterogy is the quality of the data. When data quality is higher, better conclusions can be based on these data, while low data quality can lead to skewed results.

Data quality is, like the origin, dependent on the source as well as the operations. During the data aquisition step, there is much room for error. When data originates from applications, for example, the data quality depends on the quality of the application. Some applications may force a user to enter data, or restrict the types of data that can be put in, which increases consistency, and thus quality, across exported datasets.

Operations can alter quality as well. During execution of operations on raw data, there is little guarantee that all operations are valid and provide valid results. Some operations might neglect empty values and use these if there are values, or some operations remove values that should not be removed.

4.3 Dataset model

The above observations need to be modeled in a formal metamodel. We propose the metamodel shown in figure 4.1. The root element is the “Dataset” element, which contains a key, name and description, as it is a child of the `Element` class. The `Element` class is used for the most elements in the metamodel, because many elements need a key, name and description.

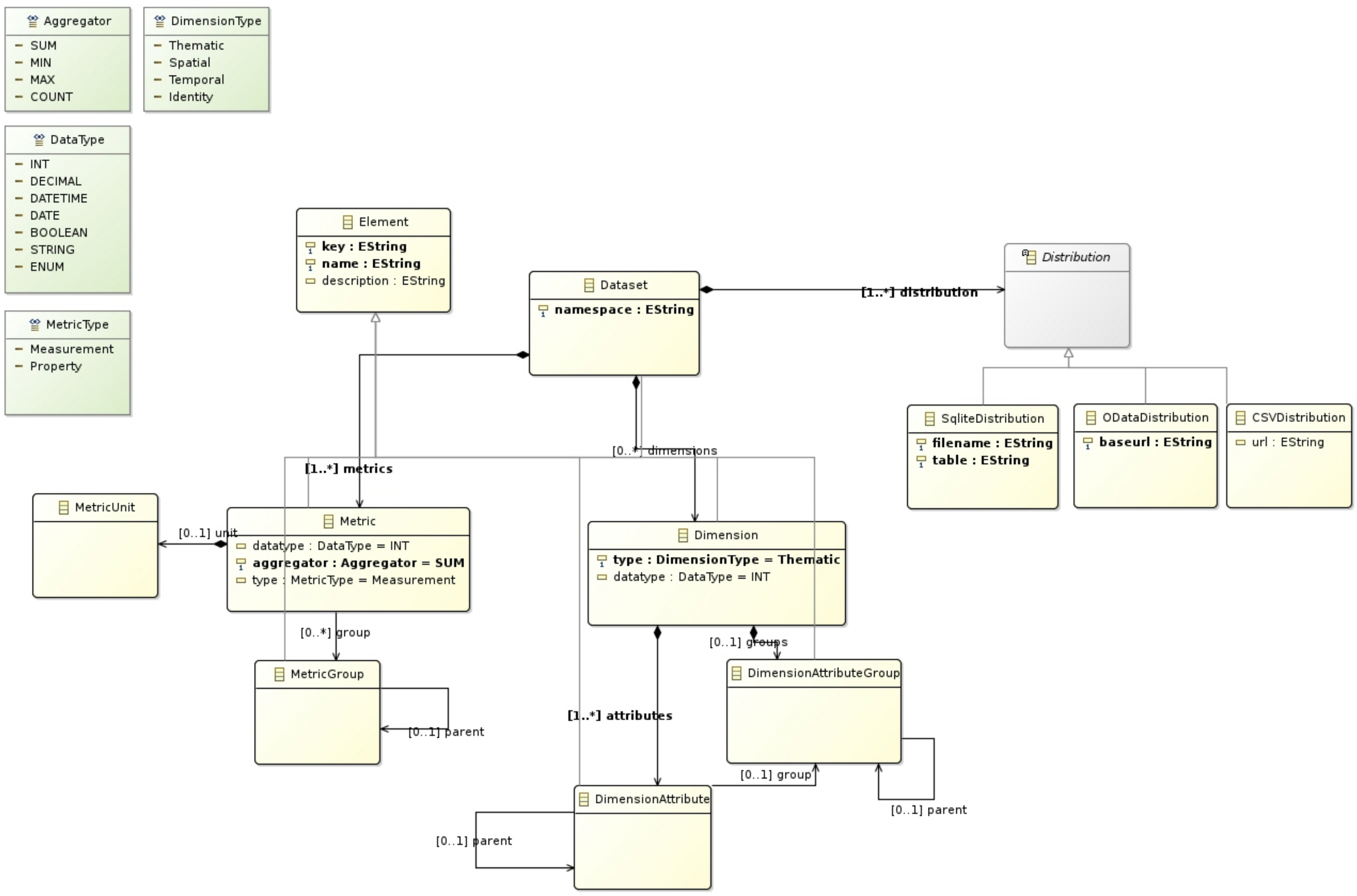


Figure 4.1: The dataset metamodel based on the dimension-metric model, including features as groups for dimensions and metrics, aggregation functions and different dimension types

The dimensions and metrics form the foundation for the metamodel. The usage of dimensions allows the distinction between values that provide meaning to the context of the data and actual measurements. Because its usage is widespread, it allows us to build upon the results and experiences of OLAP technology for data analysis and focus on how to re-use different concepts.

The metamodel is a combination of different features provided by existing data metamodels.

- The dimension-metric model used by the CBS. We consider this model to be strong in definition of description of different properties and the grouping of metrics and dimensions.
- The dimension-metric model described by Boulil et al. [13], which is strong in defining typing and the operations that can be performed on the different types.
- The open data standard DCAT [15], which is strong in providing context of the dataset itself

Not all properties required for the metadata are directly described by properties in the dataset model. Table 4.5 shows an overview of which metadata properties are covered in which manner.

Table 4.5: An analysis of the metadata categories and the level the metamodel supports

Origin	The definition of author and distribution shows the source.
Distribution	Definition of distribution defines the location and type of dataset. The keys of the dimensions and metrics define how these can be accessed.
Quality	Validation can be done based on the units of measurements.
Scope	This is implicitly contained in the dimensional information.
Structure	The combination of dimensions and metrics shows the amount of columns and typing of different columns.
Interpretation	Fine-grained descriptions allow people to specify information about the dimensions and metrics. Typing of different elements allows users to

To show the usability of this metamodel, we will describe the examples in terms of this metamodel and consider the advantages it provides us with.

Each column in the table needs to correspond to either a dimension or metric in the dataset model. This distinction already gives us a good indication of the meaning of the values.

Listing 4.2: The dataset model description for the dataset of childcare services

```
Dataset {
    key: lrkp
    name: "Landelijk register kinderopvang"
    description: "Dataset met alle register kinderopvang gegevens"
    dimensions: {
        type: identity
        key: id
        name: "ID"
        description: "Identiteit"
    }
    metrics: {
        type: "Measurement"
        key: aantal_kindplaatsen
        name: "Aantal kindplaatsen"
        description: "Het aantal kindplaatsen binnen deze instantie"
    },{
        type: "Property"
        key: type_oko
        name: "Type kinderopvang"
        description: "Het type kinderopvang"
    },{
        type: "Property"
        key: actuele_naam_oko
        name: Naam
        description: "Naam van de kinderopvang instantie"
    },{
        type: "Property"
        key: opvanglocatie_postcode
        name: "Postal code"
        description: "Postal code of the instance"
    },{
        type: "Property"
        key: opvanglocatie_huisnummer
        name: "Huisnummer opvanglocatie"
        description: "Huisnummer van de opvanglocatie"
    },{
        type: "Property"
        key: opvanglocatie_straat
        name: "Straat van de opvangloctie"
        description: "Straat van de opvanglocatie"
    }
    distributions: CSV {
        url: "file:///home/lrkp.csv"
    }
}
```

For each value in the original data file, it needs to provide information on what properties these data points have, what they mean, and how we can perform operations on these sets of data points. Every row in this file is an individual data point. We can analyse this model definition based on the context definition given earlier.

The metamodel representation for the CBS dataset is given in listing 4.3.

Listing 4.3: Dataset model representation for the CBS OData dataset

```
Dataset {
    key: wijk_kerncijfers_2017
    name: "Kerncijfers per wijk"
    description: "Aantal kerncijfers per wijk in Nederland"
    dimensions: {
        type: Spatial
        key: WijkenEnBuurten
        name: "Neighborhoods"
        description: "Wijken en buurten"
        attributes: {
            NL00: "Nederland",
            GM1680: "Aa en Hunze",
            WK168000: "Wijk 00 Annen",
            BU16800000: "Annen",
            BU16800009: "Verspreide huizen Annen",
            WK168001: "Wijk 01 Eext"
        }
    }
    metrics: {
        type: "Measurement"
        key: Gemeentenaam_1
        name: "Gemeentenaam"
        description: "Testing"
    }, {
        type: "Measurement"
        key: AantalInwoners_5
        name: "Aantal inwoners"
        description: "Test"
    }, {
        type: "Measurement"
        key: Mannen_6
        name: "Aantal mannen"
        description: "Het aantal mannen per regio gebied"
    }, {
        type: "Measurement"
        key: Vrouwen_7
        name: "Vrouwen"
        description: "Het aantal vrouwen"
    }
    distributions: CSV {
        url: "test"
    }
}
```

Due to the additional metadata the CBS provides and semantics its datasets have, modeling the results from the CBS's OData API is more straightforward. Its aggregated data is based on dimensions and metrics, and our model implements a similar pattern for dimensions and metrics. Because the semantics already inherit dimensions and metrics, there is no design decision in what columns should be defined as dimension, and which as metric.

These groups allow us to give additional meaning to a big list of metrics for larger datasets and make it easier for an user to browse and filter these.

The dataset model provides a foundation to describe different properties of the metadata. These properties are essential to make data-driven policy decisions. Some metadata elements are not described in the dataset model. This model describes a state of the dataset. This includes structure, documentation for interpretation and method of accessing. It does, however, not necessarily include complete history on the data.

Describing history of the dataset needs additional structure. We opt to provide this structure by providing metamodel defnitions for transformations on the data, and thus metadata. The next chapter presents our view on solving this problem using structured operations, and providing additional transparency during data operations.

Chapter 5

Function modeling

This chapter presents the DSL specifying the function that defines data operations, described in the form of a metamodel.

Performing operations on the data requires the definition of a function. This function enables data-analysts to specify their intentions and generate their desired results. They create a function model in terms of the function metamodel, and use the framework to transform the function to executable queries such as SQL or executable code that retrieves the results.

The definition of these functions should aid in providing information on the metadata elements stated in the previous chapter.

First, we show how the function model fits into the model driven framework. This should be be designed such that results can be reused, as well as direct use of results, while still maintaining a appropriate level of expressiveness of the language.

After that, we will show its usefulness from a data oriented perspective. The metamodel needs to be constructed in such a way that the data analysis it supports is actually useful. The dataset model has already taken a step towards this goal, by using the dimension-metric model often used in data analytics.

5.1 Functions transformation structure

The most common problem with metadata definitions is the level of outdatedness. Metadata may be present, but is made obsolete when analysis is performed on this data. To be able to have keep the proper descriptions, the data analysts must create and modify this after every data operation step. Because this is time consuming, this is often neglected and the resulting data is not reusable, since other people cannot interpret or understand the results.

We opt to maintain the same levels of documentation, by structuring the documentation and operations on the data. Using a model driven solution allows us to specify the models in such a way that the documentation information can be propagated. This propagation allows the data analyst to directly have documentation about the resulting data, and provide this documentation alongside the dataset to inform people reusing the data.

Consider an analysis about the percentage of households with children across neighborhoods within a single municipality. Analysis is needed to determine if households with children have been declining in the city center. Such information can help case 1, regarding supply and demand for childcare services.

As stated, Statistics Netherlands publishes data about neighborhoods every year in a separate dataset. The information to determine a trend is available, but separation across multiple datasets encumbers retrieval in the desired format. Data from all datasets with years of interest need to be combined into a single result. This result can then be used to generate graphs and analyse it to achieve the desired result.

A possible solution is to perform an ETL process that extracts information from the separate datasets, gets the right information and stores its results. The result is a file or database that contains the data needed analysis. This works well, but insight in the process is lost and because only the data is processed, metadata about underlying sources is lost. A person using these results has no clue about the origin of the data and cannot reproduce the steps taken.

If the analyst continues working and publishes visualisations or reports based on the results, the meaning of these data is derived from the implicit knowledge the data analyst has. Errors during this process are not transparent, and all implicit knowledge not documented is quickly lost after the data analyst continues with the next project.

Using the dataset model, operations can be defined that are able to modify the dataset model in such a way that the desired dataset is derived. Because these dataset models represent underlying data, executing the same operations on the data generates matching results. A combination of these operations can be used to define a proper function meant for analysis.

We wish to design a process that puts focus on creating a dataset with the desired results, rather than transformations on the data itself. The actual transformations on the data should then follow logically based on the transformations on the dataset. After this transformation, the result is a dataset that can be as easily used as the sources, because it has the same level of metadata, descriptive properties and query possibilities. Automatic generation of an ETL process based off of this function definition provides the possibility to transform data and metadata accordingly.

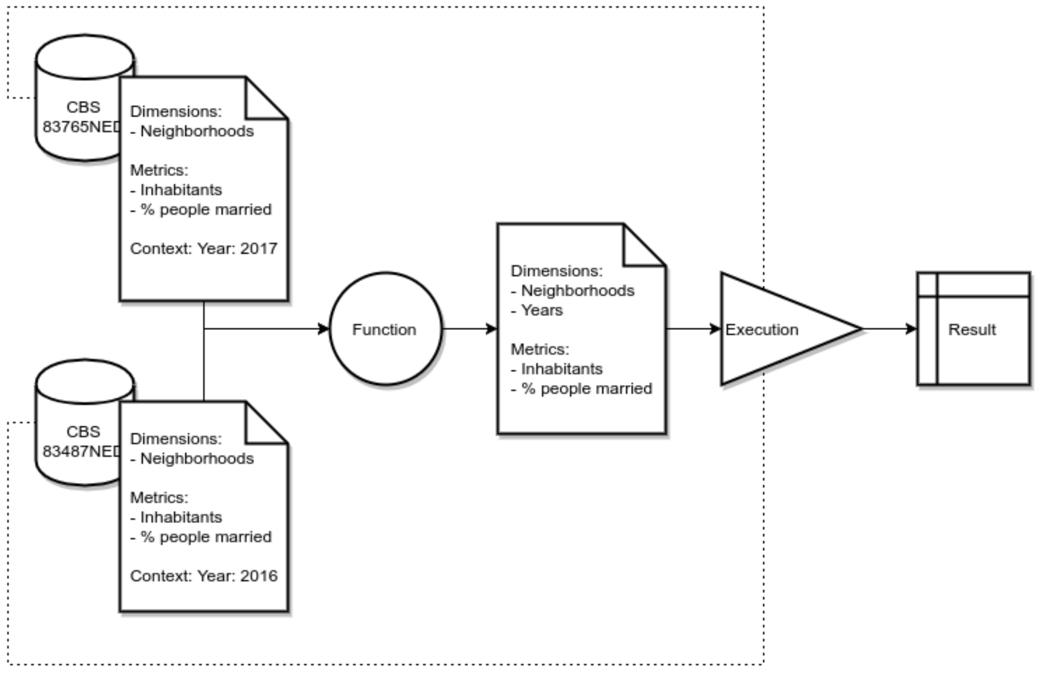


Figure 5.1: An example of the function transformation application to a merge of the CBS neighborhoods datasets

The process of such a transformation applied to the example above is shown in figure 5.1. Two dataset models are taken and a new dataset model is created containing the information of both, accompanied by a new dimension. This new dataset model now represents the resulting data, thus the user has an overview of the meaning of the data and can share this metadata definition to help others interpret the data as well.

The two original datasets include implicit and explicit context information. Explicit context information contains descriptions about the metrics, and a spatial dimension containing neighborhoods. The year of publishing of the dataset is present in the description, but not as dimensional information. This makes it implicit and not directly usable. When using the dataset on its own, it is irrelevant to contain this information explicit in the dataset, but it can prove to be useful during analysis. To create the new dimension, the function must take care of formalizing the temporal information into the model.

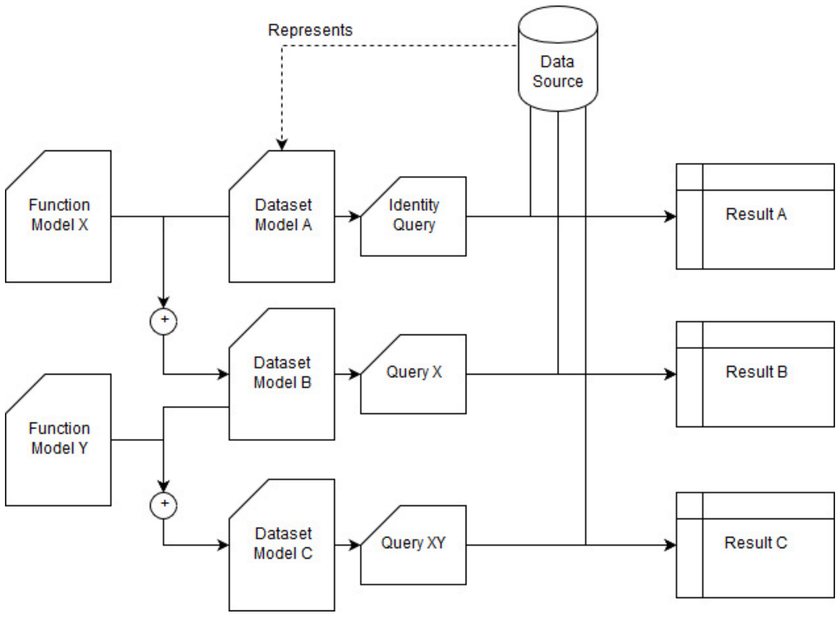


Figure 5.2: A schematic overview of the reuse of functions with relation to datasets and storage.

To realize this, a method is needed to describe and retrieve the data source of resulting datasets. This is solved by defining an identity query, as well as allowing functions to be a source of a dataset. The combination of these allows users to generate a dataset using a function, and not having to physically store its result. This maintains transparency of the operations executed, as well as allowing multiple metadata representations of the data. The data needed can be transformed on-demand using the function definitions present in the metadata model.

Figure 5.2 shows how this can be applied recursively. Dataset A represents a physical data source. By executing an identity query, the results for this dataset are retrieved. A function can be defined to perform calculations on this dataset. Function X describes a function that takes dataset A and generates a derived dataset B. To retrieve the results of the newly generated dataset B, the corresponding function X is executed on the original data source. This result can, in its turn, be used to generate another dataset. Dataset C is defined by applying function Y on dataset B. Because it is known that dataset B is the combination of query X on dataset A, the results of dataset C can be retrieved by combining query X and Y and executing it on the original source of dataset A.

Enabling this process requires a carefully designed function. Context and metadata information may become very complex, and thus hard to propagate. Even more so, some operations may be invalid based on its context.

It is a challenge to determine the level of incorporation of the semantics into this process. In the CBS example, the actual datasets contain more metrics than the two shown. These metrics are mostly similar to each other, because the CBS designed its metrics this way. But there is no guarantee that methods of retrieval of these metrics are identical throughout these datasets.

More generally, the combination of multiple datasets requires matching different properties, albeit on column or on row level. This matching process is error-prone when data quality is not 100%.

5.2 Metamodel definition and design

There are roughly two methods of defining a data transformation function. One can choose an expression style, where an expression is defined on how to retrieve data. It is a good method, since it allows users to think in terms of the data they require, rather than the method of retrieval.

The other method is to choose a sequential style where each operation is executed after the other. ETL processes or the pandas library are good examples of this. This method defines the function as a starting point and a set of ordered operations to executed on this starting point.

Our function metamodel definition uses the definition of a sequential style, because

- It results in more readable queries. These are more easy to reason about.
- The ETL processes have proven to be effective to be able to use them on data operations. New, more complex, operations can be created easily and allow for easy reuse
- It simplifies the process of investigating the metadata transformation definitions. It is sufficient to show that each operation can transform a dataset to a new dataset. This property is directly true for the complete function, since operations can be appended to each other.
- It allows easy merging of functions; complete groups of operations can simply be appended.

We propose the metamodel as shown in figure 5.3. Its root is the *Query* element that contains expressions. Each expression is a chain of operations with a single starting source. Different sources can be combined by using specific operations that have additional sources as an input. These inputs can consist of a subexpression.

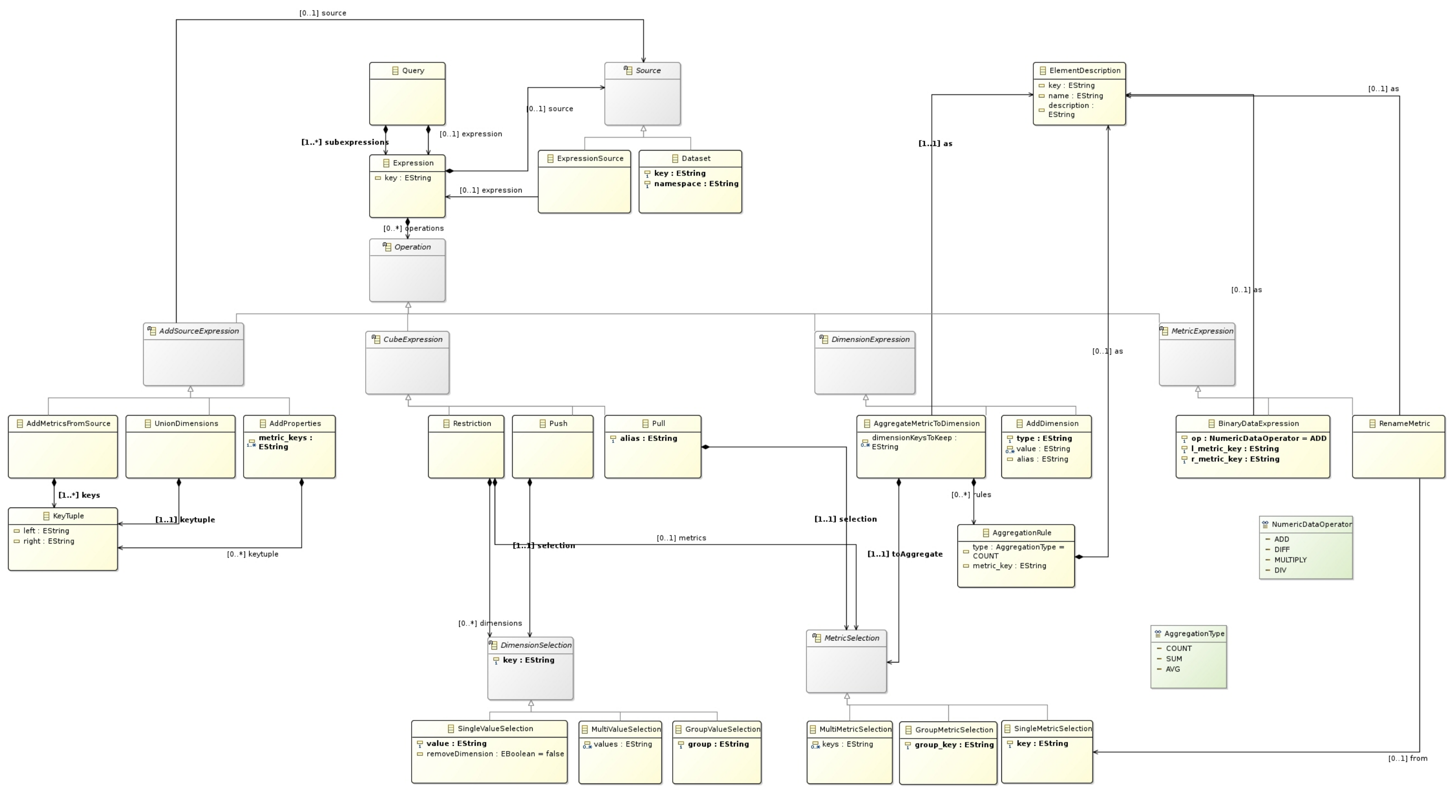


Figure 5.3: Overview of the function metamodel, including all its elements

The structure of the metamodel is simple, but is quite large due to the large amount of operations. The top section of themetamodel is shown in figure 5.4. This section holds the root object `Query`, as well as the important `Source` elements.

A query consists of multiple expressions, of which one is the main expression. All subexpressions require a key, such that they can be referenced by an expression source. This method of referencing allows operations to reference these sources for merging purposes. The operations referenced by the expression are a ordered list of elements belonging to the abstract `Operation` element. An expression, thus, holds a source that represents the original data and a set of operations that can be executed consecutively on this data source.

The dataset source definition requires a method of retrieval. During execution, the dataset referenced in this model are needed, and thus need to be retrieved. Ideally, models are registered globally, easily accessible, and thus easily referenced in the model. While implementation and proper use can be complicated, we consider this a solved problem. Many package managers for programming languages take an approach that identifies packages by using a combination of namespace, key and version and publish packages in a global registry accessible through HTTP. Examples of this include npm (for javascript), cargo (for rust) and rubygems (for ruby).

For the prototype implementation, we opted for a simple option similar to a complete package management solution: a single key and a local folder containing files where the key of the dataset matches its filename. This allows the transformation executions to simply read the file from this folder (because the filename is known by its key). This solution is easily transferable to a more complete package management approach using registries.

The other sections of the metamodel represent the complete set of operations available. Details of all operations and their classifications are presented in the next chapter. Figure 5.5 shows the category containing operations to merge multiple datasets. Since a more elaborate description is given below of all operations, the remainder of visual details of these operations are omitted.

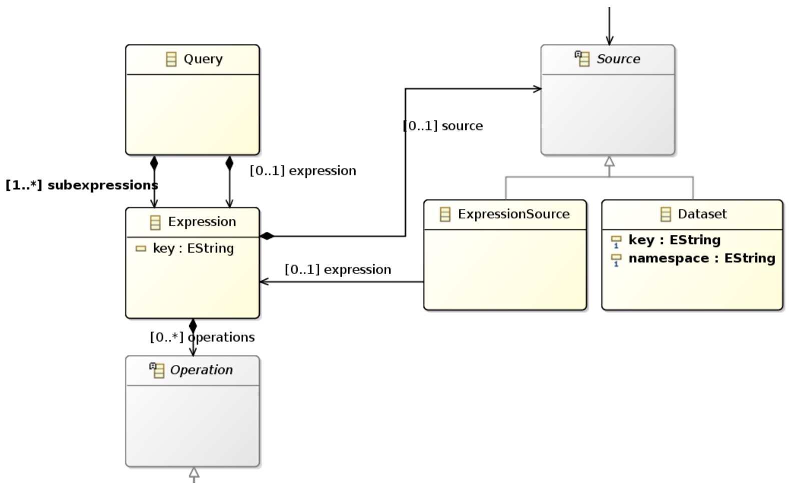


Figure 5.4: The top section of the function metamodel, containing the root element, source elements and reference to the abstract operation element.

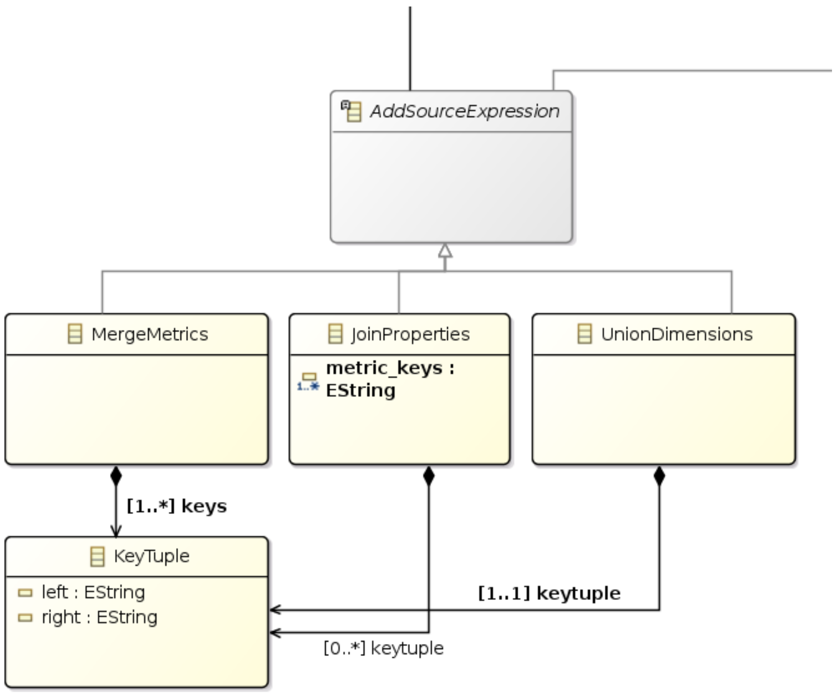


Figure 5.5: The bottom-left section of the function metamodel, containing the operations for merging multiple datasets.

5.3 Data operations

The most essential element of the function metamodel is its usability for analysing data, and consequently the operations it supports. Our proposed structure requires that all operations must be executable on the data, as well as the metadata. Data operations are well researched and can be specified easily, but the challenge is to create a subset of these operations such that every operation can be executed on the metadata in a meaningful way, while not prohibiting the data operations required.

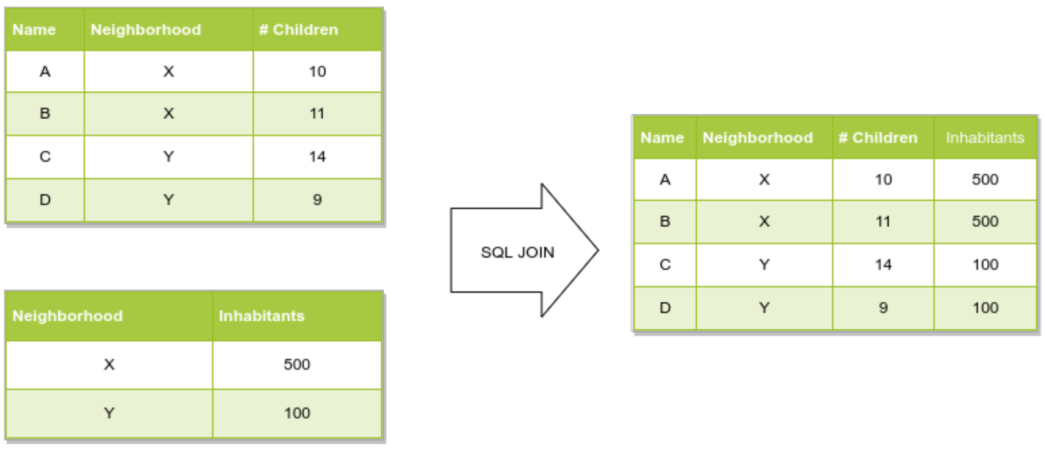
The scope of data operations and possibilities is huge, and we deem therefore the definition of the operations based on a few example cases insufficient for a proper definition. The focus of this research is not to provide a complete query language, but rather to demonstrate the vision of model driven engineering on the data analysis perspective. We therefore aim to define a subset of the operations used by other languages, show that these are useful and define how the metadata should be transformed accordingly, while neglecting the proof of being a complete data analysis language.

Data operations are not unique across the different languages, and definitions of how the data is changed for each operation is defined. The impact on the metadata, however, is not. A proper definition of these these metadata operations is needed. Based on an analysis of the operations, we categorize the operations as follows:

- Operations that only change values
- Operations that change the structure of the dataset
- Operations that combine multiple sources

These categories differ in the level of impact they have on the metadata. The more contained an operation is, the easier it is to transform the metadata. If an operation changes a single metric value across the dataset, the input dataset model can be mostly copied and propagated, except for that single metric. It becomes more difficult when the complete structure of a dataset changes, and when multiple sources are combined, their models must be combined as well and challenges arise on how to perform such an operation properly.

Consider the example shown in figure 5.6. The upper left table representing childcare services, with an column indication in which neighborhood the service is located. The lower left table represents a table containing information about the inhabitants for each neighborhood. Using SQL, we can easily join these tables on neighborhood code, resulting in the right table.



Name	Neighborhood	# Children
A	X	10
B	X	11
C	Y	14
D	Y	9

Neighborhood	Inhabitants
X	500
Y	100

Name	Neighborhood	# Children	Inhabitants
A	X	10	500
B	X	11	500
C	Y	14	100
D	Y	9	100

Figure 5.6: An example of when an SQL join loses contextual information. The inhabitants of the neighborhood are not on the same level as the amount of children per service.

While this is a valid data operation, the inhabitants metric is fundamentally different than the children one. Consider the case when using this result, we would like to get the information per neighborhood. This means the table will be aggregated on neighborhood, and the amount of inhabitants now has invalid values.

This indicates that not every data operation can be executed naively. Some data operations have requirements on the data before they can be executed. Before performing these operations, these requirements have to be fulfilled. To not prohibit these functions, additional operations must be defined that aid the user in fulfilling these requirements.

5.3.1 Operations overview

This section provides an overview of the operations specified. For each operation, requirements are defined as well as impact the operation has on the data along with the metadata. The operations are tagged with a level, according to the impact they have on the metadata. These are defined as follows:

Multi-dataset level Operations that combine several datasets in a certain way. These are the hardest to perform, since they require the combination of multiple metadata models.

Dataset level Operations that change the structure of a dataset.

Dimension level Operations that also change the structure of a dataset, but only over a single dimension. These are easier to reason about, compared to the change of the complete structure of a dataset.

Metric level These only change one, or multiple, metrics. Since these operations only change the values, and not structure, they are the easiest to reason about.

❯ Aggregation (Dimension) (dimension level)

Metamodel representation as: AggregationOverDimension

Aggregate values using an aggregation function like sum, or mean. Such a function takes a group of values and returns a single result. The aggregation needs input that determines which rows need to be grouped together.

Requirements:

- An aggregation function is needed for every metric
- A mapping for which dimension values will be aggregated is needed

Impact on data: The amount of rows is decreased, and is reduced to the amount of distinct rows in the column that is being aggregated upon. Furthermore, for every metric an aggregation function is defined and its values thus change according to this aggregation function.

Impact on dataset:

- Information and values of metrics change. This change should be reflected in the description of the metric
- The attributes of the dimensions that is aggregated on changes, according to the mapping. The dataset should reflect the originating dimension attributes.
- Dimensions that are not aggregated over stay equal in description and attributes.

❯ Aggregation (Metric) (dataset level)

Similar to the aggregation over a dimension from a data perspective, but has a much bigger impact on the dataset structure. Aggregating over a metric has the implication that this metric is converted to a dimension, where each value in the dimension represents a group (or range) of metric values. Each of the values in the calculation for the aggregation is based on the identifier of the data in the dataset, and thus all other dimensional information in the dataset is lost, and the original dimensions are used as base of counting.

Such an aggregation is most notable when targeting a dataset that represents entities, like a register. Such a dataset has a single dimension for the entity identification, but several properties that might be interesting. For example, we would like to aggregate over the region code that is provided as a property for the metric. An example of this is shown in tables 5.1 and 5.2.

Table 5.1: Table before aggregation over the region metric

location	region	aantal_kindplaatsen
A	WK015301	40
B	WK015301	40
C	WK015301	20
D	WK015302	40
E	WK015302	16
...	...	...

Table 5.2: Table after aggregation over the region metric

region_code	aantal_kindplaatsen	aantal_locaties
WK015301	100	3
WK015302	150	6

It can be seen that many information in the dataset, especially regarding information specific for each organisation. These properties could be used as an additional aggregation option, such that rows with that specific property are grouped and added into an additional dimension. Because this operation has so much impact, basically the complete structure of the dataset changes. This is desirable when new dimensions should be created because those are the ones of interest.

Requirements :

- For each other metric than the one that is aggregated over, a aggregation function is needed
- A specification is needed on how to aggregate the metrics. This can be either range-based, e.g. with numerical values, or based on exact match, e.g. on properties of a certain row.

Impact on data: The columns from the original dimensions are lost, because these are not relevant anymore.
Impact on dataset: The impact on the dataset is huge, as this operation basically alters the complete structure of the dimensions and metrics. In short:

- All original dimensions are lost, unless specifically set
- A new dimension is created, based on the mapping for the metric aggregation
- Metric that are aggregated need to be annotated in the dataset, similar to an aggregation over a dimension

› Merge metric values (multi-dataset level)

Suppose two datasets exist that have different metrics, but their dimensions match precisely. These two datasets can be merged, such that there is a resulting dataset that keeps the dimensions it had and now has the union of the metrics present in both datasets.
Requirements :

- Dimensions, including meaning and attributes, match precisely between two datasets
- Metric keys are distinct across datasets

Impact on data: Rows on left and right dataset need to be matched properly, based on the dimension attributes. The amount of columns gets extended by the amount of metrics that are appended from the new dataset.

Impact on dataset: Impact on the dataset is minimal. Dimensions stay equal, and the metrics of both datasets are merged into a single set.

› Join properties (multi-dataset level)

Despite the fact that SQL joins might lose context, it might sometimes be necessary to perform such an operation. This is the case when the user wants to aggregate over the property of a relation. Suppose we have a dataset about grades of children, and we want to aggregate these numbers based on the total budget of the school. The dataset can have a property for each child with the school, and there might be another dataset that contains info for each school.

In this case, the data that is added cannot be considered to be a regular metric.
Requirements :

- The base dataset contains a property
- The added dataset contains a single dimension, where each dimension attribute corresponds to the value of the property to be matched in the base dataset. This means the right dataset always matches to exactly one or no row.

Impact on data: Columns are added based on which properties are joined from the added dataset. The amount of rows in the resulting data is equal to the rows in the base dataset.
Impact on dataset: Impact on the dataset is similar to the merge based on metrics, with the only difference that the metrics from the added dataset are typed as properties, rather than measurements.

❯ Merge based on dimension (multi-dataset level)

When two datasets are similar in the sense that they provide the same data, but have one dimension scattered across two sources, these datasets can be merged on this scattered dimension. This is, for example, the case for the splitted data about neighborhoods from the CBS. Because they publish data about different years,
Requirements :

- Metrics are equal across datasets
- There is a single dimension that has the same meaning and type, but differs in attributes.
- All other dimensions match precisely

Impact on data: Because the actual data structures are very similar, the data operation is as simple as appending the actual data sources.

Impact on dataset: Impact on the dataset is minimal. Most features are identical, only the attributes of a single dimension are extended to reflect this increase of scope of data.

Table 5.3: An example of a data export where dimensional information is present, but encoded as metrics

Buurten	Bevolking 0-4 Man 2016	Bevolking 0-4 Vrouw 2016	Bevolking 5-9 Man 2016	Bevolking 5-9 Vrouw 2016
0101 Binnenstad Centrum	95	107	56	64
0102 Binnenstad Oost	25	29	30	11
0103 De Hofstad	19	21	25	12
0104 Binnenstad Noord	13	1	6	9
0105 Het Zand	26	34	12	20
0106 Vughterpoort	14	4	22	6
0201 Het Bossche Broek	0	0	0	0
0202 Zuid	64	85	74	74

› Pull

Sometimes, dimensional information is present in the dataset, but not in the right format. The pull operation is defined in OLAP terms and converts a set of metrics to a new dimension and new metric. This new dimension has an attribute for each of the metrics that is used to create this dimension.

An example of the required transformation is shown in tables 5.3 and 5.4. This table is an export from the amount of inhabitants in each neighborhood in 's Hertogenbosch (a municipality in the Netherlands), published by the buurtmonitor. In table 5.3, it can be seen that there are many colums that contain a very specific value for, essentially, a selection within a dimension. The issue is that this information, and thus context, is not explicit and thus not directly usable by data analytics platforms.

In order to use it, the data must be transformed into the form shown in table 5.4.

Requirements :

- A metric for each attribute used in the dimensions
- A mapping of which metrics are used
- A description of the newly generated metric
- Every metric must be used to generate the dimensional values. The dimensional attributes must provide context over the metrics in the dataset, so if these are not used this transformation does not make sense.

Table 5.4: The table in 5.3 should be transformed into this format, in order to be able to use the dimensional information.

Buurten	Age	Gender	Year	Bevolking
0101 Binnenstad Centrum	0-4	Male	2016	95
0101 Binnenstad Centrum	0-4	Female	2016	107
0101 Binnenstad Centrum	5-9	Male	2016	56
0101 Binnenstad Centrum	5-9	Female	2016	64
0102 Binnenstad Oost	0-4	Male	2016	25
0102 Binnenstad Oost	0-4	Female	2016	29
0102 Binnenstad Oost	5-9	Male	2016	30
0102 Binnenstad Oost	5-9	Female	2016	11
.....				

Impact on data: The amount of columns is decreased at the cost of adding additional rows. Every row is replaced by a new number of rows that equals the amount of attributes for the dimension that is pulled from the columns. Essentially, the data from a single row and multiple columns is transposed to a single column and multiple rows. Additionally, a column is added to represent the attribute of the dimension for that particular row.
Impact on dataset: A new dimension is created, with the specified attributes, based on the columns. Metrics are mostly deleted, and new more general ones are added.

› Push

The push operation is the opposite of the pull operation and converts the attributes for a dimension into separate metrics. This can be used when the dimension needs to be removed in order to match the dataset with other cubes, or if the metrics are to be used separately for calculations. More generally, this is a method to reduce the amount of dimensions in a dataset.
Requirements : Because dimensions are more descriptive and contextual than metrics, the information about the dimensions can be used to generate the new metrics.

- There are at least two dimensions, one of which can be pushed. This means that the attributes of the dimension must be known in the dataset model. This means identity dimensions cannot be used to push.

Impact on data: Amount of rows are decreased, columns are increased.
Impact on dataset: All metrics are replaced by new metrics that now contain the specific values for the related dimension attribute value.

› Expression (metric-level)

This operation adds a new metric off of existing metrics in the dataset on a row-by-row basis. Because the values originate directly from metrics in the same row, this operation is simple.
Requirements :

- All metrics referenced in the expression are present in the dataset

Impact on data: A new column is added with the values from the expression. This value must be calculated using the values in the other columns in a row-by-row basis.
Impact on dataset: A new metric is added, with a description and name as provided in the function.

› Restriction

Restrict dimensions by attributes, or remove metrics from the dataset. In OLAP dimensional terms, this essentially means cutting a part of the cube. Because filtering a range of a metric value is different in terms of the metadata, this cannot be done using the restriction operation.
Requirements :

- The dimensions attributes or metrics that are filtered upon are present in the dataset

Impact on data: A restriction on dimensions changes the amount of rows in the dataset, while a metric restriction removes columns.
Impact on dataset: Dimension attributes and metrics are removed based on the specifications of the function. When dimension attributes are removed, it changes the scope of the dataset, which requires additional notes on the dataset.

› Add dimension

Every dataset has dimensions, but this information is not always included. Measurements are taken for a specific year, or for a specific region, or only the results for the age group 0-18 is included, etc. If this information is not present, adding a dimensions with a constant value can aid in putting the dataset into context.

This newly generated context can be used to merge the datasets with others or used as notes in the selection for an interface.
Impact on data: A new columns with a constant value is added.
Impact on dataset: A new dimension with a single attribute is added, named according to the input of the function.

❯ Rename

In order to add descriptions and change keys of the data, there is the ability to rename. On the one hand, this serves the purpose of being able to change keys of metrics that allows for merging of multiple datasets. On the other hand, this can provide better descriptions of data.
Impact on data: The key of a single column changes
Impact on dataset: Key, name and description of a single dimension or metric change.

5.4 Dataset merge operations

One of the essential elements in data analysis is combining several datasets into a single one. An attentive reader might have noticed that the framework highly restricts possibilities to merge these datasets. While SQL allows a user to combine tables and match these on arbitrary values, this is restricted in the operations specified.

These restrictions are made to guarantee meaningful results and metadata compatibility. Furthermore, if the dataset merging operations are not restricted, it becomes impossible to define a proper metadata transformation. This forces the user into using the operations that change the structure of a dataset, such that dimensional structures match before performing the merge.

Merging datasets is based completely on using dimensions as an identification mechanism. Since these dimensions define the scope of the dataset, restricting this makes sure that the scope of these datasets can be matched properly.

Using this method makes the transformations of dimensions a critical element. The operations therefore provide multiple options to change these dimensions. These operations act on lower levels can thus be more easily captured in separate functions. When a dimension change is captured in a single function, there is a resulting dataset with this altered dimension. The first user doing this uses this result for a new operation on the higher level to merge two dataset. But the result with the altered dimension can be reused for other people that would like to use this altered dimension.

Dimension transformations are harder than they seem at first sight. For example,

- When comparing the amount of inhabitants between municipalities in 2018 and 2010, the dimensions for region do not match. Every year, there are changes in the structure of municipalities. Several municipalities might merge into a new one, or one municipality joins a bigger one. To properly compare these, the changes for each year need to be propagated.

- A comparison of two datasets that contain an age group dimension, where dataset A is split into groups of 5, while dataset B is split into groups of 10.
- Two datasets with a temporal dimension have an overlapping time range, but dataset A also includes earlier data.

Dimensions are assumed to be compared using their attributes. If attributes in a dimension are equal, they are assumed to represent the same element. When combining dimensions, both dimensions are to be converted to a common set of attributes. An identification on cases to be encountered is required to assert that a solution is present for each of these cases. The dimension-level, as well as the dataset level, have to be considered. On the dimension level, it is required that two dimensions can be matched properly using their attributes, while on the dataset level it is required that all dimensions can be paired.

When considering if two dimensions can be matched, their attributes must be compared. Since, these attributes are two sets of values, set theory can be used to identify the possible cases:

1. A equals B
2. A is a proper subset of B[1], and the identical case vice versa
3. A is not B, but do have similar attributes, i.e. their intersection is not empty
4. A is not B, and their intersection is empty

The first case is when the attributes of dimensions match exactly. In this case, no transformations are necessary to convert the dimensions to an equal state.

The second case can be solved easily. If A is a subset of B, we can match each attribute in A to an attribute in B. By definition, B has additional

[1]we use proper subset, because this does not include the case where both sets are equal, which would be identical to the first case

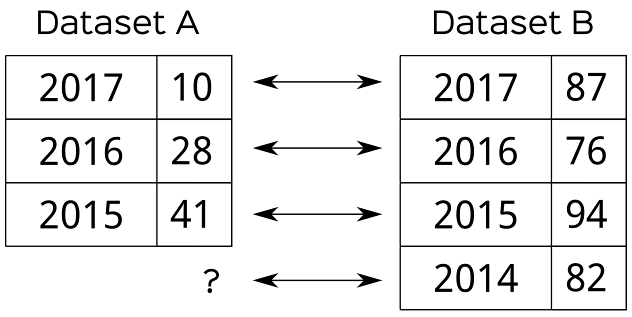


Figure 5.7: An example of two datasets with difference in dimension

attributes that cannot be matched to ones in A. Because there is no use for this data in this comparison, these attribute values must be removed from this dataset, which can be done using a *restriction*.

The third case could be solved similar to the second case, but a transformation for both dimensions is needed to limit them to the attributes within the intersection. This does, however, lose much data.

The fourth case cannot be solved by a subselection of attributes. In many cases, these dimensions probably do not describe the same element and can therefore not be matched, but this is not always the case.

In an instance of the third or fourth case, many attributes are unused and certain cases allow for another solution. If possible within the semantics of the data, attributes from A can be converted such that they match an attribute in B. Consider the case when dimensions for region codes of municipalities in 2017 need to be matched with 2018. Every year, there are certain municipalities that fuse together and form a new one. This comparison thus will have a big set of municipality codes that did not change. But 2017 will have municipalities GM0007 and GM0048, while 2018 will have the new municipality GM1950, which is a fuse of those two. With the aid of a mapping which municipalities have merged, we can transform the 2017 dataset with aggregations into this new one.

This is also the case when two dimensions describe, for example, age ranges. A can have attributes “0-10”, “11-20”, “21-30”, etc. While B has “0-5”, “6-10”, “11-15”, etc. Obviously, the values for B can be aggregated to the same values as A, but they do not have any common attributes.

When investigating a complete dataset, the only additional complexity comes from matching the dimensions. The dimensions must form pairs in order to apply the technique specified above. Each matched dimension pair can be transformed individually. An issue occurs, however, when a dimension is present in dataset A, but not in B. In this case, a pair cannot be created and such a missing dimension must be created, or the surplus dimension must be reduced.

If possible, the dimension can be created by executing a *pull* operation that extracts the dimension information. If needed, this new dimension can be transformed to suit the paired dimension. Additionally, the *add dimension* operation can be used to add a constant dimension, to be used if dimensions are merged. When creating a dimension is not possible, the dimension cannot be used for merging and must be reduced.

Reducing a dimension is different to removing all its values, because this would remove the complete dataset, it rather should be converted into a single value. This can be done by selecting a single value, or by aggregating the dataset over this dimension. The method required differs based on the dataset and the data that is needed.

This selection of a value can change the context of the dataset. For instance when a dimension provides the attributes ”Male”, ”Female” and ”Total”. Creating a selection based on the “Total” attribute keeps the global scope of the dataset, while selecting “Male” or “Female” can drastically change the context.

Defining all these transformations can become cumbersome for a user, but this is not considered a huge issue. This method provides a well-defined method of combining datasets in a meaningful manner, while simultaneously forcing the user the think about the structure of the dataset and its results. We expect this hugely decreases errors made during the data analysis process.

This structure also allows tool development that automate large parts of this process. Tools can be developed that analyse two dimensions and generate a function that can match these dimensions. Or analysis programs can be made that analyse on what level the datasets match and provide data quality scores.

Additionally, allowing function reuse would allow users to define such a dimension transformation once and use it across many different datasets. Implementing this is possible within the structure of the function metamodel definition, but considered out of scope for this project. Section 10.7 discusses this in more detail.

5.5 DSL definition

While the dataset DSL definition is a simple, JSON-like definition and is directly derived from the metamodel, using this technique for this function metamodel results in a unreadable, verbose mess. We define a more concise, elegant solution to solve this problem.

The shortened structure for the language is given in lising 5.1. Elements have been left out to preserve readability, but the complete main structure is shown.

Listing 5.1: Shortened DSL definition to parse text into the function metamodel, defined in the XText grammar language

```
Query: ('with' subexpressions*=SubExpression)? expression=Expression;
SubExpression: '(' expression=Expression ')' 'as' key=ID;
Source: ExpressionSource | Dataset;
ExpressionSource: '$' key=ID;
Dataset: namespace=ID '/' key=STRING;

Expression: 'select' source=Source
    operations+=Operation (',' operations+=Operation)* (',')?;

Operation: RestrictionOperation
    | RenameMetricOperation
    | AddMetricsFromSource
    | MetricExpressionOperation
    | PushOperation;
RestrictionOperation: 'restrict'
    ('metrics' ('to')? metric=MetricSelection)?
    ('dimensions' (dimensions*=DimensionSelection))?;
RenameMetricOperation: 'rename' metric=SingleMetricSelection
    'to' key=STRING;
AddMetricsFromSource: 'add metrics from' source=Source
    'with' 'dimensions' dimensions*=KeyTuple;
AddDimension: 'add dimension' type=STRING alias=STRING ',' value=STRING;
MetricExpressionOperation: 'calc' ('ulate')?
    expression=MetricExpression 'as' description=Description;
```

Most importantly, the `Expression` statement shows that a source is specified using the `select` keyword and then references a source. This source is either a combination of namespace and key, or an reference to an expression using a key with a dollar sign. This is used to be able make this distinction during parsing.

Based on this definition, the earlier example on combining neighborhood datasets can be defined more formally. This definition is similar to the process shown in the case description. It consists of sequential transformations on the dataset with the functions provided in the metamodel. This defined in terms of the metamodel is shown in listing 5.2.

Listing 5.2: A function definition for the case description, written in the textual semantics defined above

```
1  WITH (
2      select wijk_kerncijfers_2016
3      restrict metrics on keys Mannen_6, Vrouwen_7
4      add dimension constant "Perioden", "2016"
5  ) as wk_2016,
6  (
7      select wijk_kerncijfers_2017
8      restrict metrics on keys Mannen_6, Vrouwen_7
9      add dimension constant "Perioden", "2017"
10 ) as wk_2017
11 select wk_2017
12 union wk_2016 ON "Perioden" = "Perioden"
```

Lines 1-4 represent a subexpression with key `wk_2016`. Like every expression, this expression consists of a source selection (line 2) and a set of expressions. In this case, there are 2 expressions defined. The first expression on line 3 restricts the metrics to two metrics, while the operation defined on line 4 adds a dimension with a constant value to this dataset. Lines 6-10 repeat this same pattern the other dataset, which can be referenced with key `wk_2017`.

Line 11-12 represent the main expression. This is apparent because the expression does not have brackets, as well as lacking a key specification.

The added dimension constants are used as the dimension to combine these datasets on. Because both datasets represent a different slice of a dimension, in this case time, these slices can be directly appended to each other (taking into account the requirements of this operation). This union operation is defined on line 12.

This function definition now can be used to define complex functions on the dataset. It provides a concise definition language. Although the syntax feels like a SQL language, it is not structured like such and must be interpreted as a sequential set of operation definitions. According to the definitions provided in this chapter, it is possible to transform the data, as well as the metadata using these operations.

Chapter 6

Data transformations

The function metamodel and dataset metamodel allow us to describe the metadata and operations in a structured manner. These operations are defined such that metadata can be preserved. Yet, a proper definition of the metadata transformations, as well as a path to execution, are still missing. This chapter describes the transformation processes which transform the dataset model, and allow transformation of the data.

6.1 Transformation goal

The function definition specifies data- and dataset transformations such that the transformed dataset represents the metadata of the transformed data. The data and dataset model are, however, not transformed in the same manner. The dataset transformation can be specified directly in terms of the metamodels, but there is nu such abstraction for the data source. The structure is known, as this is described in the dataset model, but the data itself is not contained in a model. Consequently, the data operations cannot be done using a model transformation.

The dataset transformation will use traditional MDE transformations. For each function operation, a transformation step is to be implemented that transforms the dataset in a specific way. All these possible transformations based off of the operations are already defined in section 5.3.1 , which leaves this transformation to be a implementation detail.

The data transformation requires an intermediary step. There are many data analysis platforms focused on data transformations and a re-implementation of such a platform is deemed unnecessary. Rather, these platforms can be utilized to perform the actual data operations.

Using Model-Driven Engineering, code generation can be used to generate executable code based on model definitions. Since there are many suitable languages suitable for the data analysis as target code, this technique can be used to generate code that lets such a platform execute the operations defined in a function model.

6.2 Data transformation target

Model transformations allow us to, theoretically, easily define transformations such that code can be generated for different platforms. Based off of the models, code is generated for a platform and passed into its runtime environment. When a platform executes the code, it performs the data operations and results from that code execution can be retrieved.

Support of multiple platforms requires an implementation structure suitable for choosing different target platforms. Specific code for each platform is needed, but for best interoperability it is required to minimise platform-specific code. Transformations and metamodels need to be as much reused as possible to do this properly.

The problem of code generation can be split into two smaller problems. On the one hand, a definition of how to perform code generation is needed. This is platform specific and can differ highly between platforms. On the other hand, platform-specific code needs to be minimised and the reuse of existing models and transformations be maximised.

To simplify, a single target platform is used to demonstrate how the process for a single platform operates. Because SQL is widely used for data operations and storage, this is the choice for the platform demonstration. Code generation steps are defined that generate SQL code to be executed on a database platform.

One implementation method is the definition of a direct transformation from function model to code. As the abstraction gap between the function metamodel and SQL is too big to generate SQL code based on the function metamodel in a straightforward manner. This method requires a definition of transformation for each execution platform.

Another option is to generate a metamodel for each code generation target. This creates a 2-step process in which a transformation to this intermediary metamodel is performed, after which this metamodel is used for the actual code generation. The transformations to the intermediary metamodels can be trivially reused, while also simplifying the code generation step.

The 2-step process allows flexibility in defining the abstraction level of

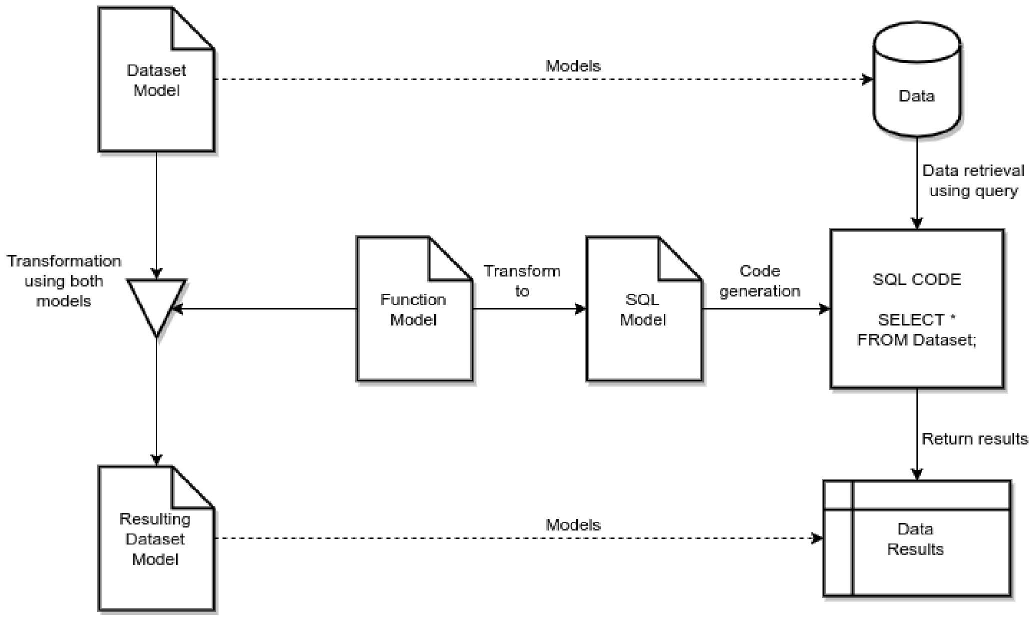


Figure 6.1: Overview of how the transformations for the function model, dataset model and data relate to each other.

the metamodels for code generation, which can be used as an advantage for reuse of models. It is a design decision to either create specific metamodels or create a general one with low-level data operations. Using more specific models generally increases the difficulty for transformations to that model, but makes code generation simpler, while code generation based on a general metamodel can still be quite complex.

Even more so, one can choose a combination of both methods. In the specific case of SQL, the same intermediary metamodel can be used for different SQL dialects, or one can choose to create specific metamodels for each SQL dialect. During implementation, it is necessary to find a balance between usability, reuse of transformations and effort of creating metamodels and metamodel-to-text transformations.

An overview of how these transformations are related to each other is shown in figure 6.1. When the user has the dataset model and function model defined, all steps in this overview can be automated using model transformations. The SQL model is derived from the function model using a pre-defined transformation.

SQL needs a specific execution platform. We deem SQLite a good choice for this, based on its ease of usage, level of usage within the data community

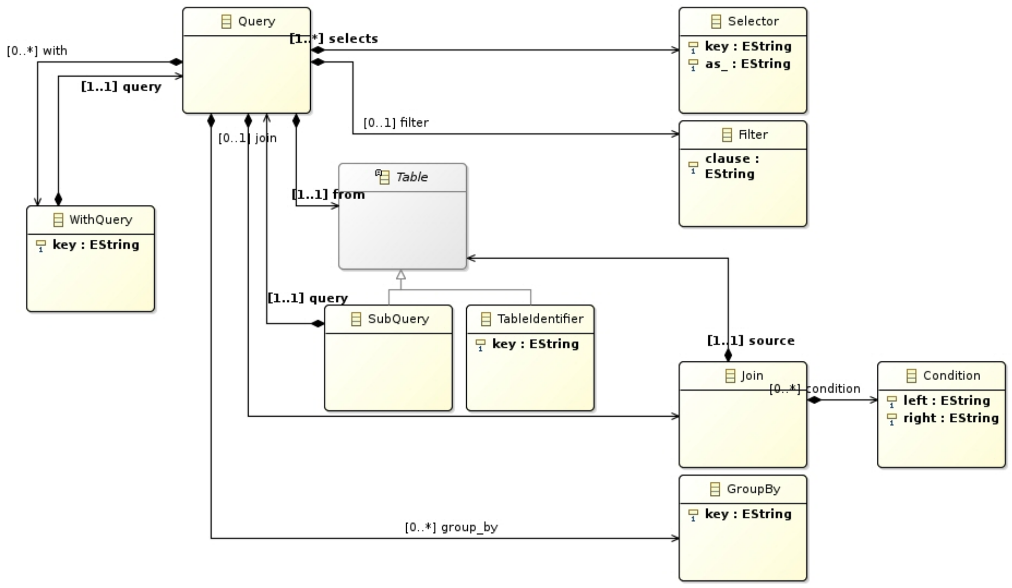


Figure 6.2: The simplified SQL metamodel suitable for code generation off of functions defined in terms of the function metamodel

and complexity of language. Since it is based on the SQL standard, it has a good foundation of data operations and because it stores data in a single file, it allows us to neglect many implementation and setup issues arising when using more complex database applications. Furthermore, while the SQLite dialect might differ slightly from other SQL implementations, the effort put into this transformation should be easily translatable to other SQL-based platforms.

6.3 Function transformation

This function transformation process requires the definition of a SQL metamodel. Its only use is to generate code based off of functions defined in our metamodel. Hence support for a subset of SQL is sufficient.

This metamodel definition is shown in figure 6.2. Because its only use-case is code generation, many simplifications can be implemented. For example, the *Selector* element is simplified to be just a string. If SQL functions are needed, this function can be incorporated into this string. While this poses problems when targeting different SQL dialects and is not an ideal solution, it allows us to show the concepts in the prototype implementation.

With the use of this new metamodel, transformation rules are defined that transform a function model into an SQL query model. The function metamodel from chapter 5, designed in such that it can represent a pipeline in which each step is executed consecutively. It also described the impact on the data for each of these operations, which are separately easily transferable to SQL operations. This means our transformation needs to handle concrete SQL syntax based off of these definitions, the initial data selection and a method of consecutive execution of these operations.

The concrete SQL syntax definition is the simplest step. Each operation can be directly converted to an individual SQL query, based on the definitions in section 5.3.1.

For data selection, the assumption is made that the SQLite database used contains a table with the actual data. The name of this table matches the key of the dataset. Every column in such a table corresponds to the keys as specified in the dataset itself. Importing a CSV file in SQLite is easily done using the `.import` command.

There are two methods to link subqueries using SQL. A *with* clause allows users to specify queries separately and reference them using an alias. The transformation can be created such that every operation from the function model is a separate SQL query and references the result of the previous function using an alias. This makes the transformation straightforward, as well as the resulting SQL query organised and understandable.

Another method is to use nested subqueries. This decreases the need of aliasing and referencing, but at the cost of the resulting query having a level of nesting for every operation and thus becoming complex to read for elaborate functions. In the end, the results are very similar. We opted for the first, as it is closer to the function metamodel and generates more readable SQL code.

Using these elements, we define the transformation structure as follows:

- Always start with a full selection of the source, because we need this data to start with, e.g. “SELECT * FROM dataset”. This query needs to be wrapped in an alias and used in the with query, which results in “WITH _1 as (SELECT * FROM dataset)”
- Create a *with* subquery that performs a query using the previous query, e.g. “SELECT key1, key2 FROM _1”. To keep the aliases simple, we use the combination of an underscore and a counter.

- After the last operation, create the main sql query that is a selection. While we could just use the last operation as the main query, that generates edge cases in the transformation that we have to solve. This method works generally, at the cost of some verbosity in the resulting query.

There is one additional complexity, which is handling the subexpressions of the initial function model. Because SQL does not support nested WITH expressions, the tree structure needs to be flattened. In order to do this, the order for execution for sub-expressions must be resolved. When sub-expression A depends on sub-expression B, B needs to be transformed before A.

6.3.1 Transformation example

As an example, we will investigate the issue of counting the amount of places for childcare services per neighborhood, such that these can be compared with statistical information from these neighborhoods. The query needed for this operation is shown in listing 6.1.

Listing 6.1: A query that calculates the amount of childcare services present per neighborhood, by first merging the neighborhood information from an external dataset followed by an aggregation operation

```
select lrkp
restrict metrics on keys actuele_naam_oko, opvanglocatie_postcode, opvanglocatie_huisnummer
add from adres_codes
    gwb2016_code
    match opvanglocatie_postcode=pc6, opvanglocatie_huisnummer=huisnummer
aggregation
    based on keys gwb2016_code, type_oko
    sum aantal_kindplaatsen as "Totaal aantal kindplaatsen"
    count ID as "Aantal kinderopvanglocaties"
```

This function consists of three different operations (restrict, add from and aggregation). For each of the operations, the impact on the data is known. Hence, conversion to SQL is as simple as translating each of these steps and merging them into a single SQL query, such that each of these steps is executed consecutively.

Listing 6.2: Resulting SQL query, generated based off the transformation from the query in listing 6.1

```
 1  WITH
 2  _1 as (
 3      SELECT * FROM lrkp
 4  ),
 5  _2 as (
 6      SELECT actuele_naam_oko, opvanglocatie_postcode,
 7             opvanglocatie_huisnummer, aantal_kindplaatsen,
 8             type_oko, ID
 9      FROM _1
10  ),
11  _3 as (
12      SELECT adres_codes.gwb2016_code, _2.*
13      FROM _2
14      JOIN adres_codes
15        ON _2.opvanglocatie_postcode = adres_codes.pc6
16           and _2.opvanglocatie_huisnummer = adres_codes.huisnummer
17  ),
18  _4 as (
19      SELECT gwb2016_code, type_oko,
20         SUM(aantal_kindplaatsen) as totaal_aantal_kindplaatsen,
21         COUNT(ID) as aantal_kinderopvanglocaties
22      FROM _3
23      GROUP BY gwb2016_code,type_oko
24  )
25  SELECT * FROM _4
```

It can be directly seen that every operation generates to its own sub-query. The `restrict metrics` operation (line 2) boils down to a selection of columns. The resulting SQL sub-query (line 5-10) selects these columns, as well as the dimension columns that are present at that location. Without adding these dimension columns, the dimensional information would be lost in the resulting data.

Similarly, the operation on lines 3-5 converts to the SQL sub-query on lines 11-17 and the operation with lines 6-9 results in the SQL sub-query on lines 18-24. Each consecutive function references the key of the preceding sub-query, just like each operation acts as on the result of the previous operation.

The last line of the query is a selection on the result of the last operation, which is a method to generalize the transformation for each operation. It does not require a special case to convert the subquery of the last operation to the main expression, instead of a subexpression.

Executing this sql query on a database including the lrkp dataset as table, yields the results as specified by the function in listing 6.1.

6.4 Dataset transformations

Parallel to the data transformations, dataset transformations are to be executed as well, such that a new dataset is generated matching the data result. Roughly the same approach is taken as during the data transformations. Start with the initial dataset as source, perform each operation sequentially and return the final result.

Each operation has a certain impact according to the specifications in section 5.3.1. The impact is that dimensions and metrics need to be adjusted, annotated or sometimes newly generated.

Because the structures of transformation are so similar, the challenges coming are mostly identical.

There is one additional challenge. The data source does not have to be retrieved, as an assumption is made for location of data and the tablename can be used easily. To be able to apply the operations sequentially, it is required to keep track of the state of the transformation. This due to the fact that

1. The transformation start with the root of the query, referencing a source. Instead of modifying a dataset, retrieval of an existing dataset is needed.

2. There can be multiple expressions in a query. The datasets resulting out of these queries need to be stored, such that other queries can reference them.

3. When using an operation that merges different datasets, the desired added source can be the result of a subexpression.

We solve these issues using an internal dataset registry that is accessible to the transformer implementation. This registry holds all datasets available for use during the transformation and the transformer can request these datasets when necessary. For each subexpression, these resulting datasets can be stored in this registry as well, allowing later expressions to request the registry for intermediary results.

Many metadata elements are just descriptions. To propagate context, the descriptions and names must be propagated accordingly. There is a design decision here on what to append, how to append it and on what level this should be described by the user.

We opted to generate additional text alongside the existing descriptions. This can mean that when aggregating a metric, its description can be prepended with the method of aggregation and leave the other information intact. Because the open data sources used and their documentation is in Dutch, the implementation adds textual information in Dutch as well. Bilingual support, or configuration of how to append information would be relatively straightforward but is considered out of scope for this project.

Chapter 7

Implementation details

For the interested reader, this chapter provides technical details on the implementation of the prototype. It can aid the reader in reproducing the implementation or providing new insights for model-driven engineering but can be skipped in its entirety without loss of conceptual meaning. Additionally, it provides a compressed overview of libraries and our reasoning of choice between different libraries of implementation.

7.1 EMF and PyEcore

The Eclipse Modeling Framework is the most widely used framework to apply model-driven engineering, and provides tightly integrated tools to enable development of models and transformations.

While the Eclipse Modeling Framework provides a solid foundation for models, its use bounds the user to the use of Eclipse and Java. Eclipse and Java are difficult to configure and make it hard to automate different steps, especially when one is not familiar with build tools used. The MWE2 workflow provides methods to chain different operations across the platform together, but we did not manage to make this work consistently. We suspect this is due to either configuration issues, versioning issues or untraceable errors after a model change.

Since this did not provide the right environment for our prototype implementation, we searched for a different method that would allow Model-Driven Engineering similar to the Eclipse Modeling Framework, but use better, more flexible, tooling that allowed us to develop a prototype more quickly.

PyEcore [10] is an open-source implementation of the Eclipse Modeling Framework in Python, and can be found on GitHub. It allows users to import existing ECore files, generate python code for the metamodels and use these classes. Because of our familiarity with many python packages, this allows us to use these packages to streamline the Model-Driven Engineering and use these libraries for parsing, text generation and automation of different steps. Documentation for PyEcore can be found at `https://pyecore.readthedocs.io` [9].

The ECore models are designed using Eclipse, since Eclipse its editor is solid and provides a good workflow. Importing these models for use in PyEcore is very simple. PyEcore supplies the tool `pyecoregen` that takes an ECore file and generates python model code. Executing `pyecoregen -e model.ecore -o .` in the terminal generates a python package called "model" in the current directory containing model code. This model code can be imported into the python code by simply using `import model`.

Python is a dynamically typed language, which might not be the ideal choice for Model-Driven Engineering. Using models implies that types of variables are static. PyEcore takes care of this by performing validation of types when a value is bound to the property of a model.

The dynamic nature of Python does give us advantages for the development of a prototype. During testing, a model can be loaded into an interpreter, allowing direct interaction and exploration of the properties of the model. Furthermore, Python is a famous scripting language to quickly tie elements of code together. This allows us to quickly combine different elements and automate execution of different steps of the framework using a complete programming language.

7.2 Text-to-model transformations

Using the metamodels with PyEcore allows us to create an python object representing a model as a python data structure. Because PyEcore can load the serialized formats generated by Eclipse, one could use the Eclipse editor to create models and load these into the application. This ECore format is however not easily readable, and creating these models in Eclipse is cumbersome. This can be solved by defining a concrete DSL syntax that allows the user to specify its model in a text file. Models are generated by parsing this text file according to the DSL specification.

Eclipse provides XText to specify these DSLs, but this is not directly available in Python. Python does have parsing libraries that can be used to parse text. TextX is a python parser that comes the closest to XText [1]. It

[1] https://github.com/igordejanovic/textX

sells itself as being a “meta-language for building Domain-Specific Languages (DSLs) in Python. It is inspired by XText”. Unfortunately, integration with PyEcore is still ongoing at the time of writing.

Therefore, we opted for a simple parser-combinator library called pyparsec [2]. Parser-combinators have their limitations, but these limitations are outside our scope. The ease of implementation of a parser-combinator library allows for a quick prototype and direct implemenation of the PyEcore models.

This parser takes the contents of a single file as input, and returns a Dataset model as an ECore model, as specified with the aid of PyEcore. A parser combinator parser a file and directly generates objects based on the elements it parsed. Listing 7.1 shows the root function to parse a dataset.

Listing 7.1: Root function to parse a dataset definition

```
1  @generate("Dataset model")
2  def dataset_model():
3      '''Parse a dataset model'''
4      yield keyword('Dataset') >> string('{') >> whitespace
5      d = DatasetMetaModel.Dataset()
6      d.key  = yield key_value('key', value_string)
7      d.name = yield key_value('name', value_string)
8      d.description = yield key_value('description', value_string)
9
10     for dim in (yield key_value('dimensions', sepBy(dimension, string(','))))):
11         d.dimensions.append(dim)
12     for met in (yield key_value('metrics', sepBy(metric, string(','))))):
13         d.metrics.append(met)
14
15     dist = yield key_value('distributions', csv_distribution)
16     d.distribution.append(dist)
17
18     yield string('}')
19     return d
```

Line 4 shows the keyword *Dataset* and the start of the bracket. Line 5 initiates a Dataset model using the PyEcore metamodel package. Lines 5-8 parse the key, name and description using a `key_value` function. This function is a parser-combinator in itself and parses a statement in the form *key* ‘:’ *value* and returns the value. Lines 16-22 provide parsing mechanisms to parse the other properties of the dataset and call other parse functions to retrieve partial elements of the model. The dimension and metrics parsers

[2] https://github.com/Dwarfartisan/pyparsec

are more complex than a simple key-value parse. The last two lines ensure that the bracket combination is closed and return the dataset itself.

Using this function requires some boilerplate code. It takes a string as input and returns a model, deliberately omitting functionality to read and write files. This is performed by a few lines of Python code. For reference, this code is provided in listing 7.2

Listing 7.2: Python boilerplate code to read a file and parse a dataset off of it

```python
def parse_dataset(filepath):
    with open(filepath, 'r') as f:
        text = f.read()
        return dataset_parse(text)
```

7.3 Model-to-Model transformations

When the text-to-model transformation have supplied the PyEcore dataset models, these models can be used in a transformation. Downside of our approach is that Python and PyEcore do not have transformation libraries themselves, so custom code is a necessity.

Still, we would like to use a similar approach used by Eclipse and its community. We could create a port of the QVT transformation framework written in Eclipse, but that is definitely outside the scope of this project.

Another tool often used for transformations in EMF is SiTra [12]. This is a simple library enabling the definition of a transformer and rules, and the transformer can be simply called to transform to the desired object. This framework is simple enough to easily port the implementation and its ideas to Python and our implementation.

The ported code for the transformer class, as well as the abstract Rule class are provided in listing 7.3 and 7.4. As Python does not have the notion of an abstract class, the `Rule` class is specified as a regular class. Its build method throws an error by default. Rules can override this method to omit these errors. When a Rule is not properly implemented, the runtime environment will throw an error, which is expected behaviour.

Listing 7.3: Transformer class from SiTra ported to python

```
class Transformer:

    def __init__(self, rules, context):
        self.rules = rules
        self.context = context

    def transform(self, obj):
        for x in self.rules:
            if x.check(obj):
                return x.build(obj, self)
        raise NotImplementedError("Could not find transformation rule for {}".format(obj))
```

Listing 7.4: Rule class from SiTra ported to Python

```
class Rule:
    source_type = None
    target_type = None

    def check(self, source):
        if type(self.source_type) != tuple and type(source) != self.source_type:
            return False

        # Check if all tuple types match, when applicable
        if isinstance(self.source_type, collections.Iterable):
            if not isinstance(source, collections.Iterable):
                return False
            for index, x in enumerate(self.source_type):
                if type(source[index]) != x:
                    return False

        return True

    def build(self, source, transformer: Transformer):
        raise NotImplementedError("The regular Rule class does
                not implement the build method")
```

An example of an implementation of such a rule is shown in listing 7.5. It sets the source type as property of the class, which is used by the `check` function to determine if the rule applies to the object that is to be transformed. The build function builds a SQL Query object from a Function object in the function metamodel. For each source object, such a rule is defined and the transform function on the transformer can be called to transform a sub element. The transformer selects the right transformation rule based on the source type set in the rule specification.

Listing 7.5

```
class FunctionRule(Rule):
    source_type = functionModel.Function

    def build(self, source: functionModel.Function, transformer: Transformer):
        sql = sqlModel.Query()
        subquery = transformer.transform(source.expression)
        sql.with_.extend(subquery.with_)

        sql.selects.append(sqlModel.Selector(key="*"))
        last_key = sql.with_[-1].key
        sql.from_=sqlModel.TableIdentifier(key=last_key)

        return sql
```

7.4 Model-to-text transformations

The metamodels and transformations have been designed to ease code generation. Since logic operations in Model-Driven Engineering should be performed in the transformations, code generation steps are easy to perform.

Because our code generation steps are so simple, the use of a templating language is deemed unnecessary. To generate the code, the built-in Python `print` functions are used to generate text. Functions are used to render sub elements to a string, which are propagated to the root function using return statements. While this highly limits the ease of code generation, as well as decreasing the readability of the code, it is sufficient for the prototype. For anything other than a prototype, we do not recommend using this approach, due to its limitations.

Listing 7.6 shows an example of how the with statements are rendered to text. These are the subquery elements of the function. On lines 4 and 7, it can be seen that other render functions are called that render subelements of the current object. Since these functions return strings as well, their combination can be easily returned.

Listing 7.6: Example of render functions to render different elements of the models to text

```
1   def render_withs(withs: EOrderedSet):
    if len(withs) == 0:
        return ""
    return "WITH " + ", ".join([render_with(x) for x in withs])

def render_with(with_ : sqlModel.WithQuery):
    return "{} as ({}\n)".format(with_.key, render_query(with_.query))
```

Similar to the parsing functionality, boilerplate code is needed to convert this properly to a file. Listing 7.7 shows this boilerplate code for the render function.

Listing 7.7: Boilerplate code to save the result of code generation to a file

```
def render_sql_file(sql_model, filepath):
    s = render_query(sql_model)
    with open(filepath, 'w+') as f:
        f.write(s)
```

Chapter 8

Validation

To validate the claims made in this work and show how a full implementation of our analysis approach looks like from a user perspective, we show the implementation of the analysis of the second case in terms of the framework. This allows us to verify the suitability of the framework for different analysis methods.

8.1 Case study

The case study used for validation has been described in Section 3.2. The analysis required in that case is to compare if the expectations of business sectors and locations from investment company OostNL matches reality. When OostNL targets an investment for a specific sector, assumptions are made on where to search for companies suitable for an investment. This may lead to skewed investment results, because other regions are neglected.

This analysis is performed by representing all datasets using the dataset DSL. During this process, the dataset model is analysed to validate if it is expressive enough to represent all information required.

The same process takes place for function definition. The function metamodel should be expressive enough to be able to define the analysis as required, and generate code able to execute this analysis.

8.2 Implementation

The implementation followed a 7-step process:

1. Identify all needed datasets. This is mostly shown in the case description, but we require a more detailed specification.
2. Represent each dataset in terms of a dataset model.
3. *Load the data for each dataset into the platform required for analysis.*
4. Define the analysis function
5. Generate the new dataset model by applying the analysis function
6. Generate *SQL code* using the defined transformations
7. Execute the generated SQL code and retrieve results

Step 3 and the execution of SQL code are required due to limitations of the framework prototype. Since only SQL is supported as execution platform, the raw data has to be loaded into an SQL table prior to execution.

8.2.1 Dataset identification

The first step is to identify the needed datasets, retrieve the data in a usable format and specify the datasets using dataset models. All data is stored on disk in CSV format. When necessary, we applied transformations on the data format to normalize these. These transformations include conversion of Excel spreadsheet format to CSV or changing delimiters in the CSV file.

Each dataset is identified using a unique key. The key is used as name of the data csv files, as well as the dataset model files, and use the appropriate extension to describe the type of file. Since all these files are stored in a single folder, this convention can be used to directly access files. The data files can be accessed by reading the file *<key>.csv* and the dataset model file can be accessed using the file *<key>.dataset*. These keys are identical to the ones used in the descriptions below.

projecten_capital This is a dataset provided by Oost NL and provides a list of investment at a specified time. Because the data contained in this dataset is confidential, data results presented in this report are randomly generated with the same format as this datafile. This file contains, among others, the name of the organisation, its address, province name, and short description per organisation.

lisa_per_corop Is a file originating from the Lisa register and provides per sector, per region code an indicator of the number of companies in that region, as well as the total number of employees of these companies.

To be able to properly use these data, a couple of conversions and mappings are needed to be able to convert the dimensions to the right format. We use the following datasets to be able to do just this. The following datasets can basically just be seen as definition files, but can be specified in the same dataset model format, to be able to use these datasets in the our framework.

sector_sbi_map This dataset describes a mapping from SBI code to a sector that Oost NL uses. This mapping goes deep unto 3 nested levels.

adres_codes This dataset has also been used for the lrkp case examples and holds information about neighborhood and municipality location of each postal code, number combination. We used this dataset to add municipality information to the companies provided in the dataset *projecten_capital*. This allows this dataset to be used including a geodimension.

gebieden_in_nederland This dataset has also been used for the lrkp case examples and holds information about neighborhood and municipality location of each postal code and house number combination.

8.2.2 Dataset model specification

For each of these files, we need to specify a dataset model using the described DSL. For completeness, we have included all dataset files in the following listings.

The first dataset *projecten_capital* is the data provided by Oost NL. It is structured such that each row represents an investment they made into the company, and has columns that represent information about the company itself. The file does not contain how much investment has been made, but rather information on the organisations themselves.

All metrics in this dataset are specified as properties.

Listing 8.1: Dataset representing the dataset with projects, originating from Oost NL

```
Dataset {
    key: projecten_capital
```

```
name: "Projecten van de Capital afdeling"
description: "Projecten van de capital afdeling gespecificeerd epr organisatie. Deze dat
dimensions: {
    type: identity
    key: id
    name: "ID"
    description: "Auto generated identity"
}
metrics: {
    type: Property
    key: Organisatienaam
    name: "Organisatie naam"
    description: "Naam van de organisatie"
},{
    type: Property
    key: "Sector organisatie"
    name: "Sector organisatie"
    description: "De sector van de organisatie zoals gedefinieerd door Oost nL"
},{
    type: Property
    key: "Bezoekadres organisatie"
    name: "Bezoekadres"
    description: "Het bezoekadres voor de organisatie"
},{
    type: Property
    key: "Postcode bezoekadres organisatie"
    name: "Postcode"
    description: "De postcode van het bezoekadres van de organisatie"
},{
    type: Property
    key: "Plaatsnaam bezoekadres organisatie"
    name: "Plaatsnaam"
    description: "Plaatsnaam waar het bezoekadres van de organisatie gevestigd is"
},{
    type: Property
    key: Provincie
    name: "Provincie"
    description: "Provincie waar de organisatie gevestigd is"
},{
    type: Property
    key: "Hoofdcontactpersoon Oost NL"
    name: "Hoofdcontactpersoon Oost NL"
    description: "De medewerker binnen Oost NL die als hoofdcontactpersoon is aangesteld
}
distributions: CSV {
    url: "file://./instances/case2/projecten_capital.dataset"
    }
}
```

The dataset from LISA is better structured and makes more extensive use of dimensions. Because it provides aggregated data, there is information about the region, year as well as their specified sector. This sector, however, does not correspond to the one Oost NL uses, as it is divided using the SBI classification.

Listing 8.2: Dataset representing the exported file from the Lisa register containing information per region about amount of companies and employees

```
Dataset {
    key: lisa_per_corop
    name: "Bedrijfsgegevens per sector, per corop regio"
    description: "Bedrijfsgegevens per sector"
    dimensions: {
        type: Spatial
        key: corop
        name: "Corop regio"
        description: "Corop regio"
    }, {
        type: Temporal
        key: Jaar
        name: "Jaar"
        description: "Jaar"
    }, {
        type: thematic
        key: Sector
        name: "Sector"
        description: "Hoogste level sector, volgens SBI codering"
    }
    metrics: {
        type: Measurement
        key: "vestigingen totaal"
        name: "Aantal vestigingen"
        description: "Het aantal bedrijven zoals berekend"
    }, {
        type: Measurement
        key: "banen totaal"
        name: "Totaal aantal banen"
        description: "Het aantal banen geregistreerd als totaal per sector"
    }
    distributions: CSV {
        url: "file://./data/case2/lisa_per_corop.csv"
    }
}
```

While the previous two datasets provide the data used for insights, other datasets are needed as reference for defining appropriate transformations. The following datasets represent definitions needed to extract information from the datasets above. When performing this analysis after further development of the framework and more elaborate data model loading, it can be assumed that these models are already predefined and thus do not provide additional effort for the user.

Listing 8.3: Dataset model of the data containing information of SBI codes, including their names and corresponding Oost NL sector. To be used to provide a mapping of SBI code to sector.

```
Dataset {
    key: sector_sbi_map
    name: "Sector SBI mapping code"
    description: "Mapping van sector naar SBI code"
    dimensions: {
        type: identity
        key: sbi_code
        name: "SBI code"
        description: "SBI code"
    }
    metrics: {
        type: Property
        key: sbi_naam
        name: "SBI code naam"
        description: "Naam van de SBI codering"
    }, {
        type: Property
        key: oostnl_sector
        name: "Oost NL sector"
        description: "Sector gedefinieerd door Oost NL"
    }
    distributions: CSV {
        url: "file://./data/case2/sector_sbi_map.csv"
    }
}
```

Since this dataset is provided by Oost NL, the assumption for other reference datasets does not hold. These kinds of custom transformations need to be loaded prior to its first use. Nevertheless, these references can be used more often, which brings big advantages when performing subsequent data analysis.

Listing 8.4: Information on postal codes and the municipality and neighborhood they belong to. Postal code and house number are the dimensions.

```
Dataset {
    key: adres_codes
    name: "Buurt en wijkcodes per adres"
    description: "Voor combinaties van postcode6 en huisnummer zijn de wijk en buurtcodes ge
    dimensions: {
        type: identity
        key: pc_zes
        name: "Postcode 6"
        description: "Postcode 6 gebieden zijn een standaard indeling in nederland"
    }
    metrics: {
        type: "Property"
        key: gwb2016_code
        name: "Buurtcode 2016"
        description: "Buurtcode zoals vastgesteld in 2016"
    },{
        type: "Property"
        key: wijkcode
        name: "Wijkcode"
        description: "Wijkcode"
    },{
        type: "Property"
        key: gemeentecode
        name: "Gemeentecode"
        description: "Gemeentecode"
    }
    distributions: CSV {
        url: "test"
    }
}
```

Listing 8.5: Dataset coupling municipalities to higher level regions. To be used to aggregate data on municipality region level to larger regions.

```
Dataset {
    key: gebieden_in_nederland
```

```
      name: "Gebieden in nederland"
      description: "Namen, codes en relaties van gemeenten in Nederland, ten opzichte van over
      dimensions: {
          type: spatial
          key: RegioS
          name: "Regio's"
          description: "Gemeenten in Nederland"
      }
      metrics: {
          type: Property
          key: corop_code
          name: "Corop code"
          description: "The corop code die bij een gemeente hoort"
      }
      distributions: CSV {
          url: "file://./data/case2/gebieden_in_nederland.csv"
      }
  }
```

8.2.3 Data loading

Since the prototype only supports SQL, data need to be loaded from the CSV files into an SQL database. In line with the earlier examples, SQLite has been used as execution platform.

SQLite is able to load CSV files using the built-in `.import` command. All datasets can be imported by executing all import commands consecutively, like shown in listing 8.6. The first argument specifies the csv file (relative to the directory sqlite is run in) and the second argument represents the table the data is loaded in. To facilitate traceability, the table name is identical to the key of the dataset.

Listing 8.6: Code to be executed using SQLite to load CSV files into the database

```
.import projecten_capital.csv projecten_capital
.import lisa_per_corop.csv lisa_per_corop
.import sector_sbi_map.csv sector_sbi_map
.import adres_codes.csv adres_codes
.import gebieden_in_nederland.csv gebieden_in_nederland
```

For this process to work properly, it is essential that all datasets have the same CSV format, i.e. the same separator. The separator using SQLite can be defined using the `.separator` command, which is to be executed before the import command to be able to load the file using this specific separator.

In a fully developed framework, these operations should either be automatically generated, or not necessary, because functions are executed on the CSV files. We envision that a dataset can have multiple sources, representing its data and that database storage can be automatically derived from the data and dataset file itself. Since these import operations are not exclusive to SQLite and are available for other data platforms as well, this concept is relatively simple to implement in other storage mechanisms.

8.2.4 Data retrieval

Steps 4-7 have been executed in an iterative manner, similar to how such a process is executed in real-life when insight is required based on data analysis. We start with a simple analysis and show its implementation. Based on that result, more complex analyses are implemented. A new function is defined and new insights are gathered, which lead to the execution of a new query. Its results are investigated new function model based off of the results can be created.

As an example for a simple analysis, we will investigate which sectors account for the most investments. Unfortunately, no numbers are present on the actual investment numbers, but we can perform an analysis on the amount of companies in each sector.

Listing 8.7: A simple function that generates an aggregation over the organisation sector based off of the Oost NL dataset

```
select projecten_capital
aggregate over metric "Sector organisatie"
    count ID as "Aantal organisaties"
```

Based on our definitions, we expect to retrieve a dataset that has the *Sector organisatie* as dimension and a metric called *Aantal organisaties*. The resulting dataset using the function-dataset transformation is shown in listing 8.8.

Listing 8.8: Resulting dataset off of the function-dataset transformation, based off of the function model from listing 8.7

```
Dataset {
key: "projecten_capital"
name: "Projecten van de Capital afdeling"
description: "Projecten van de capital afdeling gespecificeerd epr organisatie. Deze data is
dimensions: {
type: "Thematic"
key: "Sector organisatie"
name: None
description: None
}

metrics:
distribution: CSV { url: file://./instances/case2/projecten_capital.dataset}
}
```

The function-sql transformation generates SQL code based off of the function. This resulting SQL code is presented in listing 8.9.

Listing 8.9: Resulting SQL query off of the function-dataset transformation, based off of the function model from listing 8.7

```
WITH _1 as (
SELECT * FROM projecten_capital


), _2 as (
SELECT `Sector organisatie`, COUNT(ID) as aantal_organisaties FROM _1

GROUP BY `Sector organisatie`
)
SELECT * FROM _2
```

Executing this SQL code using sqlite yields results as presented in table 8.1. It can be seen that all columns in the table are represented in the dataset model. The dataset model also provides additional information on its resource.

Now, for a more interesting example let us count the amount of organisations per sector, per region. As a target for the regional analysis, we will use the COROP regions. This is a standardized classification for regions in the

Netherlands and is defined as shown in figure 8.1. For Oost NL, this regional specification is useful because it is the right level of granularity and there is enough information present on this level. Another advantage is that counts on this level can be aggregated from municipality information, or aggregated further up to province regions.

Because the Oost NL dataset does not have direct information on the region it belongs to, this function is more complex. The region information must be extracted from the address information in the dataset, in cooperation with a dataset that provides regional information for each address in the Netherlands.

This query implementation directly leads us to an issue. To be able to match the dataset *adres_codes* means that we need to extract this number from the address field in the dataset. Because this involves string processing, it is hard to do and not completely reliable. It is needed to execute an extraction function that extracts this number and adds it to a new column.

While the data process is tough, it is conceptually easy to model in the framework. It is just an operation on a metric that creates a new column. Which operations to support is, however, a hard problem, because this is highly dependent on the target platform for execution. For example, we might define such a function using a regular expression operation that extracts a piece of the string using a regular expression. Our target platform SQLite, however, does not have support for this feature, which renders this platform useless for that purpose.

We note this issue and will discuss it further during the discussion at the end of this report. To continue this case study, we perform this step by hand using the python code listed in listing 8.10 and continue based off of that dataset. It also takes case of removing all spaces in the postal code field, which is a data quality issue in the dataset provided.

Listing 8.10

Table 8.1: An overview of the output of the query from listing 8.9

Sector organisatie	aantal_organisaties
AF	22
CTE	40
FiF	15
Gelderse Specials	12
HTSM	23
LS&H	71
Maak	101

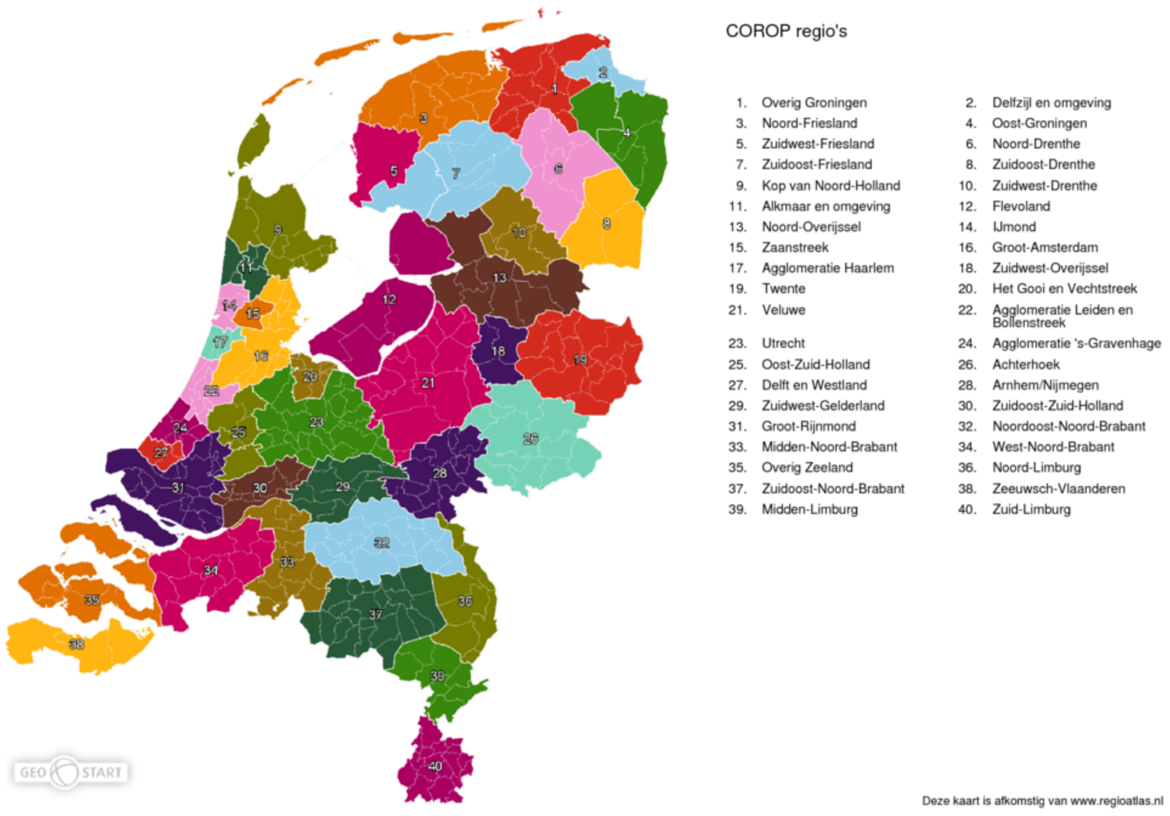


Figure 8.1: An overview of the COROP regions in the Netherlands. Image generated using www.regioatlas.nl

```
import pandas

def parse_huisnummer(x):
    gs = re.match(r'^([A-z]+\s)*([0-9]+)', x['Bezoekadres organisatie'])
    if gs is None or len(gs.groups()) < 2:
        return None
    return str(gs.group(2))

df = pandas.read_csv(csv_file)
df['postcode'] = df.apply(
    lambda x: x['Postcode bezoekadres organisatie'].replace(" ", ""), axis=1
    )
df['huisnummer'] = df.apply(parse_huisnummer, axis=1)
with open('out.csv', 'w+') as f:
    df.to_csv(f)
```

To aggregate to regional level, we first add the municipality code from the *adres_codes* dataset, followed by the addition of *corop_code* using this municipality code using the dataset with municipality information by the CBS. Using this corop code, we can aggregate using this corop code and sector of the organisation.

Listing 8.11: The query to aggregate the projects capital region to corop regions.

```
select projecten_capital
join properties gemeentecode
    from adres_codes
    match "Bezoekadres postcode"=pc6, huisnummer=huisnummer
join properties corop_code
    from gebieden_in_nederland
    match gemeentecode=gm_code
aggregate over metrics corop_code, "Sector organisatie"
    count ID as "Aantal organisaties"
```

The resulting dataset from the dataset should have two dimensions, containing the corop code and the sector of the organisation, and includes the count of organisations for each of those combinations.

The data is again retrieved using a generated SQL code based on this query specification. This SQL code is shown in listing 8.12. The results of running this query are shown in table 8.2

Listing 8.12: Resulting SQL query off of the function-dataset transformation, based off of the function model from listing 8.11

```
WITH _1 as (
SELECT * FROM projecten_capital


), _2 as (
SELECT `adres_codes.gemeentecode`, `_1.*` FROM _1
JOIN adres_codes ON _1.Bezoekadres postcode = adres_codes.pc6 and _1.huisnummer = adres_code

), _3 as (
SELECT `gebieden_in_nederland.corop_code`, `_2.*` FROM _2
JOIN gebieden_in_nederland ON _2.gemeentecode = gebieden_in_nederland.gm_code

), _4 as (
SELECT `corop_code`, `Sector organisatie`, COUNT(ID) as aantal_organisaties FROM _3

GROUP BY `corop_code`,`Sector organisatie`
)
SELECT * FROM _4
```

Table 8.2: An overview of the output of the query from listing 8.11

Sector organisatie	Corop code	aantal_organisaties
AF	CR01	18
AF	CR03	4
CTE	CR01	30
CTE	CR03	12
HTSM	CR01	2
...		

Based on this data, and the geographical information about the corop regions, a visualisation map can be created, or a graph representing the different elements. Such regional information provides insight where most companies are for each sector Oost NL invests in.

The last query that gives us insight in this case is a similar query, but executed on the data from the Lisa register, and combine it with the dataset from Oost NL. This requires several conversion steps of the LISA register, as well as the final combination of the Oost NL dataset when their dimensions match. The challenge here is the dimension transformations to match the dimensions in the dataset. The dimension transformations required are:

- Conversion from SBI classification to sector
- Conversion of Corop region name to code

These dimension transformations are difficult, because many exported files are targeted at providing a meaningful name, rather than a code in the dataset that can be used to be matched. Because this is the case, we need to convert the names in the LISA dataset to codes, or the codes in the original dataset to values. Usually using these names in a dataset brings data quality issues, because names to not fully correspond. This means there is no guarantee that the dimensions can be properly converted.

A dimension transformation consists of property join for the right region, followed by a aggregation over that property. If the mapping is proper, the resulting data is the same in size, though column with the named dimension is lost.

The query needed to perform these operations is listed in listing 8.13. In this query specification, two subqueries are used in order to pre-process the specification files that contain data about the regions and sector information. This pre-processing aligns their dimensions and allows us to match the values for the lisa dataset according to their code, rather than an possible ambiguous name.

Listing 8.13

```
with (
    select sector_sbi_map
    aggregate over metric sbi_naam
        identify oostnl_sector
) as sbi_naam_sector,
(
    select gebieden_in_nederland
    aggregate over metric corop_name
        identify corop_code
) as corop_names,
select lisa_per_corop
add properties oostnl_sector
    from sbi_naam_sector
add properties corop_code
    from corop_names
aggregate over metrics corop_code, oostnl_sector, year
    SUM "vestigingen totaal",
    SUM "banen totaal"
```

8.3 Results

During the implementation of this analysis, we can identify a couple of interesting elements that give us insight into usability and completeness of the framework. Since the prototype is not a complete implementation, issues and workarounds are expected. Our interest lies in the usability and validity of the models and transformations.

First of all, we deem the specification for dataset models sufficient to describe the data structures that we have seen. All content can be properly described, albeit in a simple manner. The dimensions that can be specified do give insight in how the data is structured and the context it is put in. Especially the concept of dimensions and metrics, and uniqueness and context they describe is useful.

This model however, does miss some functionality for dataset descriptions. For further implementation, adding elements from the DCAT model can improve the metadata description and enables more information to be stored about its original source, author and properties. While it is possible to store this information in the description of the dataset, using this method defeats the purpose of creating a specific model for a dataset metadata description.

Another element for the dataset model that has not been extensively considered is alternative dimension representations. Up until now, we have considered these to be different dimensions. Although this is valid and the conversion steps for the dimensions in the last query are useful, this operation is very common when using exported datasets. Applying this many times leads to a verbose query.

When we investigate the function metamodel, there are some additional issues. Most notably is the data quality issue that our prototype did not have an solution for. We had to use custom python code to modify the dataset in order to be able to use it. The operations and functions that can be executed on a single metric are too limited in the framework to be able to provide a solution to these data quality issues. This is problematic because resolving data quality issues is one of the most time consuming tasks.

Implementation of these functions is not straightforward, because different execution platforms need to be taken into account. When designing such a library in the Model-Driven Engineering context, different execution platforms and their standard libraries need to be taken into account. For example, SQLite does not support regular expressions out-of-the-box. This means that a solution needs to be found when such a function is specified in the function metamodel for this platform.

8.4 Conclusions

Overall, we deem the structure of the framework to be usable for these kind of case studies. The model structure works well to describe the data, the function metamodel is expressive enough to specify merges of data and metric calculations and the transformations provide usable SQL code for data analysis. Most importantly, the resulting dataset model can be exported back to valid DSL code and describes the resulting data in the way that is required.

The queries specified in the function metamodel are not necessarily more elegant or simpler than the resulting SQL queries, but the underlying modeling techniques allow for better tooling, validation of query and propagation of metadata. In its current implementation, a lot of this functionality is based on conventions. While this is not always desirable, it provides a foundation to be used further.

Furthermore, the prototype implementation lacks expressiveness in some areas. On the one hand, this is due to implementation issues in the prototype, such as only support for SQLite and retrieving dataset by filename convention. Other issues are more inherent to the models themselves. These include lack of operations for metrics and data quality issues, metadata modeling of the data itself, typing of data and dimensional representations.

We deem the structure of transformation most important and have shown potential for this structure to be well applicable to a wide variety of datasets.

Chapter 9

Conclusion

Throughout this report, we have presented our research on data metamodeling and platforms to be used for data analysis. Based on this research, we have implemented a Model-Driven framework prototype used for data analysis. Data analysis using this framework allows for propagation of metadata, automatic documentation of context and traceability of queries and data sources. Additionally, the development of the prototype and modeling concepts provide opportunities and improvements that have not been fully explored, which are documented in chapter 10.

9.1 Research questions

In the introduction, we posed the following research questions:

RQ 1. What elements are necessary to create a metamodel able to represent existing datasets?

RQ 2. How can we create models for existing datasets efficiently?

RQ 3. What is the best method to define a generalized query in terms of this data model?

RQ 4. How can the generalized queries be transformed to executable code able to retrieve data?

RQ 5. How can the context of the data be represented and propagated in the result?

Based on the related work and lessons learned during the implementation of the framework, we will present our answers to each of these questions.

What elements are necessary to create a metamodel able to represent existing datasets?

Because our inspiration is based on open data, we have used those sources mainly as reference for this question. It becomes clear metadata is needed on the data package itself, including creation date, author, size, description. These help a data analyst reason about the data as a whole.

Using only this information makes it hard for a user to effectively use the data. To solve this, metadata about the structure of the data is needed in addition. This structure needs to be modeled such that many 2-dimensional data structures should be able to be represented in this model.

How can we create models for existing datasets efficiently?

By defining a DSL, users are aided in specifying as much info about the dataset as possible. This DSL definition brings possibilities to develop tooling that aids the user in specifying such a file.

Model mining for existing datasets still remains an open problem, however.

What is the best method to define a generalized query in terms of this data model?

Generally, all queries are a representation of several data operations on a set of values. Our aim was to find a method to describe these operations into a metamodel. The two main options for this are to define a single expression like query that uses nested expressions to perform more complex operations, or to define an implementation in pipeline style. Because of readability and overview, we deem the latter the best option to define a generalized query.

To define a set of suitable operations, several other data transformation implementations have been investigated. Many of these data operations aim to achieve the same goal transformation, but are implemented in different manners. Based on this complete set, we have limited operations such that they 1) have actual meaning and 2) can be applied on the metadata too.

How can the generalized queries be transformed to executable code able to retrieve data?

The queries as specified above represent data operations to be executed. For this execution, a target platform is needed that. Because the current scope of these platforms is large, it is most logical to choose an existing platform for this execution.

We have shown that SQL can be a suitable execution platform of choice.

In a more complete implementation, multiple execution platforms can be supported.

How can the context of the data be represented and propagated in the result?

Context of data is represented in the dataset model. By allowing the query to transform these models, the context of the data is transformed as well and thus, preserved. While not all context can be preserved and some information is lost, at least the source of data and operations are preserved.

9.2 Prototype implementation

Based on these lessons, we have implemented a prototype for the framework and applied this to two use cases. We have shown that the framework is suitable for the business level analysis that is needed on open data sources to enable new insights.

Based on the cases we have presented, we deem the structure for the framework and its transformations a valid choice for performing data operations and maintaining the context. The functions defined are applicable to the metadata as well, and where needed require the user to explicitly describe his or her intentions. While this may hinder quick prototyping, this additional documentation will help in the long run.

We observe that many of the requirements posed on the metadata come from the process, rather than the model definition. While it is possible to define the provenance of data in an extensive and time-consuming manner, it is much more elegant to have these issues be covered by the process itself. After all, the goal of provenance is to describe the data analysis process and origin. By making the process inherently transparent, this documentation creates itself.

This is even more so the case with data quality. Data quality is usually defined using a set of metrics, supplied with the data. By using the specifications of data, types and units, many of these metrics can be derived. For example, timeliness as a quality metric can be derived if there is information about a temporal dimension. Ranges of values can be verified by using a proper definition of the number in the dataset model and empty values can be derived from the dataset itself as well. These features can be automatically analysed using the dataset model and, if required, stored in the model itself.

Although this may be true, the prototype implementation does not completely cover the full data analysis spectrum. Some shortcuts were used during the case implementation and the prototype needs to be extended on several areas to be able to overcome the necessity of these shortcuts and support more scenarios.

Yet, these issues are solvable within the current structure of the framework and will be further discussed in the next chapter. Some are straightforward to resolve, while others require effort to identify the proper method. For the latter type, we propose a direction for thought and some options.

Another side effect of this implementation has been the evaluation of ECore alternatives. We deem the PyEcore implementation a good alternative for programmers who like to take advantage of the flexibility of Python. This implementation allows the user to decouple the model specification from its environment and complicated configuration that comes with it.

The main disadvantage we encountered was defining a proper parser for the files. The parser library in Python must be combined with the models generated by PyEcore. Chapter 7 described our method of using a parser-combinator to achieve this result. While this works well, it is very time consuming and changes in the metamodel can lead to strange error messages. In hindsight, choosing another library might have been a better choice.

Chapter 10

Future work

Although we have shown the usefulness of Model-Driven Engineering for data analysis and metadata use, the scope of this project was too narrow to incorporate all desired features. A complete data analysis framework is difficult to accomplish due to the wide amount of formats available. The following sections present features that could be implemented to improve the framework.

To create a valid prototype within the scope of this project, some features have not been implemented. While these are not essential for a working prototype and tool on a small scale, to fully take advantage of the possibilities, an extension is needed to solidify the connections of this framework in the current data environment. These are not completely developed concepts, but rather documented ideas based on the framework as described.

10.1 Dataset model mining

In all steps we have shown, dataset models are defined by the user using the specified DSL. This task can be quite tedious, especially when the dataset and amount of columns is very large. One method to aid the user would be to create an graphical user interface that aids the user in defining dimensions, metrics and other properties. Because all properties all well-defined, such an interface would be straightforward to implement.

Even when using the interface, this task can be time intensive. One method is to implement model mining techniques to automatically create models based on the existing data. This means logic should be implemented that is able to load data sources, analyse them and create models accordingly.

This model mining can be based on other metadata models. For example, when using a dataset model from the Dutch government open data portal, it already has much information about the Author, originating source and distributions because it uses the DCAT metadata model. Transformations can be defined that create a dataset model from these metadata.

In case a dataset from Statistics Netherlands is used as a source, this mining process can be more elaborate. Its API exposes different information about the dataset, including keys of dimensions and descriptions of metrics. All these data can be extracted from this endpoint and converted into a dataset model.

The National Georegister is another portal providing an endpoint with data that can be loaded, mined and converted to a dataset model.

Even when the data is presented in a single CSV file, it is possible to parse headers that representing the columns, or parse the file metadata to extract modification date and author.

Based on the current premise, each of such an integration needs a custom implementation, which can be time intensive when there are many portals and different standards.

After this extraction process there is a dataset model. This model is, however, usually far from perfect because there can be many issues during the mining process. It is, therefore, necessary to allow users to change datasets. We suggest to define an additional transformation process that lets users just define property changes, and descriptions for a dataset. The final transformation is similar to the function-dataset transformation, but the underlying data do not change.

10.2 Dataset versioning

An issue often encountered using data analysis is that data sources change. Data that is hosted on external machines are not guaranteed to stay identical. Data owners might add data, or change column headers, etc.

One method to solve this would be to make the dataset models immutable. Current Model-Driven Engineering techniques do not have features to support this, but when creating a dataset registry, such features can be implemented. When adaptions are necessary, a new dataset model is created and its version is bumped.

Using this approach, it is also possible to validate the data sources and their structure. For example, dataset X with version 1.0.0 is created on the 1st of May, based on a SQL table. Users can use this dataset model to generate queries. Functions are defined that reference this dataset model, including version, and the transformations are used to retrieve results.

After a month, a user wants to change the dataset and inserts new information for the new month, using a new dimension attribute. The table has changed, and thus the dataset model needs to be changed as well. This change adds a new attribute to the dataset, and the new dataset becomes version 1.1.0.

Functions that are defined using the 1.0.0 dataset are still usable, because the 'new' table is backwards compatible with the old one. Functions based of off this dataset can still be properly executed.

Based on the dataset, we can define SQL queries of which the output can be calculated based on the dataset. For example, all distinct values in a column can be requested, or the different columns that should be present. This method allows us to validate if dataset models (and thus the functions based off of them) are still usable and if the right results are to be expected.

10.3 Data quality

Another issue we have barely touched is the data quality of datasets, while data analysis spend much time on these issues before they are able to use the data. Operations that are executed during the data cleanup are also relevant for the data provenance, to increase transparency. A more complete implementation of this framework should therefore also be able to handle these issues.

We notice that many of the data quality issues that arise are on the metric level. In the example case of Oost NL, we encountered the issue that some postal codes did have a space between numbers and letters, while others did not, e.g. the difference between '1234 AA' and '1234AA'. Because these data originate from user defined input, they are not consistent and for proper use, these values have to be made consistent.

While not much effort has been put into a proper set of function definitions applicable on a metric level, this is essential for these kind of operations. The language from trifacta is specialized in this use case and is therefore a good foundation for implementing functions that work in this level. Extension of the function meta-model is relatively straightforward and since these functions acts on the metric level they do not have much impact on the dataset model as a transformation.

Although conceptually easy for the framework, such an implementation does pose issues for an actual implementation and support for multiple execution platforms. During implementation of these functions, support for such a function on the execution platform is not guaranteed. This is further discussed in section 10.6 below.

Besides these issues, there can be more complex data quality issues. Databases can be inconsistent, have strange relationships, data may be in the wrong columns, etc.

10.4 Dataset annotations

The amount of descriptions and data that can be added to the dataset is still quite limited. To increase knowledge about the dataset and its context, we suggest a method able to define notes on different levels in the dataset.

Because of the way dimensions are defined, a selection of dimension values can be used to specify a data range unambiguously. During a data transformation, it is straightforward to determine if the data needed falls within such a range, since such a selection is based fully on the dimensions. Thus, we can use this method to annotate a certain part of the dataset.

We propose adding an additional *Note* element to the dataset metamodel. This note contains a dimension selection, metric selection (similar to the Restriction operation in the function metamodel) and a description. Extensions of this Note object can include multiple selections, a certain type (e.g. data quality, additional information or warning), or relations to other notes or datasets.

This selection method also maintains the requirement that the metadata can be propagated through the functions. Since it is known how the function alter the dimensions, these notes can be propagated through the transformations. For example, when an aggregation is made on a region that has many data quality issues, these data quality issues can be added to the aggregation itself.

This allows users to add information to the metadata of a dataset, which can be useful for users who will reuse those data.

10.5 Data typing

A more complicated addition would be to include a more extensive data type system into the model. In the current model definition, there is a distinction between some basic types, like an integer, string or date. These types are similar to the types generally seen in SQL databases. While this can work well generally, it is limited in functionality.

One enhancement can be a simple addition of different types. Types that are missing but could be useful are, for example, spatial data including coordinates or regional data, date ranges, or even some more complicated objects. Definition of custom types allows users to specify a better intention of the data value they provide.

Data analysis platforms do not benefit from extensive typing, like many programming languages do. If the metadata models could incorporate a more advanced type system, validation of these data operations can be more extensive. For example, if there is a type system that incorporates a “Percentage” type, we can base further analysis on this type alone. A warning could be provided if the percentage is above 100%, or if all values in the dataset are between 0 and 1, or if aggregations are made over this metric. Or there could be the definition a “Postal code” type, which contains certain semantics on a location, which can be used as base for aggregation functions.

In reality, the checks that could be defined per type can vary a lot. A percentage should be between 0 and 100, but there are exceptions, e.g. the current population is 103% compared to the previous year. Postal codes are also different in structure or might have other caveats. This is especially true in between countries.

Typing these elements allows the framework to be more specific on what operations are possible and meaningful. The current typing implementation can prohibit the addition of a number to a string, but cannot prohibit an addition of euros to percentages.

Extension of supported types is straightforward to implement, yet it is a challenge to define the right types that support a wide range of use cases, without bringing a lot of clutter to the framework.

An implementation of a more extensive type system is even more complicated. A typing system is complex and requires many elements that require extensive thought, especially in a model-driven environment. The typing system implemented must be flexible, because it is impossible to define all types beforehand, which requires extendable types. While this is possible to model in the dataset metamodel, this would limit reuse.

10.6 Multiple execution platforms

The prototype implementation we have shown only supports SQLite. The promise of Model-Driven Engineering is to be able to use the same models to target different execution platforms.

To implement this, a solution needs to be found for the different operations and features each of the platforms support. If the framework should be expressive enough to be able to handle different data quality and analysis aggregations, its support needs to be extensive. When considering open data platforms, it cannot be assumed that each of these platforms supports all features.

For example, the OData API platform that the CBS uses is mainly used for accessing data. It has some functionality for operations and expressions, but this functionality is limited. It cannot perform aggregation operations. it cannot do complex modifications on the dimension structures, etc. This means that functions cannot be easily applied to this OData API.

An obvious solution would be to create a database, load all raw data into this database and then perform the operations as defined. This does, however, require the need of a database and all complexity that is involved using that. It also is more complex for users to be able to execute.

Another solution is what we call partial query rendering. The result of each operation is a dataset. Therefore, a function model can be split up such that the results are two (or more) separate functions that each provide their own resulting dataset. When the result of function 1 is fed into function 2, a pipeline is created that provides the same result as the initial function. The advantage is that each function split can be transformed separately.

If functionality is not compatible across two platforms, these operations can be split up into two functions, rendered to their two platforms and executed separately. While this has downsides (most notably performance of exporting and importing datasets to the platform), it does provide functionality to target multiple platforms. If required, code generation can also be applied to automate the pipeline calculation process.

10.7 Function re-usability

Because one of the main promises of this framework is reuse of datasets and functions, function reuse is an important element as well. The method of implementation described in this report does not help much towards that goal.

To be able to properly reuse function models, we deem the following elements necessary:

- Function parameters should be definable and operations must be able to use these parameters.
- An operation must be added that executes a function and is able to define the parameters.
- A function registry, similar to the dataset registry is needed to be able to retrieve the functions to be implemented.

While each of these operations seems simple, their implementation can be quite complex. Implementation of these features generates a structure of nested functions that must be properly taken into account when transforming these to executable code.

Bibliography

[1] Dataset "vestigingen van bedrijven; bedrijfstak, regio". https://opendata.cbs.nl/#/CBS/nl/dataset/81578NED, .

[2] Sbi explanation, cbs. https://www.cbs.nl/nl-nl/onze-diensten/methoden/classificaties/activiteiten/sbi-2008-standaard-bedrijfsindeling-2008, .

[3] The fair data principles. https://www.force11.org/group/fairgroup/fairprinciples.

[4] Landelijk register kinderopvang. https://www.landelijkregisterkinderopvang.nl/pp/StartPagina.jsf.

[5] Lisa, employment register. https://www.lisa.nl/home.

[6] Open dataset landelijk register kinderopvang. https://data.overheid.nl/data/dataset/gegevens-kinderopvanglocaties-lrk.

[7] Dcc metadata standards. http://www.dcc.ac.uk/resources/metadata-standards/list.

[8] Prov-reccomendations. https://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/#Broad_Recommendations.

[9] Pyecore documentation: A pythonic implementation of the eclipse modeling framework. https://pyecore.readthedocs.io/en/latest/, .

[10] [github] pyecore: A pythonic implementation of the eclipse modeling framework. http://www.github.com/pyecore/pyecore, .

[11] Wet hergebruik van overheidsinformatie. http://wetten.overheid.nl/BWBR0036795/2016-10-01.

[12] David H Akehurst, Behzad Bordbar, Michael J Evans, W Gareth J Howells, and Klaus D McDonald-Maier. Sitra: Simple transformations

in java. In *International Conference on Model Driven Engineering Languages and Systems*, pages 351–364. Springer, 2006.

[13] Kamal Boulil, Sandro Bimonte, and Francois Pinet. Conceptual model for spatial data cubes: A uml profile and its automatic implementation. *Computer Standards & Interfaces*, 38:113–132, 2015.

[14] dr. A. Abela. Choosing a good chart. `http://extremepresentation.typepad.com/blog/2006/09/choosing_a_good.html`.

[15] John Erickson and Fadi Maali. Data catalog vocabulary (DCAT). W3C recommendation, W3C, January 2014. http://www.w3.org/TR/2014/REC-vocab-dcat-20140116/.

[16] Neil Foshay, Avinandan Mukherjee, and Andrew Taylor. Does data warehouse end-user metadata add value? *Communications of the ACM*, 50(11):70–77, 2007.

[17] Paul Groth and Luc Moreau. PROV-overview. W3C note, W3C, April 2013. http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.

[18] Jörn Kohlhammer, Tobias Ruppert, James Davey, Florian Mansmann, and Daniel Keim. Information visualisation and visual analytics for governance and policy modelling. 2010.

[19] Algemene rekenkamer. Trendrapport open data 2016. `http://www.rekenkamer.nl/Publicaties/Onderzoeksrapporten/Introducties/2016/03/Trendrapport_open_data_2016`.

[20] Jane Ritchie and Liz Spencer. Qualitative data analysis for applied policy research. *The qualitative researchers companion*, 573(2002):305–329, 2002.