FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Andrej Jurčo

Data Lineage Analysis for Qlik Sense

Department of Distributed and Dependable Systems

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

Prague 2020

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

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

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

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

Copyright (c) 2020 Andrej Jurčo
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software“), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sub-license, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED “AS IS“, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

In Prague date ....

Author’s signature

This way I would like to thank to my supervisor, doc. RNDr. Pavel Parízek, Ph.D., for his help and advice whenever I needed anything.
Thanks should also go to Ing. Petr Košvanec and RNDr. Lukáš Hermann for their valuable advice regarding Manta Flow platform and making sure I had all the information I needed to work on the assignment.
I'm also extremely grateful to my parents, relatives and friends for their support throughout all my studies and without whom it would be very difficult do get this far.

Title: Data Lineage Analysis for Qlik Sense

Author: Andrej Jurčo

Department: Department of Distributed and Dependable Systems

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

Abstract: Business Intelligence has become essential for all companies and organizations in the world over the past few years when it comes to decision-making and observing long-term trends. It often happens that Business Intelligence tools that are used become very complex over time and it can then be very difficult to make any changes. Data lineage solves this problem by visualizing data flows and showing relative dependencies. Manta Flow is the platform which creates such lineage which supports programming languages (Java, C), databases (Oracle, MS SQL) or Business Intelligence tools (Cognos, Qlik Sense).
The goal of this thesis was to implement a prototype of a scanner module for the Manta Flow platform, which would analyze data flows in Qlik Sense and create a data lineage graph from data sources to the presentation layer. This module extracts metadata necessary for the analysis, resolves the objects that are present in the Qlik Sense applications, and analyzes data flows in them. The resulting data lineage graph is then visualized by other components of the Manta Flow platform.

Keywords: data lineage, data flow, business intelligence, qlik sense

Contents

1 Introduction **2**
- 1.1 Goals . . . 3
- 1.2 Glossary . . . 4
- 1.3 Outline . . . 4

2 Technologies **5**
- 2.1 Business Intelligence . . . 5
- 2.2 Qlik Sense . . . 6
- 2.3 Data Lineage . . . 13
- 2.4 Manta Flow platform . . . 16

3 Analysis of Qlik Sense **20**
- 3.1 Metadata in Qlik Sense . . . 20
- 3.2 Objects in Qlik Sense . . . 24
- 3.3 Visualization Expressions . . . 38
- 3.4 Load Script . . . 45
- 3.5 Required Features of the Qlik Sense Scanner . . . 52

4 Implementation **54**
- 4.1 Technologies used . . . 54
- 4.2 Extractor . . . 55
- 4.3 Model . . . 58
- 4.4 Resolver . . . 59
- 4.5 File Reader . . . 64
- 4.6 Data Flow Generator . . . 64
- 4.7 Finishing Analysis . . . 77

5 Case Study and Evaluation **79**
- 5.1 Example application . . . 79
- 5.2 Extraction . . . 81
- 5.3 Analysis and visualization . . . 84
- 5.4 Space for improvement . . . 88

6 Conclusion **91**

List of Figures **94**

List of Tables **95**

A Attachments **96**
- A.1 User Documentation . . . 96
- A.2 Contents of the Attachment . . . 96

1. Introduction

Ever since the use of computer information systems has become a common practice in almost every sphere of business, companies have been trying to get as much useful information as possible from the data they managed to collect. It does not matter whether we are talking about sales statistics of a retail company, production efficiency data of a factory, or average truck utilization of a transport company. In each of these cases, employees responsible for making important business decisions use the data to make informed decisions.

This is when business intelligence (BI) comes handy and businesses around the world spend a lot of money annually to have all their data processed, analyzed and visualized, so that business-affecting decisions can be made correctly based on the real-world data.

There are many BI and visual analytics tools available on the market. BI can be done in a tool as common as Microsoft Excel, or special, dedicated tools can be used - Tableau, Microsoft SQL Server Reporting Services, Cognos or Qlik Sense. All these tools are more or less the same and the functionality usually does not differ in more than user interface provided and some unique aggregation functions.

When a company wants to start using one of these tools, it needs to, aside from providing the data for analysis, configure the tool to process the data in the desired way and to visualize results in an appropriate way. This means that sales performance by month is better to represent in a bar chart with each bar representing one month (ordered chronologically) rather than visualizing it in a single pie chart.

It is important to note that these reports or applications configured in BI tools have got two very important properties:

1. Tools are usually configured by developers and used by managers. These two groups of people are usually not the same.

2. These applications are often configured once and used for a long time without a change.

This means that, for example, it is very easy to develop a misunderstanding between what was desired by the manager and what was actually created by the developer. It can then result in making informed decisions based on wrong data and this could lead to huge losses or even bankruptcy. Another problem is that when somebody configured a tool 10 years ago and it needs to be changed now, there is a high chance that the employee does not work in the company anymore or does not remember what and how exactly was configured. This can be a problem when changes (or migration) has to be performed.

One way to solve these problems is by using data lineage. Data lineage can visualize the way data flows into, inside, and out of an application. In our case we could, for example, visualize data flows from databases or files into a BI tool we use, then analyze how the data is transformed inside the tool and in which visualizations or outputs is this data used.

This would allow a manager to see if a chart he or she is looking at is based on valid data, whether the data was prepared correctly according to his or her expectations. The programmer can also see if the configuration performed resulted in the desired result. If a tool has to be reconfigured after 10 years, data lineage can visualize data flows in the tool and help to quickly understand what happens in the tool and what will happen if we change one little thing in the configuration.

In a real-life use case this would mean that, for example, a company would sell computers and computer accessories online. They would like to visualize computer sales per salesman in order to award the top-selling employees. However, it could happen that the sum of computer sales of salesmen is way higher than what company account books say. A company would normally contract a consulting/audit company to see where the mistake is. This, however, takes months and costs a lot of money just to find one single mistake. Instead, a data lineage tool can be used to see what data is used and to backtrack problems from report to the database through all transformations. In the end, it turns out that the report formula included the sales for computer accessories as well, not just computers. This small mistake could cost the company a lot of time and money if an audit was performed, or it could be fixed quickly and in a money-efficient way if a data lineage tool was used.

Of course, data lineage is not a cure-it-all medicine for any BI solution there is, but in many cases, a brief look at data lineage of a tool can save hours of the tool analysis.

One of the major companies providing BI solutions is US-based company Qlik with its products Qlik View and Qlik Sense. They have over 50 000 customers such as Samsung, Cisco, Generali, or Deloitte [3].

On the other hand, one of the companies providing full data lineage is Manta Software based in Prague. Via its data lineage platform, MANTA Flow, it supports data lineage in around 30 technologies, mostly databases, data integration, and reporting tools [23].

1.1 Goals

The overall goal of this thesis project was to create a scanner module called *Qlik Sense scanner* that would be implemented into the Manta Flow platform. The scanner module must perform extraction of metadata needed for data lineage analysis from a Qlik Sense server. After that, it must perform a syntactic and semantic analysis of the extracted reports in order to determine data usage across the whole tool and to transform the result of the analysis into a graph that can be used in the Manta Flow platform.

The list of specific goals includes:

1. Get to know the Manta Flow platform and the Qlik Sense tool.

2. Design the scanner module for analysis of Qlik Sense in the Manta Flow platform. Use already existing infrastructure of the platform.

3. Implement a prototype of the scanner, document, and test it properly.

1.2 Glossary

Let us define some important terms that are often used in the whole text.

A *BI tool* is an application software which is used for Business Intelligence. It usually provides users with user interface and other tools to create reports - data visualizations placed in a layout.

A *report item* is a bar chart, a table, or any other visual element which can be placed in a report of a BI tool. It can also be an element that does not display any dynamic data, for example, a static text field.

A *data flow* is a relation between two objects where one object provides the data and the other object consumes the provided data.

Data lineage is a representation of data relations between objects. This means that if two objects work with the same data set in a firmly set order, there is a data relation between them. Data lineage, therefore, represents where the data comes from, how it is modified or used, and where its life cycle ends. This representation is usually visualized as a graph or a diagram.

A *scanner module* is a piece of software which can be implemented into the MANTA Flow platform and can extract, analyze and generate data flows of the technology it was developed for.

1.3 Outline

In Chapter 2, Qlik Sense, we first talk about business intelligence in general, then introduce Qlik Sense, describe its structure and functionality necessary to understand what kind of tool is about to be analyzed. We also describe data lineage and the Manta Flow platform into which the scanner module is implemented.

Chapter 3, Analysis, describes how Qlik Sense works, how it processes, stores and loads data, what can be done with the data and how Qlik Sense stores and provides the metadata about its structure. We further specify required features of the scanner module which are based on the analysis outcome.

Chapter 4, Implementation, describes how the scanner module was implemented based on the outcome of the analysis to meet the required features. It describes the implementation details of all three major components of the scanner module - the Extractor, the Resolver, and the Data Flow Generator.

The last chapter, Evaluation, demonstrates the functionality of the scanner module on a real-world example and discusses limitations of the prototype.

Because data lineage graphs and other visualizations used in figures of this work usually display many objects, their readability is greatly reduced when used in the text. Because of this, we included all figures in the attachment of this work, so that the reader can see and examine them in an appropriate resolution if it is desired.

2. Technologies

In this chapter, we will introduce Business Intelligence (BI), its purpose, common visualization layer and object structure and features. Then we will show how BI concepts are implemented in Qlik Sense, how the application works and what key features we are going to focus on. In the end, we talk about the Manta Flow platform in order to be able to understand what it takes to implement our scanner module into the platform.

2.1 Business Intelligence

Business Intelligence, in relation to this thesis, can be understood as a set of processes, tools and analyses used to process, aggregate and visualize business-related data used for making an informed decision. It helps people see data in a processed and visualized form so that the person that interprets the data can see clearly what the situation is like.

The key use case for all BI tools is monitoring company metrics based on collected data and make informed decisions based on these metrics in order to make the company more successful. Some of the common metrics can be quarterly sales performance, production efficiency data or sales margin information.

A more concrete example could be a transport company can see in their BI tool reports that fuel expenses take up 50% of their expenses altogether. Thanks to their BI tool they already found out what their problem was. Now that they know the problem, they can take precautions - they can, for example, teach their drivers how to drive fuel-efficiently or they can invest in newer trucks. To see how successful their effort is, they can compare the data over time in the BI tool. If the share of fuel expenses shrinks, their actions were successful, otherwise, they have to change their policies and look for other options.

Research performed for Qlik found out that 94% of business decision-makers think that using data helps them do their job better while only 24% of them feel fully confident in their ability to read, work with, analyze, and argue with data [9]. Moreover, the importance of BI is highlighted by the fact that the BI software industry itself was valued in 2020 at $14.7 billion [1].

Even though there are many BI tools on the market, their concepts and features are usually very similar, even though individual components are named differently. Most of the functionality is concerning data preparation and the visualization itself is usually only different in the list of different graphs and other visualizations provided.

Since we are concerned in data flows inside BI tools (more specifically, in Qlik Sense), let us describe typical data flows in BI tools:

1. **Fetching data from data sources**

   In this phase, the tool connects to data sources and fetches the data it needs to create a report. BI tools usually offer several options for data-loading.

   The most common is loading data directly from the user's databases, but there are also other options. Some tools provide loading data from files, such as Microsoft Excel sheets, CSV or XML files saved locally or in a cloud storage, while it is not rare to see an option to manually insert values.

Data values loaded from these data sources are usually unmodified, but most tools offer options to modify loaded data in this phase.

2. **Processing loaded data**

   After the data is fetched, the data is usually transformed, modified and customized to prepare it for insertion into the visualization. Among common operations are joins, unions or various data object creation.

3. **Visualization of data**

   Once all the data is prepared, it is forwarded into the reports and its visualizations which, based on the data provided, create graphs that the user can see and use.

Data lineage should ideally capture all data flows because they are all interconnected and there can be several inconsistencies and incomplete information if any of these data flow types are absent from the lineage. This can ultimately lead to an incorrect data lineage which usually has no value for the customer.

If we, for example, know our data sources and data processing actions, but our data lineage does not cover the visualization layer of the application, we usually have no idea how significant each loaded column or table is as we do not know if the column is used in every single graph of the visualization layer, or it is never actually used and it can be ignored.

2.2 Qlik Sense

Qlik Sense is a complete analytics platform that provides an intuitive user interface, data transformation options, many visualization objects and extensibility.

2.2.1 Qlik Sense products

Qlik offers several products in the Qlik Sense product family. Some products are stand-alone applications, other products are just implemented in these stand–alone applications.

A Qlik Sense distribution usually includes a server and a client. A server can run on a remotely accessible machine, for example at the company headquarters, or it can run on the computer of the user. It processes all client requests and stores all data created by its users. The server is usually accessed by users from the web browsers, just like common websites. The server processes all data and sends the output to the user who can work with the Qlik Sense without installing anything additional on his or her computer. Qlik Sense Desktop is distributed as a server and a client in a single Windows application, so the client can be in this case both the Qlik Sense Desktop application and the web browser.

Below are a list and a short description of various products offered by Qlik.

Qlik Sense Enterprise

Qlik Sense Enterprise is the full version of Qlik Sense, supporting a full spectrum of analytics use cases on a multi-cloud platform. Thanks to its containerized multi-cloud architecture, Qlik Sense Enterprise can be deployed on-premise, in a private cloud or in Qlik's hosted cloud [11].

Qlik Sense Business

Qlik Sense Business is a SaaS (software as a service) that combines Qlik's analytics platform with collaboration and sharing options. This means that an application can be shared with other users or groups.

Qlik Sense Desktop

Qlik Sense Desktop is the Windows version of Qlik Sense which can be run locally. When compared to Business or Enterprise versions, it provides less functionality. Since it is run locally, collaboration options are not available and applications created in this Qlik Sense Desktop can not be shared with other users. Security functionality is disabled because of absent sharing options.

Qlik Sense Mobile App

Qlik Sense Mobile App is an iOS application that allows a user to connect to a Qlik Sense server, download and view applications offline.

Qlik Analytics Platform

Qlik Analytics Platform is a platform for developing custom analytic applications using a wide variety of Qlik-authored APIs and SDKs. It extends Qlik Sense Enterprise and Qlik Sense Desktop with editors for developing extensions (extending visualizations with custom behavior/visual effects) or mashups (web pages that connect to a Qlik Sense server and visualize data from various sources).

2.2.2 Qlik Sense Object Structure

Even though there are several versions of Qlik Sense, they all implement the same concepts and form the same structure. The difference is mostly in what can be done with the object structure created. Some versions allow users to share it, some allow creating duplicate or publishing on the web.

Every Qlik Sense server, whether it is a cloud version (Qlik Sense Business/Enterprise) or a local version (Qlik Sense Desktop), consists of objects called Applications. These applications are individual units in Qlik Sense and there is no relation between any 2 of them. At most, they can load data from the same data source.

When a user creates an application, it is empty and there are several things a user can do. It can be connected to data sources, which can be various databases, file hosting services or files on the server's file system. Additionally, data can be input manually.

It is very common in Qlik Sense to perform data transformations when data source connections are created. These transformations usually include a table or a column renaming, table merging and splitting, creating aliases, modifying column values using maps or storing transformed data into a file. These commands, together with commands defining which columns and tables from the data sources shall be loaded, are stored in a *Load Script*.

Once the data loading and transformation commands are set up, a user can trigger data-loading action which performs the configured commands and saves resulting data locally on the server. This data is used for visualizations and other usages until data-loading action is triggered again. Qlik Sense does not refresh this data automatically. However, there are some extensions that automatically launch data-loading action.

After the data is loaded, it can be used to create visualizations. Only tables that are saved locally after data-loading and system-provided statistics tables can be used for the calculation of graphs and other visualizations.

The main component of data visualization is a visualization, in other BI tools usually referred to as a *Report Item*. Visualizations are grouped in sheets, which are usually focused on some part of data (sales, turnover or growth). This means that in a sheet named *Sales* we could find a bar chart of sales by quarter, a gauge measuring total sales in the past 12 months or a KPI (Key Performance Indicator) showing average and target profit per sale.

When we talk about developing a Qlik Sense application, it is usually the actions mentioned above that developers do - defining data loading and transformation commands and creating visualizations.

These visualizations are, however, dynamic, since they refresh with every new data-loading action. That is why Qlik Sense provides *Stories* functionality. The easiest way to understand what a story is is by comparing it to a Microsoft PowerPoint presentation. A story is a collection of slides with graphs, text and images that can be used for presentations.

A user can, for example, save some important visualizations at the end of a fiscal quarter for a presentation to his/her superiors without a need to use any external tools for persisting the graph image and creating a presentation about the quarterly results.

As we have shown above, the main structure of a Qlik Sense application is not very complicated and high-level data relations are quite obvious. As visualized in Figure 2.1, first there is connecting to data sources, from which data are loaded and transformed. The resulting data forms tables which are saved locally and it is used in visualizations and stories.

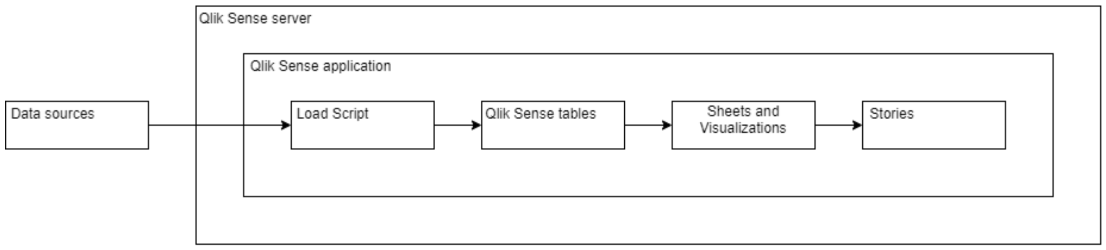


Figure 2.1: Visualization of data flows of a Qlik Sense application.

Now that we know what the basic structure of Qlik Sense is, it is a good time to explain what individual components are and what they do in more detail.

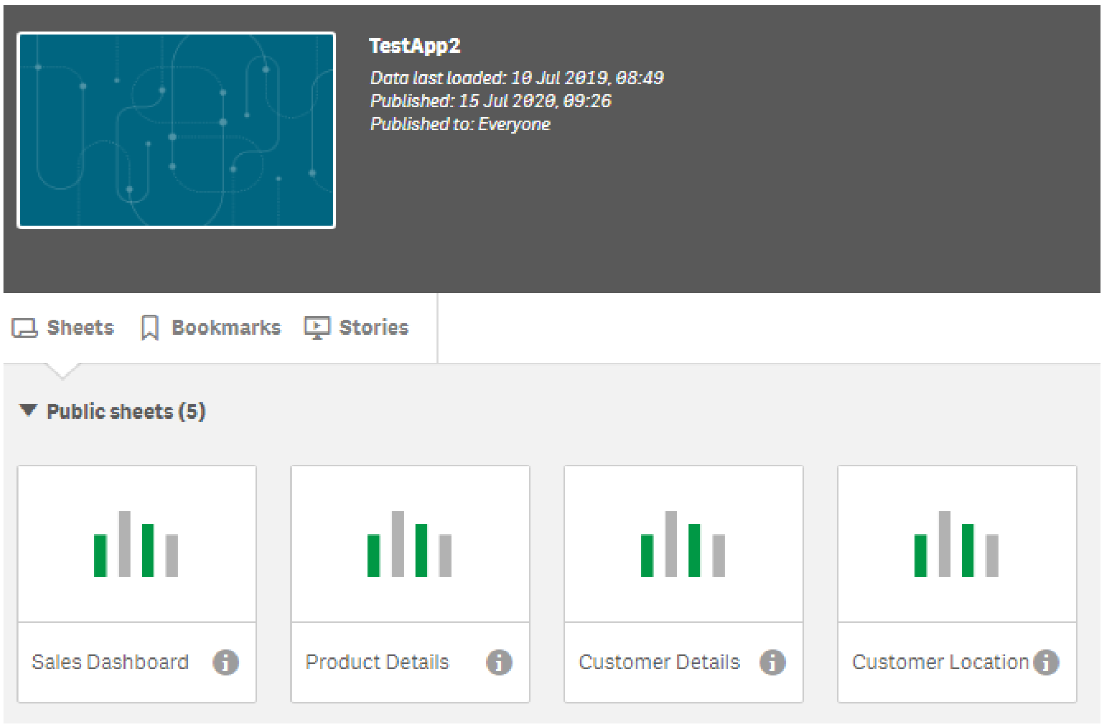
2.2.3 Applications

Application is an elementary building block of Qlik Sense. Apart from server settings, every action done in Qlik Sense is related to a single application. This means that an application is an independent entity in Qlik Sense.

All objects in Qlik Sense such as sheets, visualizations, stories or loaded data belong to a single application and once the application is deleted, all its objects and structure is deleted as well.

Applications are stored as QVF (Qlik View Format) files and are loaded by Qlik Sense when they are opened. Since applications are not server-dependent, it is possible to migrate applications between servers by copying the application's QVF file. However, when cloud versions of Qlik Sense are used, application access rights are related to the server and are not included in the QVF file.

Apart from a simple usage, where data loading, transformation and visualization is performed, there is one other common usage. When data sources provide very large data sets, it is sometimes convenient for performance reasons to split data preparation and visualizations into several apps. This means that data is loaded from data sources and transformed in one or more applications and resulting tables are stored in QVD (Qlik View Data), CSV or TXT files. Then visualization application loads these processed tables from the files and uses the data in visualizations. In Qlik Sense Enterprise it is also possible to split the work using on-demand apps. You can see a Qlik Sense Application user interface in Figure 2.2.


TestApp2

Data last loaded: 10 Jul 2019, 08:49
Published: 15 Jul 2020, 09:26
Published to: Everyone

Figure 2.2: An application overview user interface of the Qlik Sense Enterprise client.

2.2.4 Data Loading and Transformation

Data loading is an essential feature that each BI tool must provide. All steps that are needed to be performed in this part of data-processing are saved as *Load Script* which is a script that uses script language created by Qlik. We will talk more about the script language in Chapter 3.

Connections to data sources are done using Qlik connectors. There are many built-in connectors for REST data sources, Salesforce accounts, web files or common database engines, such as Oracle, PostgreSQL, IBM DB2 or Teradata. On top of this, data can be downloaded from Qlik DataMarket (Qlik Sense Enterprise) and there are additional connectors for download on Qlik's website.

Qlik Sense provides users with Data Manager, which is a graphical user interface that generates *Load Script* without any need for understanding the script language. However, only simple data transformations can be generated and complicated operations may need to be written directly.

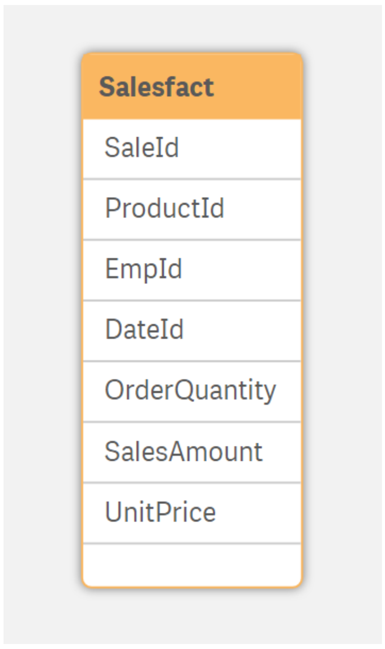


Figure 2.3: A script-loaded table visualized in Qlik Sense's *Data model viewer*.

A simple table-loading script statement can be seen below. This statement loads data from a database into a table named *Salesfact*. Qlik Sense then visualizes the table in its *Data model viewer* very simply, as illustrated in Figure 2.3.

```
[Salesfact]:
SELECT SaleId,
   ProductId,
   EmpId,
   DateId,
   OrderQuantity,
   SalesAmount,
   UnitPrice
FROM dbo.Salesfact;
```

2.2.5 Sheets

As mentioned earlier, sheets are objects belonging to individual applications that group visualizations. Each sheet has a layout, either fixed-sized or extended, where visualizations are placed. These sheets can be duplicated within an application, but they cannot be copied across applications. An example of how a Qlik Sense Sheet looks like can be seen in Figure 2.4.

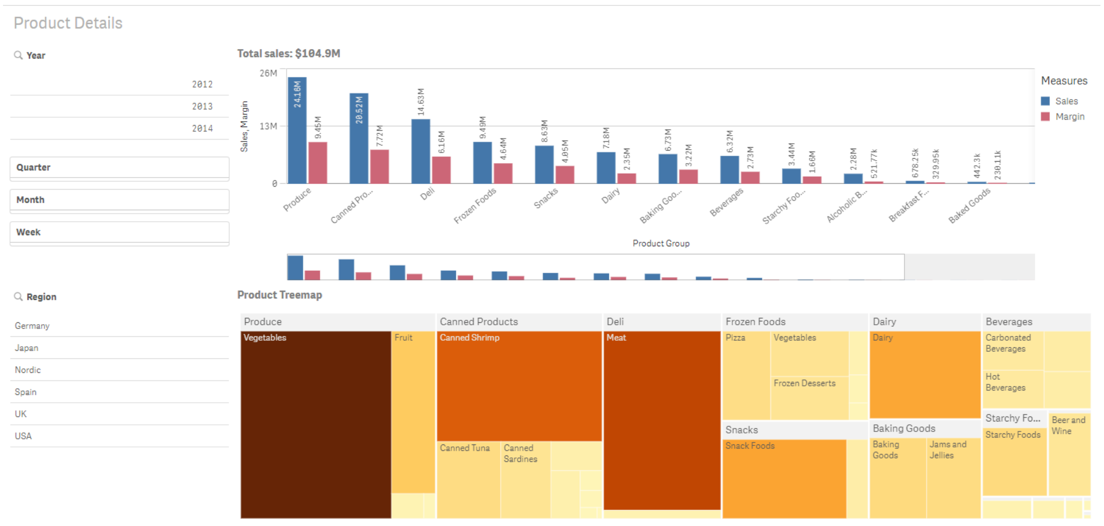


Figure 2.4: Visualization of a sample sheet in Qlik Sense.

2.2.6 Visualizations

Visualizations are objects which have the highest value for users. It is here where the actual analysis is shown and where they get all their BI tool value they need. Qlik Sense provides many builtin visualizations, for example, a bar chart, a pivot table or a distribution plot. It is also possible to create custom visualizations using the Extension or Widget editors which are available in Qlik Sense Desktop and Qlik Sense Enterprise.

Visualizations use Multidimensional Analysis, which means that all data that is provided for visualization is either a dimension or a measure.

A *Dimension* specifies the way data is supposed to be grouped. It is the easiest to imagine a dimension as an X-axis in a 2-dimensional graph. If we make a sales chart, we may want to group data by quarters or when we evaluate the production efficiency of factories, we may want to group production data by the city in which is the factory.

There is also a hierarchical dimension, sometimes called a drill-down dimension, which is a combination of related parameters. It is most commonly used in connection to time, such as relation year-quarter-month-week or location - town-province-country-continent. However, Qlik Sense allows users to define their own drill-down dimensions which are not limited to time or location only.

A *Measure* defines what shall be calculated for the dimension groups. Measures usually contain an expression that uses an aggregation function, such as sum, average, maximum or minimum. In a 2-dimensional graph, we could compare a measure to the Y-axis. If we have sales grouped by quarter (dimension), we could want to set a measure to the sum of sales to see sales statistics per quarter.

It is worth noting that each visualization uses a different amount of dimensions or measures. Some visualizations use only 1 dimension and no measures while other visualizations can use 2 dimensions and 3 measures. You can see an example of the Qlik Sense user interface for setting dimensions and measures of a visualization in Figure 2.5.

Qlik Sense offers an option to save a dimension, a measure or a visualization for reuse. This means that a user can define some dimensions, measures and visualizations once and use them at several locations without any need for repeated creating. If a shared item is modified, all its usages across all sheets and visualizations are modified automatically. This is particularly useful for cases when an object is used several times and modification of every single usage would be a very repetitive and possibly time-consuming activity.

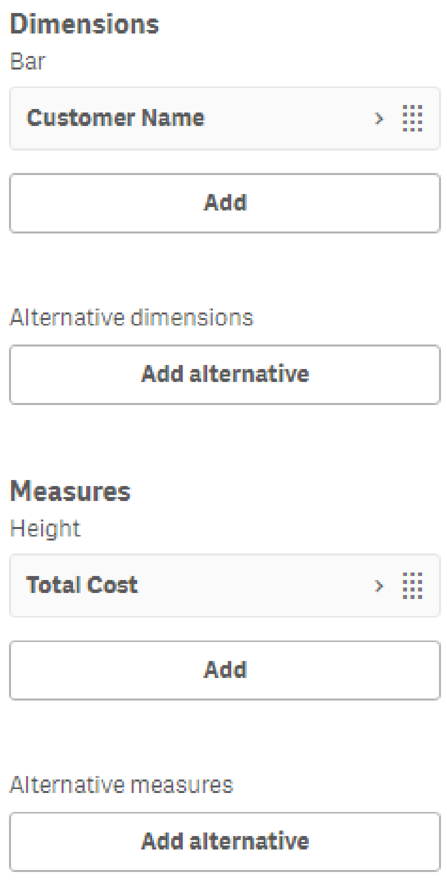


Figure 2.5: An example of how Qlik Sense user interface for setting visualization's dimensions and measures looks.

2.2.7 Stories

Storytelling is a feature intended for sharing of data discoveries, reporting and presentation. Each story consists of several slides that can be populated with slide items such as snapshots, images, text or live sheets.

Snapshots are commonly used for highlighting some interesting data findings, usually after applying some selection filters while providing the ability to jump directly to the sheet and visualization that is the source of the given snapshot. Live sheets add interactivity to the presentations which means that data could be updated every time a presentation is given without any need for modifying the story. Elements such as text, images, shapes and effects offer options for slide customization, however, they have nothing to do with data flows.

2.2.8 Other Objects

Except for the above-mentioned objects, which are the essence of Qlik Sense and most functionality this tool provides, there are two other commonly used objects - variables and bookmarks.

Variables are particularly useful in cases where dynamic data needs to be supplied. A user can create a variable during *Load Script* execution using one of its statements or they can be created using a graphical user interface in the *Analysis* interface of Qlik Sense.

Variables can be either assigned a static value or they can be set using an expression which is either evaluated at variable creation or it can be evaluated dynamically every time the variable is used while taking into account currently set filters and evaluating filtered data only. A common usage of variables is in the *Load Script* for-cycles when iterating over a list of tables is desired since a variable can contain a comma-separated list of strings. Moreover, if we wanted to include total sales amount in the title of a bar chart displaying sales statistics, this number could be added into the title string dynamically using a variable and string concatenation.

As mentioned earlier, Qlik Sense provides filtering options during analysis. Users can create their own filters by creating data selection from a list of values each column offers. This means that if we have a sheet of sales and we only want to see sales statistics for the top 5 salesmen, we can limit selection to data of these five salesmen and all visualizations in the application are recalculated with the data created by selection filtering.

Once these selections are created and the user wants to keep them, they can be saved as bookmarks. In other words, bookmarks are Qlik Sense's name for saved selections. These bookmarks are not related to any sheet, if they are used, their selections are applied application-wide.

2.2.9 Qlik Sense Use Case

Needed? There were small use cases across the whole Qlik Sense part to show what individual objects are used for.

2.3 Data Lineage

Data lineage provides information about the origin, movement, transformation and usage of information throughout its life cycle. It allows users to track information step-by-step and monitor how it was created, where it was used, what transformations it underwent and what other data was created based on it. Data lineage helps to answer some common questions that developers or data analysts usually ask - Where does this information come from? What happens if I modify this piece of code? How do I know that the data I am looking at is correct?

Data lineage is no new term in the world of data and there are many companies that provide data lineage. It can be done either manually, commonly done by audit and consultancy companies or it can be automated, where the key players are MANTA Software, Collibra and Informatica.

The role of data in everyday life and in the business sphere is very important in these days and it is crucial for each company to know where its data come from and what happens to it. Large companies usually spend large sums to consulting or audit companies to have their data lineage created or updated and despite the price they have to pay it often pays off. It goes therefore without saying that an expensive, but a well-informed decision is better than a cheap and misinformed decision. Data lineage is usually visualized in form of a graph and that is why it is not very hard to navigate in it even for less technical employees.

There is probably no company in the world that would not find a single use case of data lineage in their business. Health companies find it useful to see how data of their clients is used in databases, processed and used since health records are a very sensitive piece of data and privacy and security are required on a very high level. Database developers certainly appreciate impact analysis options and what-if analyses that come with data lineage. Thanks to it they can see what would happen if they changed some piece of code and what would be affected. Data quality can also be examined using data lineage which is a feature that many chief officers appreciate in companies across all business sectors.

Data lineage in BI tools such as Qlik Sense is usually used together with data lineage of databases, ETL[1] tools and other data sources to show how all the data that undergoes some transformations and various stages of life cycle is used in (usually) last phase of life cycle. It is indeed most often the BI tools that are at the end of the data life cycle since reporting tools almost never produce any new data, but they consume data directed to them from data sources.

It is often important to know where report data originates, what happens to it and how the final visualization was calculated in order to ensure that data quality is not flawed and the data that decisions are based on are valid. BI tools are usually complex and if any changes are made anywhere on the way from a data source to the BI tool, an impact analysis is necessary. Even a tiny change in the beginning of the data life cycle can cause some serious changes in reports that are reviewed by the decision-makers.

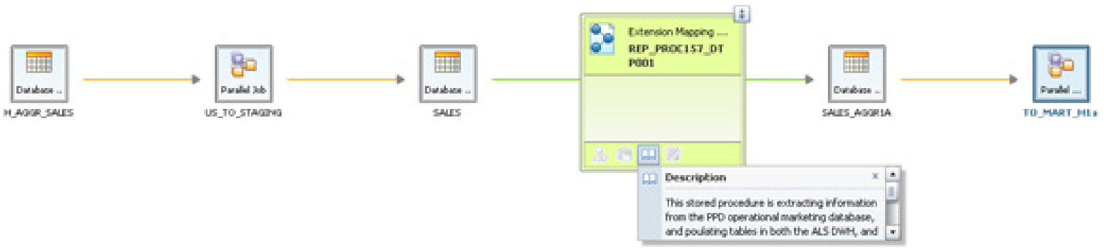
2.3.1 Analysis output

Data lineage can be performed and visualized in many different forms which result in similar output, however, some data lineage analyses can only use direct data flows and some analyses also analyze indirect flows. The difference is that a direct flow is between two data sets where the target data set contains data based *directly* on the source data set. An example of a direct data flow would be a data set of monthly sales which was created from sales data loaded from a database using an aggregation function of summation. Indirect flows, on the other hand, only affect the source data set. Indirect flows are usually filters, limitations or various conditions. A good example would be extending of the direct data flow example with condition that sales records are filtered to sales to customers based in Germany. Here, the country of origin of a customer is an indirect data flow as it does not provide any data to the target data source, however, it affects what sales records are used from the source data set.

[1]Extract, transform, load

Another important difference between data lineages provided by different companies is the form of visualization. A very common form of visualization is a directed graph that provides very good readability even for large graphs. It is no wonder then that almost all data lineage providers use graphs. Some graphs are interactive with zooming options, various filters and several detail levels, some are static graph images providing only high-level data lineage information without too much detail.

An example of a visualized data lineage graph in the MANTA Flow platform can be seen in Figure 2.6.



This stored procedure is extracting information from the PPD operational marketing database, and poulating tables in both the ALS DWH, and

Figure 2.6: An example of how a data lineage graph can look [6].

2.3.2 Use Case

A very good and creative use of data lineage was from an international bank which was using a workshop-type software to release internal production software. Because this internal database software was so complex, they had 10 to 15 patches of code each day between releases. To make sure that the environment did not crash, they had to deploy those patches of code all at once. However, each patch was developed by a different team and they did not know how other pieces of code were dependent on their code. This lead to frequent environment crash as these patches were applied in the wrong order.

However, after the company started to use a data lineage tool, MANTA Flow, they could see how each piece of code was related to other pieces and they could use MANTA API to generate correct the order of applying patches so that the environment did not crash and releases were on time. Compared to the old way of workflow in the bank, this was their new routine where only one person was needed to make sure patches are applied correctly:

1. Gather the patches
2. Upload them into MANTA
3. Use MANTA to determine the order of deployment
4. Order the scripts accordingly
5. Release them in that order [13].

This example shows how even complex and potentially unsolvable problems can be resolved fairly simply by using data lineage tools available on the market.

2.4 Manta Flow platform

MANTA Software is a Czech company developing an automated data lineage platform. It performs data lineage analysis using metadata retrieved from individual technologies and representing the lineage in form of an interactive graph. First scripts for automated data lineage analysis were written in 2005 [8] and since then the company, which started as a project of a Czech consultancy company, Profinit, became an independent company with its own platform supporting over 30 technologies.

The main advantages that MANTA and its platform provide in comparison to other competitors in the field are the number of technologies supported, detail of data lineage and performance. It is capable of analyzing very large applications and databases in great detail and creating a data lineage graph across several technologies used by the customer within a few minutes.

2.4.1 Supported BI tools

It has to be said that Qlik Sense is definitely not the first BI tool that became supported by the MANTA platform. Among already supported tools are Cognos, PowerBI, Tableau or OBIEE. It was, therefore, possible to reuse some concepts used in data lineage analysis of these tools and since once of the goals of this work was to use already existing structure provided by the MANTA platform, the architecture of our scanner module is similar on a high-level to scanner modules used for the analysis of the above-mentioned tools.

2.4.2 Visualization of data lineage in MANTA Flow

MANTA Flow uses graphs to visualize data lineage as well. In order to be able to visualize data lineage across several technologies in a single graph, they use a graph that is split into several columns where every two columns represent one operation on data - nodes in the left column are input data operation and nodes in the right column represent the end of the operation. This way it is quite simple to follow the data life cycle in the graph starting in the leftmost column and slowly iterating across columns to the right to see what is happening to it. However, it is important to bear in mind that this is just a general structure of graphs in MANTA Flow and since it covers various different technologies which can have various, often incompatible model structures, the actual graph structure varies to match these structures as accurately and comprehensibly as possible. An example of a data lineage visualization can be found in Figure 2.7.

Data lineage which is a result of data flow analysis usually includes both direct and indirect flows. The degree of detail often depends on the metadata that can be retrieved from the technology as not all technologies provide API that allows the analysis to be performed in maximum detail. Moreover, some metadata can be encrypted or modified in a way that no relevant information can be retrieved.

Once the graph is generated after successfully performing an analysis, it can be modified by the user to match his or her needs. The user interface allows user to specify which objects (databases, tables, report items or even Qlik Sense applications) shall be visualized in the graph and in addition to that MANTA

Flow provides the user with a possibility of modifying graph with visualization parameters which are:

- Detail

  Defines the level of detail in which the graph objects will be visualized. There are three levels of detail, high, medium and low, and objects for each level are defined individually for each supported technology. For example, low detail could mean that only databases and root directories will be shown in the graph if a database technology is visualized. On the other hand, the high detail level displays all nodes that are created in the graph and are available for visualization including columns of database tables and data items of data sets.

- Direction

  Allows a user to either only visualize forward edges which are edges going from the selected objects or to visualize backward edges which are edges that lead to the objects selected. Alternatively, both sets of edges can be shown in the graph.

- Flows

  Allows hiding indirect data flows.

- Filters

  Allows a user to either show objects of all technology groups or only objects of group *DBs, files & reports*, *Database objects* or *Important objects*. These groups are not mutually exclusive and technologies in each group are preconfigured.

- Steps Displayed

  Sets the number of data flow steps shown in the visualized graph. Its value is between 0 and 10.

There are also other filters available. Nodes sometimes only serve as a container for organization and nesting of other graph nodes, but are not relevant for data lineage and their visualization would only result in a more complicated graph with no added value. For this reason, MANTA Flow provides vertical filters that remove these 'container' nodes from the visualized graphs. Vertical filters are defined as technology-specific and a user cannot modify them. A user can, however, use horizontal filters which can remove nodes that represent inner transformations, inner tables and procedures and functions. You can see the MANTA Flow's *Viewer* interface in Figure 2.8.

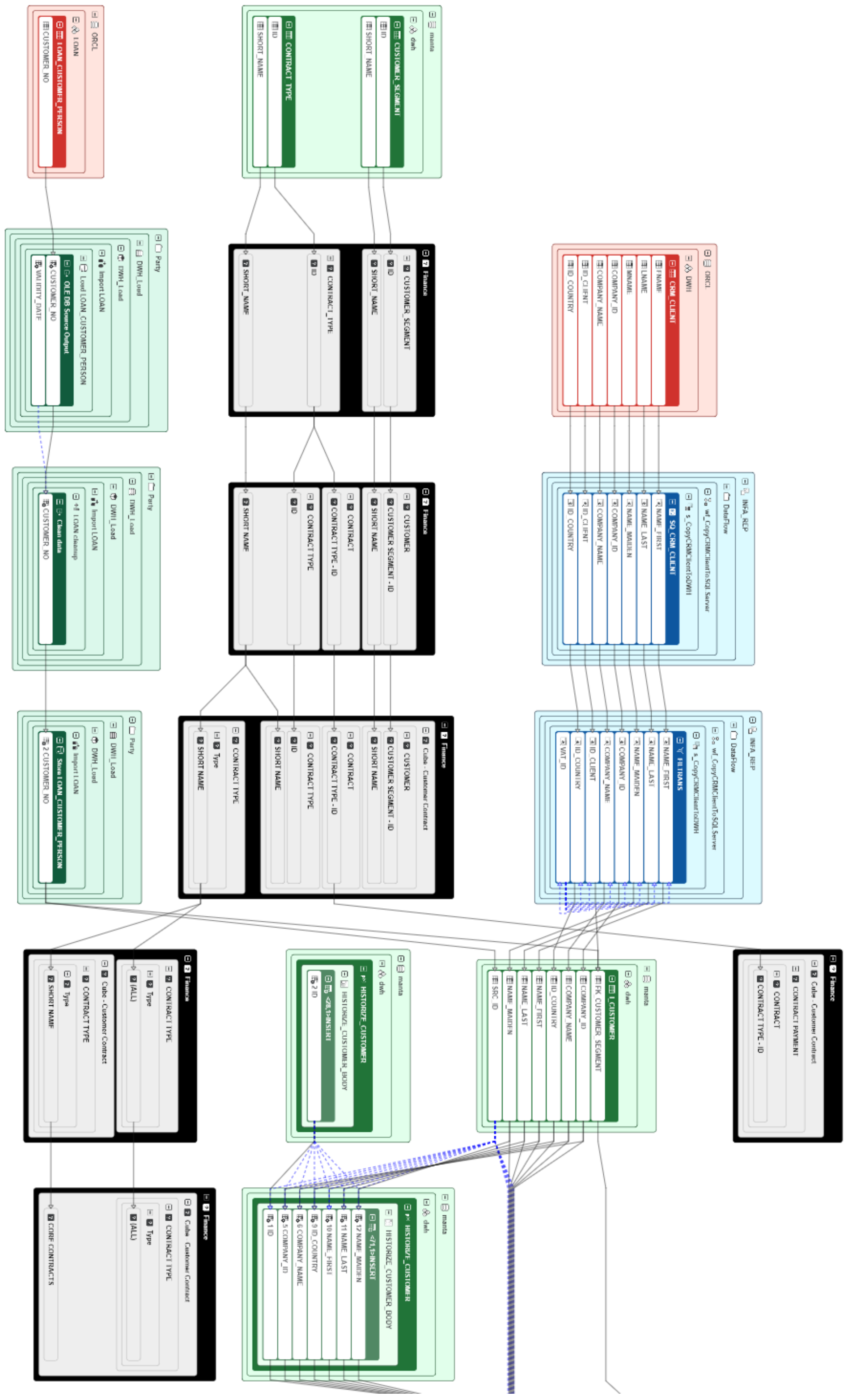
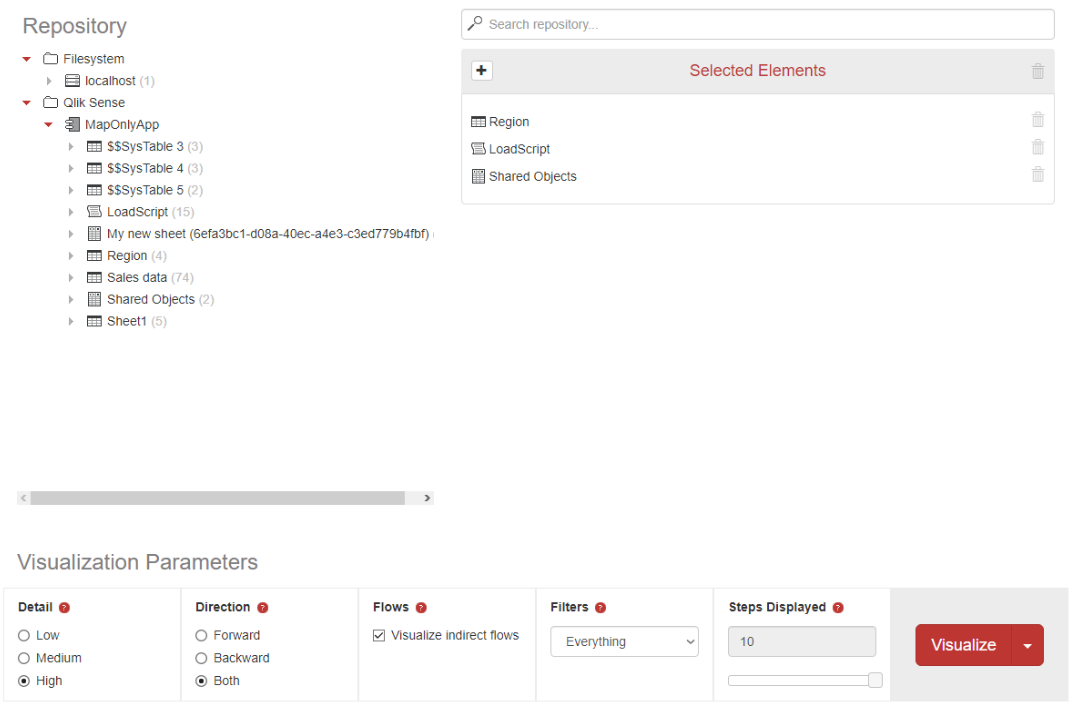


Figure 2.7: A data lineage visualization example of data flows of several technologies created by the MANTA Flow platform (green - MS SQL, orange - Oracle, blue - Informatica PowerCenter, gray - SSAS).


Repository

- Filesystem
  - localhost (1)
- Qlik Sense
  - MapOnlyApp
    - $SysTable 3 (3)
    - $SysTable 4 (3)
    - $SysTable 5 (2)
    - LoadScript (15)
    - My new sheet (6efa3bc1-d08a-40ec-a4e3-c3ed779b4fbf)
    - Region (4)
    - Sales data (74)
    - Shared Objects (2)
    - Sheet1 (5)

Search repository...

Selected Elements

- Region
- LoadScript
- Shared Objects

Visualization Parameters

Detail	Direction	Flows	Filters	Steps Displayed
Low [✓ High] Medium	Forward Backward [✓ Both]	[✓ Visualize indirect flows]	Everything	10

Figure 2.8: The MANTA Flow platform's *Viewer* interface.

3. Analysis of Qlik Sense

In this chapter, we will analyze Qlik Sense to understand what needs to be done in order to create a module for data lineage analysis of Qlik Sense applications. The most important information for us will be how Qlik Sense stores its metadata, how it can be retrieved from the server, what the object model of a Qlik Sense application looks like and what relations there are between individual objects of the model.

3.1 Metadata in Qlik Sense

Metadata or even application data in any system or an application can be saved in several ways to store the state of the program. Using JSON or XML files for this purpose is a common practice in the world of BI tools. Qlik Sense, however, saves its application data in QVF[1] files where each file contains all data and metadata of a single application. This means that whenever an application is opened in Qlik Sense, only a single file has to be loaded. QVF files are binary and it is not possible to read their content directly. Therefore it is not possible to simply open one such file and find all metadata information from it directly - we have to use the Qlik Sense Engine to process the file (open the application) and then, using a Qlik Sense API, we can retrieve information in form of responses to our server requests.

3.1.1 Qlik Engine API

Because only a Qlik Sense server can read and work with the information in a QVF file the only way to retrieve the information we need is to communicate with the server. Qlik Sense provides APIs and SDKs for building visualization extensions, managing data on the server, integration of Qlik Sense in windows applications or building custom data connectors [20].

In this section we are going to describe how the API works, how does the communication protocol look like and what kind of information can be retrieved using it.

Overview

The API that is useful for our scanner module is called *Qlik Engine API*, sometimes also called *Qlik Engine JSON API*. It is a Web Socket protocol that uses JSON to pass information between the QES[2] and the clients. The Qlik Engine API works on any platform and with any programming language that includes a Web Socket library [17].

This protocol uses generic objects. The main concept is that a method points out to a JSON object that is generic and that has a hierarchical structure. A generic object:

[1]Qlik View File, which was originally used in the older Qlik product - Qlik View

[2]Qlik Sense Engine Service, the application service that handles application calculations and logic [5].

- Can be a sheet, a story, a list object, a hypercube, a slide, a bookmark, a dimension, a measure or a pointer to another generic object.
- Is hierarchical, meaning that it contains the definition of all its children and grandchildren.
- Can be the concatenation of several generic objects. For example, a generic object could be both a list object and a chart. In that case, it would contain the definition of a list object and the definition of a chart.
- A generic object can get any type. This implies that the engine does not care about the type of the object.

The Qlik Engine API can be used to communicate between a client, for example, our scanner module, and the QPS[3] or between the QPS and QES in case of Qlik Sense server installation (Qlik Sense Enterprise), as shown in Figure 3.1, or directly between the client and the Qlik Sense Engine Service in case of Qlik Sense local installation (Qlik Sense Desktop), see Figure 3.2.

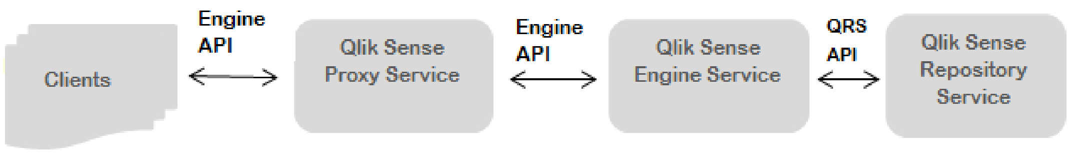


Figure 3.1: APIs that can be used for communication between individual Qlik Sense services in case of a Qlik Sense server installation.

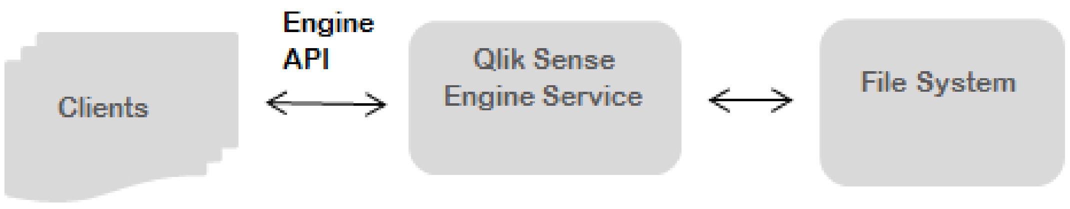


Figure 3.2: APIs that can be used for communication between individual Qlik Sense services in case of a Qlik Sense local installation [17].

The difference is caused by the fact that Qlik Sense Desktop does not need any authentication since it is intended to be used exclusively by the user on whose machine the Qlik Sense installation runs.

It is important to mention that it is not possible to communicate with Qlik Sense cloud instances [17]. It is because there is no API available for retrieving metadata from Qlik Sense Business.

[3]Qlik Sense Proxy Service, manages the Qlik Sense authentication, session handling, and load balancing [16].

Communication

The API works on the principle of exchanging messages, following the JSON-RPC 2.0[4] specification, between the client and the server. Each message sent to the engine represents one request to perform an action on a particular JSON object. This object is specified by including object's handle in the request. If the response returns an object, that object's handle is included in the response [18].

The structure of messages sent to the engine has the following members [18]:

- jsonrpc

  Mandatory member. The version of JSON-RPC, equals 2.0.

- id

  Optional member. Identifier established by the initiator of the request. If this member is not present, the RPC call is assumed to be a notification.

- method

  Mandatory member. Name of the method.

- handle

  Mandatory member. The target of the method. The member handle is not part of the JSON-RPC 2.0 specification.

- delta

  Optional member. Boolean. If set to true, the engine returns delta values. The default value is false.

  Example of use:
  The delta member is set to true to get the delta of the layout or the delta of the properties of an object.

- params

  Optional member. Sets the parameters. The parameters can be provided by name through an object or by position through an array. Each method requires a different set of parameters.

- return_empty

  Optional member. Returns default and empty values.

- cont

  Optional member. Non-validating call. Is thrown if the object is not in a validated state.

A request object can look as follows:

[4]A stateless, light-weight remote procedure call (RPC) protocol using JSON as a data format [12].

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "method": "DestroyChild",
  "handle": 2,
  "params": [
    "LB02"
  ]
}
```

The structure of messages sent from the engine has the following members [18]:

- jsonrpc

  Mandatory member. The version of JSON-RPC, equals 2.0.

- id

  Mandatory member. Identifier. This identifier must be identical to the identifier in the request object.

- result or error

  Mandatory member. The member result is required on success. In case of failure, the member *error* is displayed. The *result* member contains the data resulting from the action requested. Can be empty or it can, for example, return metadata of some object.

- change

  Optional member. Handles of the objects that have been updated.

- closed

  Optional member. Handles that have been released (following a remove for example).

A response object can look as follows:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
      "qReturn": {
      "qType": "GenericObject",
      "qHandle": 5
      }
    },
  "change": [
    5
  ]
}
```

Methods

As mentioned above, the API can be used for several different use cases which mostly work with object structure and relations between them. The methods provided by this API are able to retrieve, modify and create objects in the QES which makes it very convenient for metadata extraction needed in our case.

The fact that it is not possible to get all metadata we need in one response from the engine means that metadata of individual objects have to be extracted in several message exchanges and the data is going to be split into several JSON objects each containing data retrieved as a response to one message.

3.2 Objects in Qlik Sense

In this section, we are going to describe what objects are used in Qlik Sense, what structure they create and what relations are between them.

3.2.1 Methodology

Before any analysis of metadata could be performed, we first had to find ways how to examine what metadata we can get, what is its structure and what information can be retrieved. APIs of BI tools or databases are usually not built to provide nicely-structured data needed for data lineage. This is the reason why analysis is a major part of data lineage module development when compared to, for example, a piece of software built from scratch without any limitations or dependencies.

Since the data lineage output graph is supposed to show the structure of a Qlik Sense application as accurately as possible to help users navigate across it, we first had to determine what objects are present in the Qlik Sense Engine. After we determined the objects that we could encounter in the engine, we created approximate structure of objects - parents, children and relations between various objects.

After we had the list of objects and their mutual relations mapped, we used Qlik Sense Engine API Explorer, a Qlik Sense helping tool for developers, to see which metadata is returned in responses from the engine when different requests are sent. Once we had the object definition metadata collected we could analyze the JSON objects and try to look for as much metadata which can be useful for data lineage as possible. As mentioned earlier, not all information can be retrieved and therefore we had to pay close attention to detail as some data can be stored under non-intuitive keys or be absent altogether if some object settings are left blank.

Qlik Sense provides Qlik Engine JSON API reference which, among other things, defines the structure of JSON objects returned in response messages by all methods. However, these definitions are usually very general and do not provide the detailed information that we need. This is why manual analysis of metadata was performed and hundreds of JSON objects were analyzed to map JSON keys to object properties assigned via Qlik Sense's user interface.

Below is the list of objects important for data lineage with their description, some important properties or behavior and fields important for data lineage. We split the objects described into two groups - data layer and presentation layer.

While the data layer provides an application with data defined by expressions, presentation layer creates the structure where this data can be input. Even though these two layers exist together, their functionality differs greatly and therefore it makes sense to distinguish between the objects of the data layer and the objects of the presentation layer.

3.2.2 Data Layer

There are only two data objects that Qlik Sense uses in the application - tables and fields[5]. Naturally, a table is a named collection of its fields and each field belongs to exactly one table. The process that precedes the creation of tables and fields that are used in an application is somewhat complicated, however, it provides users with great customization options, and thanks to that, data from almost any data source can be loaded into the application.

At the beginning of the data-loading process, there is a *Load Script*, which contains instructions about what data shall be loaded, from where it shall be fetched and what transformations have to be performed. Each data-loading action executes this script and the tables and their fields that are present in the memory at the end of the script execution are saved on the server in the application's QVF file. We describe the *Load Script* itself, its syntax and statements in an own section below.

All connections to data sources are terminated at the end of the *Load Script* execution and from this point on the application only works with the data that is saved in its QVF file. The only exception to this are system fields which provide dynamic data information, for example, values of the field *$Table* contains all names of loaded tables and field *$Rows* contains the number of rows in the tables.

Fields in Qlik Sense are always used in visualization expressions whose syntax is also described in a separate section below and no other data usage is available. It is not possible to create new fields or tables after the termination of *Load Script* execution, therefore the data available for use is final until data-loading is performed again.

If a user tries to load a new table named the same as a table already loaded, Qlik Sense automatically renames every new table, adding a suffix *-tableNumber*. Qlik Sense automatically concatenates two tables if it detects that two tables have the same number of fields and all of these fields have the same names unless these tables are explicitly split.

On the other hand, if two fields in any 2 tables have the same name, the latter is renamed into *<table name>.<field name>*. It can happen that the former field is renamed instead of the latter if *Load Script* is modified to explicitly define the same name of the fields. Since Qlik Sense ensures that there are no two fields named the same, expressions refer to the fields always by their name and never together with their table name, as we can sometimes see in SQL (*<table name>.<field name>*) unless it is the above-mentioned case, when this form is the name of the field itself and not the field's parent table specification.

Data flows on the Data Layer are visualized in Figure 3.3.

[5] A more suitable term to use would be *columns*, however, Qlik Sense refers to them as *fields*.

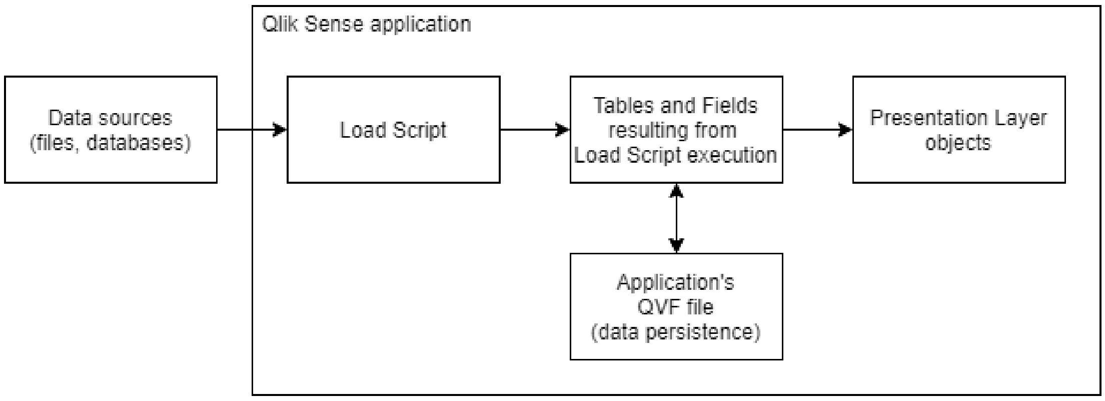

Qlik Sense application
Data sources (files, databases)
Load Script
Tables and Fields resulting from Load Script execution
Presentation Layer objects
Application's QVF file (data persistence)

Figure 3.3: Diagram of data flows on the Data Layer.

3.2.3 Presentation Layer

There are many objects on the presentation layer and therefore we decided to describe the analysis outcome of each object separately. As mentioned earlier in this chapter, presentation layer objects only use fields of the data layer in form of visualization expressions, whose syntax is described later in this chapter.

In our analysis, we will list objects from the most elementary ones, dimensions and measures, while gradually progressing to bigger objects ending with Sheets. Then we will proceed to describing objects of the *Stories* functionality.

Dimension

Dimensions determine how the data in a visualization is grouped. For example: total sales per country or number of products per supplier [4]. They are usually used as the x-axis in graphs or as slices in pie charts.

A dimension can be either *single-use* or *reused*. The difference between them is the owner of the dimension.

A *single-use* dimension is, as its name suggests, used one time only and this means that once a dimension is defined, it cannot be referenced from elsewhere and other objects do not know about its existence at all. They usually don't have an explicit name, even though they are provided with a unique dimension ID. The owner of a *single-use* dimension is the hypercube that contains its definition. The concept of the hypercube is described later in this section.

The *single-use* dimension is defined by a single field expression which prevents it from being a drill-down dimension (a kind of dimension that has got several, typically hierarchical, definitions, such as year-month-week and which can be changed by the user interaction).

A *reused* dimension, on the other hand, is a dimension that has got its own name and other objects can see it. The owner of this kind of dimension is the application itself and it is not possible to share these objects across applications. It can have several expressions defining it in case a user wants to create a drill-down dimension. If this dimension is used in a visualization, it is referenced to by its unique dimension ID.

Dimension object definitions can contain several properties that can be set as expressions. Below is a list of all dimension definitions and properties with possible expressions and properties identifying the object - ID or label.

- Definitions

  Field definitions of dimensions. They are provided as expressions that are used to creating them. When a *reused* dimension is referenced, this array of definitions is empty. The ID of the referenced dimension and field definitions are mutually exclusive.

- Field Labels

  An array of field labels [19]. The default value is an empty string and it can only be changed programmatically using Qlik Sense APIs.

- Label Expression

  An expression that's result is the name of the x-axis in the graph. In case this property is empty, the dimension's first field definition is used as a label.

- ID

  A Qlik Sense-generated short alphabetic string defining the ID of the object.

- Saved name

  The name under which the user sees the *reused* dimension in the user interface. Empty for *single-use* dimensions. Does not have to be unique.

A *reused* dimension can not have assigned any advanced styling properties like *single-use* properties. However, a *reused* dimension can be referenced in a hypercube using its ID and the hypercube can extend the *reused* dimension with these properties for that particular usage.

For example, a dimension used in a bar chart allows a user to limit the number of bars displayed using a static value or an expression. This is an additional property that is specifically based on the visualization used and cannot be stored as a property of a *reused* dimension. This property, however, can extend the referenced dimension when used in the bar chart. Any other usage of the *reused* dimension will not include this property.

See Figure 3.4 for a demonstration of dimension fields on a simple graph.

Measure

Measures are calculations used in visualizations [14]. The easiest way to think of measures is the y-axis in a bar chart. A dimension typically groups data according to the dimension's definition and the measure object performs a calculation on the grouped data to determine the *measure* of each group which is visualized in the graph or some other visualization.

Example: we have a table of sales and cities in which sales were made. We can define the dimension as 'City' and Measure as 'Sum(Sales)'. The hypercube, which is in charge of data preparation for visualizations, groups sales by 'City' and for records of each group performs calculation of the sum of sales. The height of each graph bar is then based on the sum result of sales for the given city.

The structure of the measure object in Qlik Sense is very similar to the one of a dimension. Measures, too, can be *single-use* and *reused* with the same behavior as described for the dimension (extending *reused* measures in a hypercube, visibility of other objects, visualization-specific properties) except for the fact that there is no such thing as a drill-down measure. Each measure can have at most one field definition and an empty definition, logically, provides no data and results in an empty visualization.

Below is a list of properties and definitions that are included in the measure object definition which can contain expressions and are therefore interesting in terms of data lineage or properties which help us identify the object.

- Label

  The name of the measure. If label expression is empty, it is displayed as the measure label in the visualization (if both label and label expression are empty, measure definition expression is used).

- Label Expression

  Defines the expression that's output is to be used as the label of the measure in the visualization.

- Measure Definition

  Defines the expression that shall be used to calculate the value of the measure.

- Expressions

  This parameter is used when cyclic measures are used, which is a functionality that allows users to change measures used in visualizations while using it. This is a functionality that was fully supported in Qlik View (an older product of Qlik reporting ecosystem) and mostly workarounds and own extensions are used to enable this functionality in Qlik Sense.

- Trend Lines

  Trend lines are measure objects that display some data visualization based on previous data in the visualization. For example, in a bar chart of sales per month, we can add a trend line displaying total sales from January to the bar representing monthly sales. This graph would then show sales bar for e.g. March and above that a point of trend line representing sales for January, February and March combined.

- ID

  A short alphanumeric string uniquely identifying the measure.

- Saved name

  The name under which the user sees the *reused* measure in the user interface. Empty for *single-use* measures. Does not have to be unique.

Even though Qlik Sense stores the static string of the label and the expression-defining label in two different JSON keys, there is only one field in the user interface which is used for setting the label of measure. Qlik evaluates the value of this field and if it matches the expression, it is stored into the key of the label expression, otherwise, it is stored into the key of the label. When creating the visualization, the program looks into label expression key and if its value is empty, it looks into the label key's value for determining the label of the measure. In the case of *reused* measures, saved name property is used as the measure's label primarily.

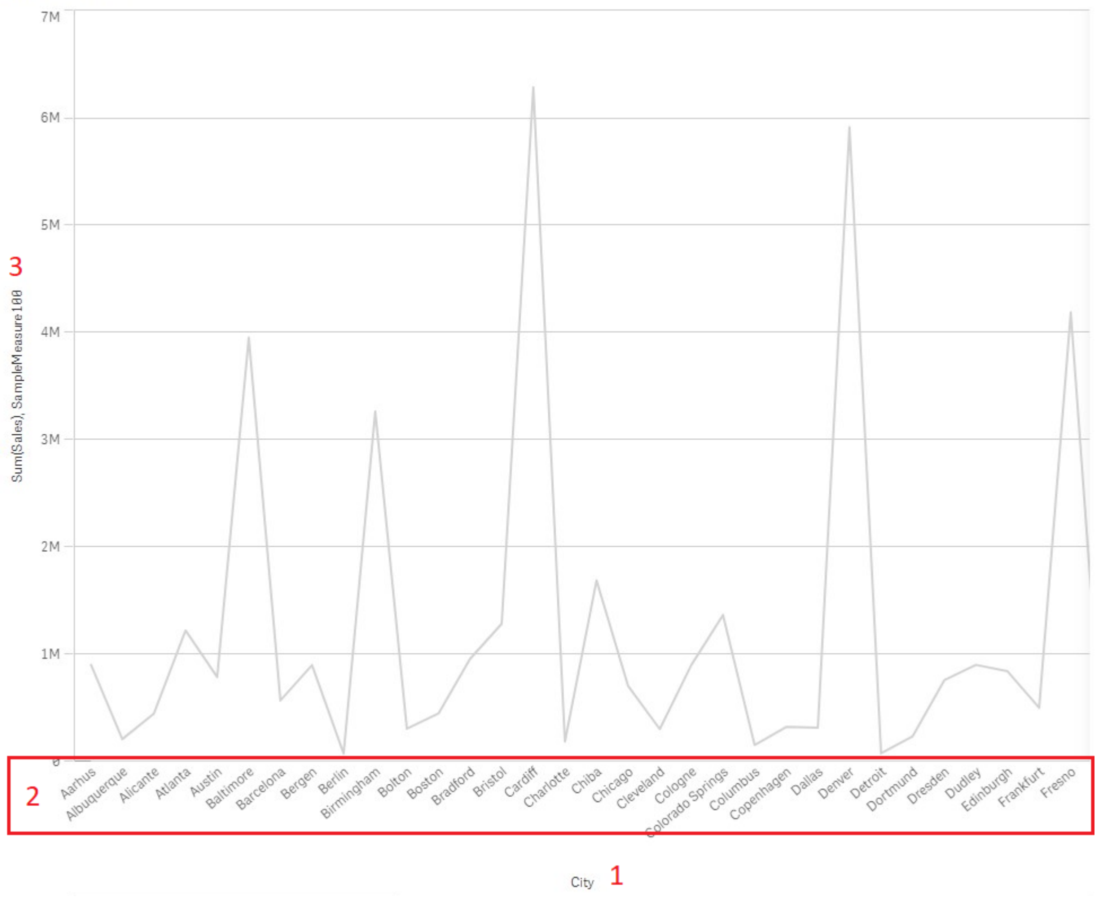


Figure 3.4: Dimension, Measures and some of their properties in a chart. 1 - Dimension label, 2 - Dimension field labels, 3 - Two measure labels, the first one being measure expression (when no explicit label or label expression is provided), the other being *saved name* property of a *reused* measure.

Hypercube

Hypercubes in Qlik Sense represent extractions of the data loaded for apps. A typical example of such data is the data needed to display a particular type of visualization object, for instance, a bar chart. Configuration of what the contents of a hypercube should be consists mainly of defining a set of dimensions that should be observed by the hypercube, and a set of measures that should be computed based on those dimensions.

The resulting hypercube is a table where the first set of columns represents dimensions, and the second set of columns represents the measures. Each row of the table will contain one unique combination of dimension values along with measures computed as if those dimension values were selected [10].

Hypercube's dimensions and measures can be used actively or they could be defined as alternative objects. Active dimensions and measures are displayed in the visualization that's parent of the hypercube immediately after the visualization is displayed. A user can, however, interact with the visualization and change the dimensions and measures displayed. The options he or she has are the alternative dimensions and measures defined in the hypercube. Once an alternative measure or a dimension is picked, the visualization is redrawn to match the *currently active* dimensions and measures.

Hypercube object definition only contains dimension and measure definitions (or references to *reused* objects) that are active. Alternative objects are stored in a separate hypercube object definition which is nested in the definition of the hypercube with the active dimension and measure definitions forming a hypercube-in-a-hypercube structure.

User interactions do not change these two hypercubes since the changes in the visualizations are just temporary and once the application is closed, these changes are lost.

The definition of a hypercube has got the following items important in terms of data lineage:

- Dimension object definitions

  A list of dimension object definitions. In the case of *single-use* dimensions, a complete definition is placed here (and nowhere else), including property values. When a *reused* dimension is used, dimension definition expressions are omitted and instead a dimension ID is present here that provides reference information. All other properties of a *reused* are specified here.

- Measure object definitions

  This item has got the same meaning as the one for dimension object definitions, but it is intended for measure objects.

- Hypercube of alternative dimensions and measures

  Contains dimension and measure object definitions of alternative objects.

- Calculation condition

  An expression that is evaluated before hypercube calculation is performed. If it evaluates to false, the calculation is not performed and visualization does not display any data. This is usually used for checking whether the input data is valid for visualization. A custom message (which can also be an expression) can be shown if the condition evaluates to false.

It is important to note that a hypercube does not have any name as it is more of an aggregation and logical object (groups dimensions and measures and creates a table for visualization) and a user may never know that this object exists at all. The metadata structure, however, relies on this object because it separates active and alternative dimensions and measures in visualization and creates a useful layer between the visualization and the data-providing objects.

Report Item

So far we have been talking a lot about visualizations and report items are a general term to refer to these objects across all BI tools. They can sometimes be referred to as gauges, charts, visualizations or graphs. Report items are the essence of BI tools since it is these objects that deliver the actual value and without which would the tools such as Qlik Sense or PowerBI lose almost all of their business value. It is not surprising then that visualizations are the most complicated object in Qlik Sense.

Qlik Sense is distributed with a set of 20 standard visualizations that are usually sufficient for most users, however, in order to provide as much functionality for the customers as possible, Qlik Sense provides APIs, briefly mentioned earlier in this chapter, that allow users to create practically any visualization. Since the set of custom visualizations is endless, we are not going to analyze these custom visualizations and our focus will be set on visualizations provided in Qlik Sense released in June 2019.

Similarly to dimensions and measures, visualization can too be *reused*. When a user wishes to create a visualization that can be used more times, he or she can create a visualization with a hypercube and all settings of the report item and then store it as a *master item*, which is the name used in Qlik Sense for *reused* report items. Unlike dimensions and measures, once the *master item* is used in a sheet, its properties can not be changed individually. If a property is changed in a *master item*, this property is changed in every single usage of it. Therefore instead of a definition of a report item in the list of children of a sheet, there is simply an ID of the *master item* that shall be used together with information about the placement of the visualization in the sheet.

Because each visualization in Qlik Sense needs different data and allows different customization (for example a bar chart allows a user to set the color of all bars and pie chart allows a user to place the legend in various locations of the visualization), there are also differently-structured report item objects.

Nevertheless, there are some items in report item definition objects that all report items share regardless of the visualization type.

- Title

  The title of the report item that is shown at the top of the visualization and can be an expression. If no title is provided, it is left empty.

- Subtitle

  The subtitle of the report item displayed right under the title. Can be an expression, too, and if no value is provided, it is not present in the visualization.

- Footnote

  The footnote of the report item. Same as subtitle can be an expression and if it is empty, it is not included in the visualization.

- Reference Lines

  A list of reference lines defined for the report item which can be set as expressions. Reference lines in the graph are graphical projections of targets, warning points or even comparison with data from a year ago that help comparison of current data with some preset *reference value*.

- Master Data

  If the report item is *reused*, it has an additional title, description and label expression fields defined so that a user can understand what *master item* he or she is using. The title is used if label expression is not used and both of these fields do not have to be unique. The description can also be written using expressions.

All of the standard report item definitions have the same structure that includes the above mentioned items, a hypercube used as a data source and some report item-specific properties except for 5 report items with a little bit different structure - *filter pane, container, map, button* and *textimage*.

Standard Report Items

A standard report item, as mentioned above, usually only has 3 groups of fields in its definition - common properties of a report item, such as the title, subtitle, reference line definitions or master data in case of a *reused* report item, a hypercube and visualization-specific properties. Bar chart, histogram, scatter plot or waterfall chart are some of the visualizations defined in a standard way utilizing a hypercube.

The standard report item's properties can usually be an expression as well as a constant value with very few being constants-only. These properties are in most cases simple key-value pairs in the JSON definition of a visualization. A constant value and an expression is distinguished by the fact that the expression conforms to the syntactic rules of visualization expressions described later in this chapter and usually start with an equals sign (=).

Coloring of a standard object is a little bit more complicated than a typical property. Qlik Sense offers different coloring options for different visualizations, but it is always one of these:

- Solid color coloring

  This is the only coloring not important for data lineage as it is only set as a constant value.

- Coloring by expression

  Coloring of bar chart bars or pie chart slices can be defined by an expression, for example using the height of a bar.

- Coloring by dimension

  A dimension is used as the base for coloring, where different values of a dimension have different colors. Color scheme and range can be picked and Qlik Sense automatically colors the visualization.

- Coloring by measure

  Similarly to dimension, when a measure is used for coloring, its values are used as the base for coloring. A bar chart, for example, can be painted in a blue color scheme where low bars have a lighter blue shade and the higher the bar is, the darker blue it is.

Solid color coloring is unimportant and coloring by expression is set using a simple key-value entry in the definition of the report item. Dimension- and measure-based colorings, however, are stored in a special triple of values - label, expression and type. This type is common for some object properties in Qlik Sense which can reference a *reused* dimension or measure. In this case, the label value is unused and copies the expression field value.

If the user wants to use a newly defined dimension or a measure, its definition is saved as an expression into the expression field and the type is assigned to *EXPRESSION*. However, the user can also use an already existing *reused* dimension or a measure. In this case, the field expression contains the ID of the *reused* object and the type is set to *LIBRARY_ITEM*. If the dimension- or measure-based coloring is not used, label and expression fields are empty strings and the type of this triple is *EMPTY*.

Container

A container is a very simple structure - it contains a list of report items definitions (or references) which can be displayed inside the container. These report items can either be displayed statically and change based on a show condition set for the report item or they can be changed using a tabbed bar in the top, similarly to web browsers.

The item in the container's definition interesting from the data lineage perspective is, in addition to the common report item fields, a list of report items included (either directly defined inside or using a master item reference). A container cannot contain another container inside of it. Each of these report items can also have a *show condition* defined which prevents the report item from being shown in the visualization if it is evaluated to false.

Filter Pane

A filter pane is a collection of dimensions that can be used for the application-wide dimension filtering. It is quite difficult to understand the functionality easily, so a set of figures can be used for aid. Let there be a graph showing sales per customer per city as shown in Figure 3.5.

If the user wants to filter the visualization to Baltimore, Dallas and Dresden and customers Customer A, Customer B and Customer C, he or she can use the sheet UI's *Selections* functionality or a filter pane listing all values of the dimensions included in it (in this case City and Customer) can be provided (see Figure 3.6).

The user can simply click on items that he or she wants to filter in or out and graphs in the application using filtered dimensions update themselves. So if we choose the above mentioned required filter, we can update the graph easily (see Figure 3.7).

Despite a difficult use case, its structure is fairly simple - it uses ListBox items, which represent dimensions in a property-limiting manner, and *reused* dimension references to populate itself with relevant data entries (as seen in Figure 3.6) and transforms the user action (clicks) into selection updates.

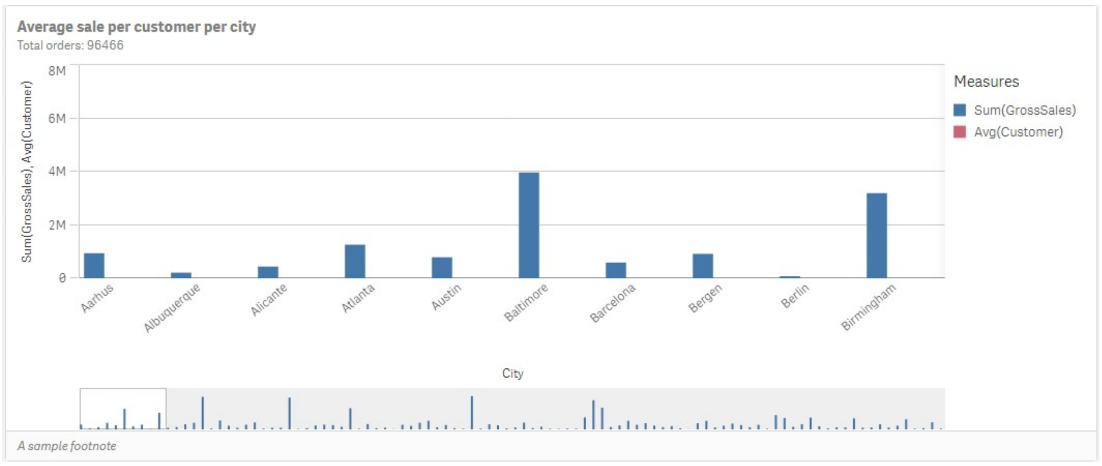
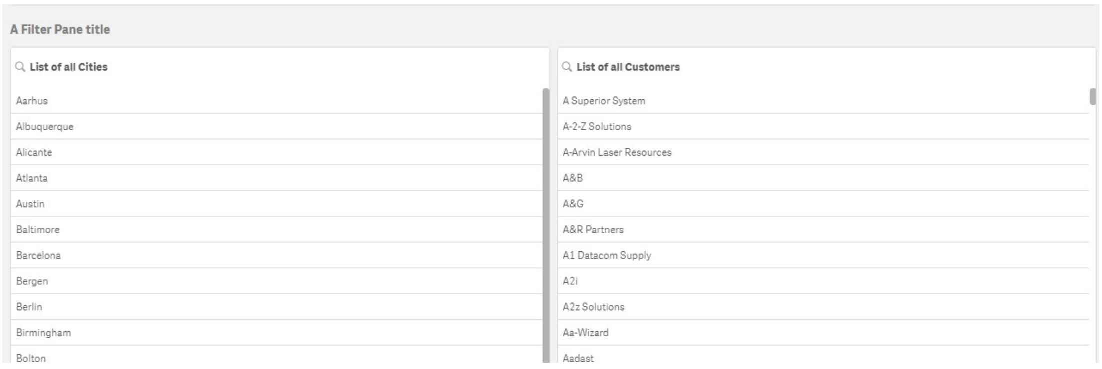


Figure 3.5: A standard bar chart used showing average sale per customer per city.


A Filter Pane title

List of all Cities	List of all Customers
Aarhus	A Superior System
Albuquerque	A-2-Z Solutions
Alicante	A-Arvin Laser Resources
Atlanta	A&B
Austin	A&G
Baltimore	A&R Partners
Barcelona	A1 Datacom Supply
Bergen	A2i
Berlin	A2z Solutions
Birmingham	Aa-Wizard
Bolton	Aadast

Figure 3.6: A filter pane with two dimensions - City and Customer.

Filter pane definition contains no fields interesting in terms of data lineage except for common report item fields and a list of ListBox definitions and *reused* dimension references.

A ListBox definition, on the other hand, contains fields similar to the fields of a dimension object definition - name, which is its label in the filter pane and can be an expression, field definitions and field labels, which are the same as dimension definitions and labels, and expression that is used to sort data entries of the ListBox in the filter pane.

Map

A map is a very complex report item with many fields possibly containing expressions and therefore data flows. A map is a collection of layers that display points, charts or lines on a map positioned according to a geographic location. This way there can be fifteen small pie charts, each for some country or other location on the map, each showing data limited to some area. Except for the common report item properties, its important data flow information is located in the above mentioned internal objects called layers.

There are 5 types of visualization layers:

- Line Layer

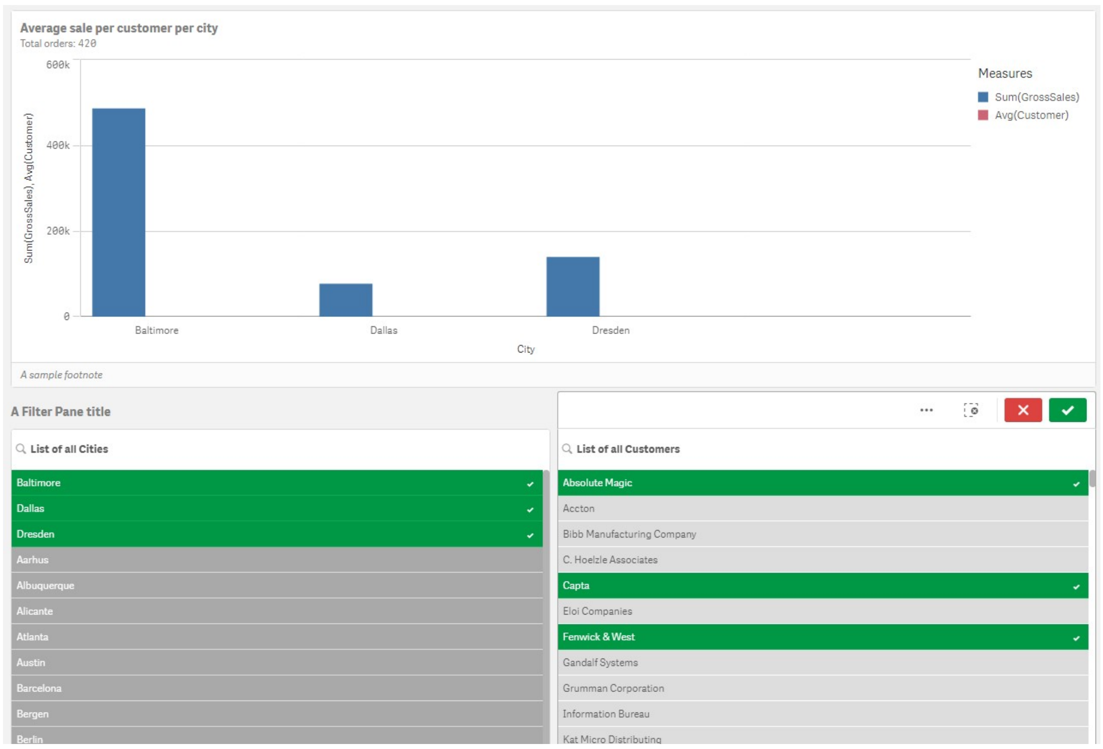


Figure 3.7: Resulting bar chart after filter pane was used.

Creates lines between pairs of points on the map.

- Density Layer

  Displays a measure based on location.

- Area Layer

  Paints areas (cities, counties or even countries) which are provided to it in the hypercube.

- Point Layer

  Marks points on the map based on the hypercube data.

- Chart Layer

  Displays a defined chart for each location on the map using the hypercube data.

In addition to these layers, there is another layer - the background layer. A background layer defines how the *base* of the map shall look like - if it shall be a world map, a custom map or, for example, a zoomed-in map.

Layer definitions have a very varying structure that contains many properties where expressions can be used. These properties usually define the location, or locations in case of the line layer, and visualization properties of the individual layer. A triple structure consisting of label, expression and type mentioned in the part about coloring using dimensions and measures, is used a lot in layer property definitions.

In order to be able for a layer to provide some data to the map, it has to contain some definition of what data shall be calculated and delivered to the map object. This is why even layers, except for the background layer, have got a hypercube defined inside of their definition objects.

Button

A button is a visualization that is useful when assistance with selections is desired or navigation to sheets, stories or websites is needed. When creating a button, it can have assigned an action that is performed when a button is clicked. Among the most important actions are bookmark application, clearing selection or switching to a different sheet or a story.

TextImage

A textimage visualization is a very simple minor report item that is used to display some text/image only. If it displays a text, it can be defined as an expression and if any fields are used, they are defined in the textimage's hypercube as dimensions and/or measures. This object can also be a background image that is loaded from the server's media library. The image is set using Qlik Sense UI and does not contain any expression, just a relative path in Qlik Sense server's file system.

Sheet

A sheet is a top-level container in a Qlik Sense Application that contains the report items as its children. We could find an analogy between a sheet in Qlik Sense and a report or a dashboard in other BI tools. A report item can either be directly defined as a sheet's report item (therefore it is only used once in the whole application) or it can reference a *master item*.

A sheet definition contains no hypercube or properties. Expressions can, however, be used in its title expression and description. In case a title expression is not used, a static title is shown as the name of the sheet. Children of a sheet are either nested whole definitions of report items or they can be present in form of a reference to an ID of a *Master item*.

Objects until here were part of the *Analysis* section of Qlik Sense which can be used for the final analysis of data, but Qlik Sense provides one other functionality for presentation - *Stories*. The following four objects represent stories, although they are often closely tied with the *Analysis*, especially report items.

Snapshot

A snapshot is a graphical representation of the state (type and data) of a data object at a point in time that can be used when slides are created in data storytelling. The snapshot is a copy of the state. This means that the state of the snapshot does not change when the state of the corresponding data object is updated. A snapshot contains the data and the layout needed to render a visualization object [22].

It means that a snapshot is not saved as an image, but it saved as a metadata object the same way a visualization is, so that it is possible to render it any time. A snapshot is always tied to a visualization in some sheet and IDs of these two objects are contained in metadata objects of snapshots.

Slide Item

Snapshots are not the only objects that can be used in the storytelling feature of Qlik Sense. A user can use text boxes, which are just static text fields, shapes, images from Qlik Sense server's image library and sheet mirroring. In terms of data lineage, the only interesting features for us are images since we could potentially map an image to its source in the server's file system and sheet mirroring.

If a user decides to use a sheet in a story, the slide item references a sheet by ID and it is possible to interact with it during a presentation - filters can be used to show various insights or to make visualizations more readable.

Slide

A slide is a simple collection of slide items and snapshots with placement information. All slide items belong to one slide only and their definitions are nested inside the definitions of their parent slide.

Story

Atop of slides, there are stories, which are slide collections. One story represents one complete presentation with slides, effects and own name. A user can have several stories created and can switch between them the same way he or she can do so in case of individual sheets of an application.

There are, in addition to the above-mentioned objects that provide data and presentation features, two other objects which do not really belong to either layer, but their importance is severe and are used commonly - variables and bookmarks.

3.2.4 Variable

A variable in Qlik Sense is a named entity, containing a data value. When a variable is used in an expression, it is substituted by its value or the variable's definition [2]. Variables are a very important feature of Qlik Sense in a way that they can be used to create some changing behavior in combination with expressions, mostly with conditional statements, or they can be used as a workaround to create some unsupported features.

For example, variables can be used for changing the language of a Sheet. Qlik Sense does not support multilingual sheets or stories functionality, however, since most titles and labels support expressions, they can be styled in a way that if, for example, a variable *lang* has value *CZ*, the expression is evaluated to the Czech string and if it has some other value, it is evaluated to its English equivalent. There are many examples of variable usage on the Qlik Sense community forum and virtually anything can be achieved by the correct usage of variables and expressions.

3.2.5 Bookmark

As mentioned several times already, Qlik Sense allows a user to create filters of the data, whether it is using a filter pane, interacting with visualizations directly or using the selection user interface. These filters can sometimes be quite complex spanning across several table fields, dimensions or measures and it is not always easy and fast to create them. However, if a filter is used very often, it can be quite time-consuming and annoying to have to create a filter every time it needs to be used.

For this purpose Qlik Sense allows users to save their filters for reuse as bookmarks and then it is very simple to apply the filter just by picking it from the bookmark list.

A bookmark is saved using a set expression, which is described in the section about visualization expressions, and it is always bound to a certain sheet, referencing it by its ID, which opens when a bookmark is applied. It is possible, though, to change sheets with keeping the filter of the bookmark. It is also possible to simply get a list of *selection fields* used by the bookmark.

3.3 Visualization Expressions

An expression is a combination of functions, fields, and operators. Expressions are used to process data in the application in order to produce a result that can be seen in a visualization. This means, for example, that instead of the title of a visualization being a static text, it can be defined as an expression that's result changes according to the selections made [25].

Expressions are a very important feature of Qlik Sense defining the whole usage of the product. If it was not for this feature, users would not be able to load data, transform it, use in visualizations and analyze it. Typically, data is prepared using SQL statements and the output of these statements is used in the visualizations. However, because Qlik Sense loads and uses data in two different steps, it uses an own expression language that makes it possible to separate these two actions.

These expressions sometimes need to perform several calculations at the same time and since they cannot be split into several smaller expressions, they can become very complex and long. Qlik Sense provides an expression editor that helps with editing, selecting valid fields and checking expression syntax.

Visualization expressions can vary in length and complexity anywhere from

```
1 + 1
```

to

```
=$(=
  Concat(DISTINCT {<[$Field]-={$(vSelection.IgnoreFields)}>}
  'If(GetSelectedCount([' & [$Field] & ']) > 0, ' & chr(39) &
  '<[' & chr(39) & ' & Concat(DISTINCT {<[$Field]={"*"} -
  {'& chr(36) & '(vSelection.IgnoreFields)} - {' & chr(39) &
  [$Field] & chr(39) & '}>} [$Field], ' & chr(39) & ']= ,[' &
  chr(39) & ') & ' & chr(39) & ']= > + ' & chr(39) & ')' , '  & ')
)
```

and even though these complex expressions are usually avoided or can be rewritten in a more understandable and simple way, there can be some use cases when they need to be used unedited.

3.3.1 General syntax

The output of an expression can be either a value that is interpreted as a string, a logical value or as a number. This way expressions can be understood appropriately in various contexts even if different value types are expected. An expression can, for example, be interpreted as a boolean value in a conditional statement and elsewhere can the same expression provide data for a visualization.

The Qlik Sense documentation defines the expression using the BNF[6] notation accordingly:

```
expression ::= ( constant                           |
                 expressionname                     |
                 operator1 expression               |
                 expression operator2 expression |
                 function                           |
                 aggrfunction                       |
                 ( expression ) )
```

where:

- *constant* is a string (a text, a date or a time) enclosed by single straight quotation marks, or a number. Constants are written without thousands separator and with a decimal point as decimal separator.
- *expressionname* is the name (label) of another expression in the same chart
- *operator1* is a unary operator
- *operator2* is a binary operator
- `function ::= functionname ( parameters )`
  `parameters ::= expression { , expression }`
- `aggrfunction ::= aggrfunctionname ( parameters2 )`
  `parameters2 ::= aggrexpression { , aggrexpression }`
- The number and types of parameters are not arbitrary. They depend on the function used [25].

3.3.2 Aggregation syntax

Aggregations are used in almost every visualization. Operations such as summation, getting minimum or maximum of a set of values, these all are aggregations and nearly every measure definition uses at least one aggregation function. They take multiple values and return a single value based on the input. Except for the Load statement of the load script, which is described below, aggregations can only be used in visualization expressions.

The syntax of aggregations differs from that of general expressions and is defined in the BNF notation followingly:

[6] Backus-Naur formalism

```
aggrexpression ::= ( fieldref                                  |
                     operator1 aggrexpression                  |
                     aggrexpression operator2 aggrexpression   |
                     functioninaggr                            |
                     ( aggrexpression ) )
```

where:

- *fieldref* is a field name
- *operator1* is a unary operator
- *operator2* is a binary operator
- `functionaggr ::= functionname ( parameters2 )`

Expressions and functions can thus be nested freely, as long as *fieldref* is always enclosed by exactly one aggregation function and provided the expression returns an interpretable value, Qlik Sense does not give any error messages [25].

3.3.3 Operators

Qlik Sense provides more operators than just +, -, *, and /. They are split into five groups [15]:

1. Bitwise operators

   All bitwise operators convert (truncate) the operands to signed integers (32 bit) and return the result in the same way. All operations are performed bit by bit. If an operand cannot be interpreted as a number, the operation will return NULL.

   Operators: *bitnot, bitand, bitor, bitxor, »* (shift right), *«* (shift left)

2. Logical operators

   All logical operators interpret the operands logically and return True (-1) or False (0) as result.

   Operators: *not, and, or, xor*

3. Numeric operators

   All numeric operators use the numeric values of the operands and return a numeric value as result.

   Operators: *+, -, *, /*

4. Relational operators

   All relational operators compare the values of the operands and return True (-1) or False (0) as the result.

   Operators: *<, <=, >, >=, =, <>, precedes, follows*

   In the case of *precedes* and *follows* operators, no attempt is made to interpret values numerically and therefore:

```
'1 ' precedes ' 2' returns FALSE

' 1' precedes ' 2' returns TRUE

' 2' follows '1 ' returns FALSE

' 2' follows ' 1' returns TRUE
```

since the ASCII value of a space (' ') is of less value than the ASCII value of a number.

5. String operators There are only two operators:

   - Concatenation (*&*)
     String concatenation. The operation returns a text string, that consists of the two operand strings, one after another.
     Example: 'abc' & 'def' returns 'abcdef'
   - Operator *like*
     String comparison with wildcard characters (* and ?). The operation returns a boolean True if the string before the operator is matched by the string after the operator.
     Example:

     ```
     'abc' like 'a*' returns True

     'abc' like 'a??bc' returns False
     ```

There is no mention of the operator precedence in the Qlik Sense documentation. However, it is possible to test the operator precedence for example in the load script or even expressions of visualizations. Thanks to this it was possible to determine the precedence accordingly (from highest to lowest priority):

1. terms inside parentheses
2. unary minus
3. bitnot
4. bitand
5. », « (right and left shift)
6. multiplication and division
7. addition and subtraction
8. like, & (string concatenation)
9. bitor
10. bitxor

11. comparisons
12. not
13. and
14. or
15. xor
16. left to right

Multiple operators in a position mean that the operations are executed from left to right, for example, *2 - 1 + 3* means that the addition is preceded by the subtraction.

3.3.4 Functions

Qlik Sense offers a very large set of approximately 400 functions [7]. Some functions can only be used in the load script, the visualizations or both. It makes no sense to describe these functions or even their groups (of which there is 25) individually. In general, almost everything is possible using the functions, there are numeric, monetary, range, table or conditional functions available.

Functions are used in a way that almost any other language uses them and as were defined together with aggregation functions in the BNF notation in the beginning of this section:

```
function ::= functionname ( parameters )
parameters ::= expression { , expression }

aggrfunction ::= aggrfunctionname ( parameters2 )
parameters2 ::= aggrexpression { , aggrexpression }
```

3.3.5 Aggregation scope

There are usually two factors that together determine which records are used to define the value of aggregation in an expression. When working in visualizations, these factors are *dimensional value* (of the aggregation in a chart expression) and *selections*. Together, these factors define the scope of aggregation [25].

If the dimension, the selection or both shall be disregarded in an expression, *TOTAL* and *ALL* qualifiers can be used as well as set analysis.

TOTAL qualifier

The *TOTAL* qualifier used in an expression disregards the dimensional value which means that the aggregation will be performed on all possible values of a field.

The TOTAL qualifier may be followed by a list of one or more field names within angle brackets. These field names should be a subset of the chart dimension variables. In this case, the calculation is made disregarding all chart dimension variables except those listed, that is, one value is returned for each combination of field values in the listed dimension fields [25].

An example can be seen in table 3.1.

Year	Quarter	Sum(A)	Sum(TOTAL A)	Sum(A)/Sum(TOTAL A)
		3000	3000	100%
2012	Q2	1700	3000	56,7%
2013	Q2	1300	3000	43,3%

Table 3.1: Example of the effect of TOTAL modifier (A is short for 'Amount').

Set analysis

Set analysis inside an aggregation overrides the selection. For more information on set analysis syntax, see the part about set expressions and set modifiers below.

TOTAL qualifier and set analysis

TOTAL qualifier and set analysis used together override both dimension and selection.

ALL qualifier

If an *ALL* qualifier is used, dimensions and selection are disregarded and cannot be overridden.

3.3.6 Set expressions and set modifiers

Before a data set is used in an expression function, this set can be additionally changed to use modifier data for calculation. This can be performed using set expressions.

Set expressions are expressions used for local and specific data selection consisting of *set modifiers*. The modifier consists of one or several field names, each followed by a selection that should be made on the field, all enclosed by angled brackets: <>. For example: <Year={2007,2008},Region={US}> [21].

A set modifier modifies the selection of the preceding set identifier. If no set identifier is referenced, the current selection state is implicit. There are several ways to define the selection [25]:

- Based on another field

  A simple case is a selection based on the selected values of another field, for example <OrderDate = DeliveryDate>. This modifier will take the selected values from *DeliveryDate* and apply those as a selection on *OrderDate*.

- Based on element sets (a field value list in the modifier)

  The most common example of a set expression is one that is based on a list of field values enclosed in curly brackets. The values are separated by commas, for example <Year = {2007, 2008}>. If values are enclosed in double quotation marks, these values can contain wildcards.

- Forced exclusion

  Forced exclusion is only available in the AND mode in Qlik Engine API and allows exclusion of specific values by adding a tilde before the field name.

  For example, expression

```
sum( {$<~Ingredient = {"*garlic*"}>} Sales )
```

Returns the sales for the current selection, but with a forced exclusion of all ingredients containing the string 'garlic'.

3.3.7 Dollar expansion

Dollar-sign expansions are definitions of text replacements used in the script or in expressions. This process is known as expansion - even if the new text is shorter. The replacement is made just before the script statement or the expression is evaluated. Technically it is a macro expansion.

The expansion always begins with '$(' and ends with ') ' and the content between brackets defines how the text replacement will be done. Dollar-sign expansions can be used with either of [24]:

- Variables
- Parameters
- Expressions

Dollar expansion using variables

Using dollar expansion with variables is the most used case. The content of the dollar expansion is the name of the variable enclosed in single quotation marks or with no quotation marks at all:

```
$( variablename )
```

The variable name can be preceded by the number sign (#) if the expanded value shall be interpreted as a number instead of a string. The difference is that the number interpretation always uses the system's valid decimal separator while the string interpretation always uses a comma. If the value of the variable cannot be interpreted as a number, it is expanded as a zero instead.

The name can also be preceded by a name in curly braces which explicitly states what state of the variable shall be used. Variables can have different values depending dynamically on different application states (selections).

```
$({MyState} variablename) expands the variable in MyState state
```

Dollar expansion using parameters

If parameters are used in dollar-sign expansions, then the variable's value must contain formal parameter marks, such as $1, $2 and $3. When expanding the variable, the parameters should be stated in a comma-separated list and they replace the variable value's marks.

```
Set MUL='$1*$2';
Set X=$(MUL(3,7)); - value of X is '3*7'
Let X=$(MUL(3,7)); - value of X is 21
```

Note: *Set* statement stores the string value stated on the right to the equals sign, *Let* statement interprets the value to the right as an expression and stores the expression result in the variable.

Dollar expansion using an expression

If an expression is used in the dollar expansion, it has to start with the equals sign and it is replaced by the result of the expression.

```
$(=Year(Today())); - returns the string with the current year
$(=Only(Year)-1); - returns the year before the selected one
```

Finally, dollar expansion can be used to put contents of a file into the load script or a visualization expression using the keyword *include* and providing a path to the file containing text that shall be inserted.

```
$(include=C:\Documents\MyScript.qvs);
```

3.3.8 String representation

In Qlik Sense, it does not matter whether a value is a date in any format, a number or just an array of characters, it is always represented as a string and Qlik Sense automatically tries to cast the string to a number or a date when it expects an input of such type. It is possible, for example, to store a string '123' into a variable and later work with it as if it was a number without any need for explicit casting. Number constants are written with no thousands separator and a decimal point is used as the decimal separator [24].

To distinguish between string literals, field names and variable references, Qlik Sense provides 4 different ways of enclosing strings which define how the string is interpreted based on the context.

If a text is enclosed in apostrophes, it is interpreted as a string literal. If it is enclosed in square brackets or grave accents (‘ ‘), it is always interpreted as a field name.

If a string is enclosed in double quotation marks and is used outside of the load script's *Load* statement where an expression is expected then it is interpreted as a variable reference and when the string is inside of the statement, it signals that the string shall be perceived as a field name [24].

It is necessary to enclose the text in some kind of string marks accepted by Qlik Sense if the string contains whitespace characters. If the string contains a character that is used as an enclosing character at the end of the string, these characters have to be properly escaped by using a double character in the string (two right brackets, two double quotation marks, etc.) or a different kind of enclosing marks is suggested.

3.4 Load Script

When working with Qlik Engine JSON API, we are capable of extracting metadata that describes what tables and fields are used in the application, what objects consume data and how is everything related. However, it is quite difficult to construct the data lineage before the data is already loaded in the application.

The Engine API lets us retrieve a list of connections, which are defined separately in Qlik Sense and stored as connection strings, so that we know which databases or file systems are used, but we cannot easily get the metadata telling us if a field A in table T was loaded by loading a column from table X of a database D1, or if it was created by loading from table Y of a database D2 while renaming the column (see Figure 3.8).

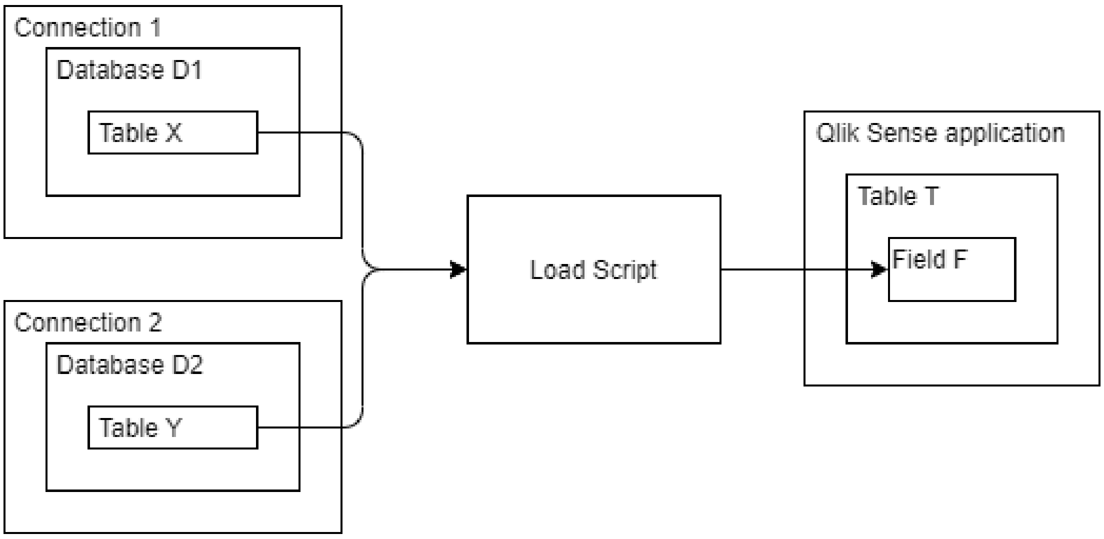


Figure 3.8: A problem of field origin.

Without parsing and analyzing the *Load Script* there is no possibility of telling the data lineage user how the data was actually created and what it actually is. A load script takes the defined connections of an application, executes its commands while substituting connection references in commands with actual connections and returns a list of tables and fields which can then be used in expressions, dimensions or measures of visualizations and other objects.

Before we continue let us mention that even though we can retrieve connections from the server it can sometimes be impossible to analyze load script because connections:

- Are not stored in the application's QVF file and migration across servers does not include migration of connections. Therefore these connections can be completely missing.
- Connections can be deleted after the needed script execution was completed and there is no problem with it since load script is not used until another execution is requested. Any invalid state of connections and the load script which appears when it is not executed raises no errors.

In this section, we will describe some basic syntax of the load script, important constructions and special cases. The load script language is very complex and we skip some parts of it which are not useful for data lineage on purpose.

3.4.1 Structure

Load script is written in a language quite similar to SQL, but it has somewhat unique structure and constructions which make it differ from the SQL, especially the functionality related to working with Qlik Sense Engine's table and field associations and QVF files.

A typical load script will commonly have these main parts:

- Variable declarations
- Subroutine definitions
- Data load from data sources
- Transformations of loaded data such as a field or a table renaming, a field value mapping or a new field derivation

In the user interface of Qlik Sense dedicated to editing the Load Script, called *Data load editor*, the script is split into several sections, which can be user-defined. However, Qlik Sense Engine API provides the load script as a whole and therefore we can work with it as a whole.

3.4.2 Statements and Keywords

Script statements and keywords are split into 3 main groups:

1. Script control statements

   These statements do not perform any action and are used only as helper statements. The statements included here are loop statements, conditionals, subroutine definition and calling or the script exit statement.

2. Script regular statements

   The most important statement group which performs all data manipulation. A typical example of regular statements are `load`, `select` and `store` statements, field alias definition or table and field renaming statements.

3. Script prefixes

   Prefixes are a group of many non-reserved keywords that provide additional directive to the script executioner. The `left/right/inner/outer join` is a very useful prefix as well as the `first` *n* keyword which limits data-loading to the first *n* rows.

Each clause of a control statement must be kept inside one script line and may be terminated by a semicolon or the end-of-line [21]. The syntax does not define any limit count for nesting control statements.

Regular statements, on the other hand, may be written over any number of lines in the script and must always be terminated by a semicolon [21].

Prefixes may be applied to applicable regular statements but never to control statements. The `when` and `unless` prefixes can however be used as suffixes to a few specific control statement clauses [21].

All script keywords can be typed with any combination of lower case and upper case characters. Field and variable names used in the statements are however case sensitive [21].

Even though there is still no final list of fields that are present in the application, load script statements can use expressions in a very similar manner to visualization expressions, only differing in the list of available expressions.

3.4.3 Keywords and identifiers

Qlik Sense does not explicitly define reserved keywords of the script language and some tests had to be performed. Only a few of statement keywords turned out to be reserved: `call, do, for, next, elseif, sub, switch` and `case`.

In addition to that, Qlik does not define any pattern for accepted identifiers. The only way to see if the character is valid in Qlik Sense is by trying all options. Moreover, some characters are forbidden as the first characters (probably to avoid mixing with other constructs in the language), while they are allowed in the second or latter character of the identifier. From ASCII characters, these are allowed characters in the beginning of the identifier:

```
A-Z a-z # $ % ? | @ \ ] ^ _
```

In other identifier positions, dot and numbers are also allowed:

```
A-Z a-z # $ % ? | @ \ ] ^ _ 0-9 .
```

3.4.4 Paths

Some statements need to have a specified path in a filesystem, for example for data loading and storing. Normally, this would be done by specifying the absolute path or relative path. When Qlik Sense *Load Script* execution starts, it creates a private value called the *working directory*, which points at a location on the server set in server settings, by default it is in the server's application data directory. When a user references a file omitting path, Qlik Sense looks for the file in the *working directory*. The working directory can be changed using the DIRECTORY statement.

However, in some cases, many different filesystem locations can be needed and since Qlik Sense is capable of connecting to cloud storage, these locations do not even have to be on the server. For this problem, the application makes use of its connections feature where a user can define a folder connection which contains an absolute path on the local or a remote filesystem. When the user wants to point to a file relatively to a folder connection, a simple connection name and relative path needs to be used:

```
lib://<connectionname>/<relativepath>/<filename>
```

In *Legacy scripting mode* of Qlik Sense, this formulation could be replaced by a string constant containing an absolute path, a path relative to the working directory or a URL.

3.4.5 Commenting

Commenting can be done either in a way common for many programming languages:

- Single-line commentary

  Begins with two slashes and includes everything until the end of the line.

- Multi-line commentary

  Begins with a slash and asterisk characters (/*) and ends with an asterisk followed by a slash (*/). Everything between these two marks is a commentary and the application does not execute this part of the script.

In addition to these two common commentary practices, there is also a load script statement called `REM` (short for *remark*) which interprets everything between the statement keyword and the next semicolon as a commentary and this text is not executed.

3.4.6 Variable parts

Using the dollar expansion, which is described in the section about visualization expressions, the *Load Script* can be altered. Because dollar expansion is performed before the statement where the dollar expansion is executed, it is possible to store part of the script in a variable and 'import' it using the dollar expansion.

There is hardly any use case for this since there are control statements or the dollar-expansion's include construction, however, if a variable contains statements and its value is expanded in the script, the statements are executed at the place in the script where it is expanded.

At the end of this section, we would like to describe some of the most important statements which are essential for the *Load Script* and form elementary functionality of it.

3.4.7 Load statement

The load statement is the most used statement of them all. It is used for loading data from 6 different sources and its BNF definition is:

```
LOAD [ distinct ] fieldlist
    [( from file [ format-spec ] |
    from_field fieldassource [format-spec] |
    inline data [ format-spec ] |
    resident table-label |
    autogenerate size |
    extension pluginname.functionname([script] tabledescription)]
[ where criterion | while criterion ]
[ group by groupbyfieldlist ]
[order by orderbyfieldlist ]
```

The structure of the load statement is very similar to the structure of a common SQL statement - there is a list of loaded fields, a data source, a `where` clause, a grouping clause and an ordering clause. `Fieldlist` is a comma-separated list of fields that shall be loaded from the source. New fields can be renamed using the keyword *`as`*:

```
source_table_field_name as field_name_in_new_table
```

Individual data sources are:

1. `From file`

   From is used if data should be loaded from a file using a folder or a web file data connection. The source file is specified using the path in the format mentioned above.

2. `From field`

   From_field is used if data should be loaded from a previously loaded field.

   ```
   fieldassource ::= (tablename, fieldname)
   ```

   The field is the name of the previously loaded *tablename* and *fieldname*.

3. `Inline data`

   Inline is used if data should be typed within the script, and not loaded from a file.

   Data entered through an inline clause must be enclosed by double quotation marks or by square brackets. The text between these is interpreted in the same way as the content of a file. Additionally, format specification defines what is the delimiter between row values, by default it is a comma, but it can also be whitespace characters or any other character. Newline character defines a new row.

4. `Resident`

   Resident is used if data should be loaded from a previously loaded table. The *tablename* defines the name of the table from which the data shall be loaded.

5. `Autogenerate`

   Autogenerate is used if the new data shall be generated automatically by Qlik Sense and its parameter *size* defines how many rows shall be generated.

6. `Extension`

   Extension is used for loading data from analytic connections, usually using Python or R languages.

There is one additional usage of the load statement which is loading from a succeeding table. It is used when the current statement shall use the table that is returned by the `load` or `select` statement that succeeds it. Qlik Sense, for example, generates a succeeding `load` when there is a `select` statement loading data from a database and the fields loaded are renamed. Qlik Sense then retrieves the table from the database with the original field names and uses the load statement to rename these fields. Example:

```
LOAD [field1] as [newName1],
     [field2];
SELECT field1, field2 from sch.table1;
```

The example above loads fields *field1* and *field2* from *table1* and fields in the resulting table are named *newName1* and *field2*.

3.4.8 Select statement

A `select` statement is a pure-SQL statement that returns the data fetched from the database as a table that can be directly used in Qlik Sense. Its syntax depends on the SQL dialect used by the driver that is used to communicate with the database. Qlik Sense never evaluates these statements on the server, it only sends the SQL select query to the driver and waits for the data to be delivered back from the database.

There are many statement prefixes which are used exclusively with the load and select statements as these two statements are practically the only ones that load data while providing some flexibility.

3.4.9 SQL statement

The `SQL` statement is very similar to the select statement except that the `SQL` statement can contain any SQL query, for example, an `INSERT` or an `UPDATE` statement. Everything between the `SQL` keyword and the next semicolon is sent to the driver and it is not evaluated by the application just like it is done in the case of the select statement.

Since the `SELECT` statement is an `SQL` statement, the following are equal:

```
SELECT * FROM t1;
SQL SELECT * FROM t1;
```

However, the `SQL` statement does not support all the prefixes that the `select` statement does and therefore it does not provide as much flexibility and customization which can be sometimes required by the user.

`Load, select` and `SQL` statements can be preceded with a name tag:

```
tag_name:
LOAD, SELECT or SQL statement
```

The *tag_name* sets the name of the new table in Qlik Sense. If no tag is set for the statement, the resulting table is either used by the preceding `load` statement or the table is dropped.

3.4.10 Store statement

This statement creates an explicitly named QVD, CSV, or TXT file where a table is stored. The statement can only export fields from one data table. If fields from several tables are to be exported, an explicit join must be made previously in the script to create the data table that should be exported [21]. Statement definition:

```
Store [ fieldlist from] table into filename [ format-spec ]
```

`Store` statement is mostly used in the data preparation Qlik Sense applications. Sometimes it is not desired to load all application data every time data loading is launched. It can either be because data sets loaded from some tables are too large and it takes too much time to load and transform or the data is no longer available in the database. For this purpose, a user usually creates several data preparation applications which can be updated independently and all their output is then aggregated in one presentation application which only loads the QVD, CSV or TXT files prepared by other Qlik Sense applications.

3.4.11 Rename statement

The `rename` statement changes the name of fields or tables provided. Two fields cannot be renamed to have the same name. Additionally, fields or tables can be renamed using a rename mapping.

A rename mapping is a table with two columns where the first column of each row defines the name of the old field or table which shall be renamed and the second column defines the new name of the field or table. It is not necessary to provide the source table name of the renamed field since all field names in Qlik Sense are unique.

3.4.12 Drop statement

The `drop` statement removes listed fields or tables from the script memory and these tables or fields are not available for further actions in the load script. Logically, these tables and fields are not present in the load script output either (and therefore cannot be used in expressions).

3.5 Required Features of the Qlik Sense Scanner

From the analysis above it is clear that to create a MANTA Flow scanner module which would analyze all data flows in Qlik Sense is beyond the scope of this thesis. For this reason, we consulted MANTA stakeholders to determine what features are essential for the minimum viable product (MVP), which is the product which only contains essential features that satisfy the first customers and provides feedback needed for further development.

The features which were agreed to be included in the MVP are:

- Metadata extraction from the Qlik Sense Desktop server

  The scanner module has to be able to connect to the local Qlik Sense Desktop and extract the metadata needed for metadata analysis and data flow generation.

- Metadata resolving

  Reconstruct the hierarchy of objects similar to the one used in Qlik Sense from the extracted metadata. These objects need to contain all information that will be needed in the data flow analysis.

- Data flow generation

  Implement analyzer that will analyze data flows and generate data flow graph of the following objects:

  - Qlik Sense internal tables and fields
  - Applications
  - Sheets
  - Visualizations distributed in Qlik Sense, release June 2019, except for the Button report item

  - Dimensions
  - Measures

- Data flow detail

  Analyze only direct data flows. Indirect data flows are not required in the MVP.

- Visualization Expressions

  Create a parser that will parse visualization expressions according to the syntax defined in Qlik Sense documentation. Implement a feature that will analyze parsed expressions and will list fields used in it.

- Load Script

  Create a parser which will parse the *Load Script*. Implement a feature that will analyze `LOAD, SELECT, SQL, STORE, DROP` and `RENAME` statements, partially also `CONNECT` and `DIRECTORY` statements, and will represent their behavior in the data lineage graph.

  Analysis of the following statement parts does not need to be implemented:

  - Statements where deduction would be used - values depending on data loaded from the database or where the asterisk (*) is used.
  - `Load` statement with the *`extension`* data source.
  - `Rename` statement where a mapping table is used.
  - Dollar expansion replacement or evaluating nested dollar expansions.
  - Evaluation of expressions and substitution of expressions with their results.
  - Prefixes of `LOAD, SELECT` and `MAP` statements.

- Integration with MANTA Flow

  Use Query Service, the MANTA Flow platform's tool for creating data lineage graphs of SQL queries, to connect Qlik Sense graph with database graph nodes and Node Creator, the MANTA Flow platform's tool for creating folder-like node structure in data lineage graphs, to connect it with file and folder graph nodes. The Query Service and Node Creator make it possible for MANTA Flow to connect the graph with the output of data flow analysis of other supported technologies used by the customer.

4. Implementation

We are going to describe main features, solutions to technical problems and the architecture of the scanner module in this chapter.

A typical scanner module used in the MANTA Flow platform consists of two main parts - a connector and a data flow generator.

The connector is responsible for connecting to the server of the technology analyzed, extracting metadata necessary for data flow analysis of defined parts of the technology and resolving, which is in our context the processing of the extracted metadata and creating a hierarchy of Java objects that accurately represent the objects that are going to be analyzed and only contain information necessary for data flow analysis.

This means, that, for example, if the Qlik dimension object's definition contains information about an internal value, such as coloring hash value, which is a purely static value with no possible table column references or other data flow options, it does not provide any useful information for our analysis and it can be ignored during the resolving phase.

On the other hand, if the dimension object's definition provides the expression used for defining the dimension, we would most definitely want to have this value analyzed as expressions usually refer to table columns or other data sources.

A data flow generator then uses a file reader, which is a part of the connector, and sequentially analyses each Qlik Sense application that was extracted. The file reader uses the resolver fro creating the Java-object-represented hierarchy of the Qlik Sense application and provides this to the data flow generator which analyses the objects.

The output of the analysis is a graph which can be combined with the graphs resulting from other scanner modules - a user may want to run an analysis of a database that provides data to Qlik Sense applications and the final MANTA visualization of the graph then displays an entire data flow chart from the database to Qlik Sense report items showing deep insights into what is actually going on.

Based on the above mentioned general structure of the majority of MANTA scanner modules we had to implement the following artefacts:

- Connector
  - Extractor
  - Model

    Defines the interface for all classes implemented in resolver.
  - Resolver
  - File Reader
- Data Flow Generator

4.1 Technologies used

Because our scanner module is supposed to work on the MANTA Flow platform with already set up environment, the choice of the technologies was relatively simple. To keep the dependency list as short as possible, we tried to use libraries already included in the platform and use technologies that were already used in other scanner modules (for configuration or parsing).

- Java 8

  Because most of the MANTA Flow code is already written in Java 8, it was logical to implement our solution in the same language. Thanks to this we were able to easily use already existing infrastructure provided by the platform.

- Spring

  Because MANTA Flow is a Spring application, it was necessary to perform all configuration of the scanner module in the same way as other modules. We used Spring's XML configuration, which is similar to other components, and additional configuration is done using the *.properties* files.

- ANTLR v3

  ANTLR was used for writing parsers for the Load Script and the visualization expressions. Version 3 was used because other projects use the same version as well.

- Maven

  Maven was used to manage the artifacts' dependencies as it is a very common tool suiting our needs sufficiently. Because other scanner modules also use Maven, we were able to solve some configuration problems by comparing our *pom.xml* files with other modules' files in case we encountered a problem.

- JUnit 4 and Mockito

  JUnit 4 was used for testing as it is a standard library when it comes to writing tests. For creating mock objects for tests when needed, we decided to use Mockito framework mainly because of its simple interface and previous experience.

- Java-WebSocket

  To communicate with the Qlik Sense server via web sockets we decided to use the Java-WebSocket library because it was already included in the MANTA Flow platform and it was one of very few Java web socket libraries with a convenient license that did not require publishing our code.

4.2 Extractor

The extractor's sole purpose is to extract Qlik Application information from the server and save it. For each extracted application on the server, a separate folder is created, where all data is saved into corresponding files. The application folder structure is the following:

- Root folder (application title)

- dimensions (each dimension in its own file named *dimensionID.json*)
- masterObjects (each master object in its own file named *masterObjectID.json*)
- measures (each measure in its own file named *measureID.json*)
- sheets (each sheet in its own file named *sheetID.json*)
- snapshots (each snapshot in its own file named *snapshotID.json*)
- stories (each story in its own file named *storyID.json*)
- *associations.json*
- *connections.json*
- *media.json*
- *tables.json*
- *variables.json*

It was necessary to create this structure as it is not possible to, for example, retrieve detailed information about all shared dimensions in one server response. It is also more convenient to have one file per one dimension (or other object). This way it is easier to work with it later in the resolver.

4.2.1 Extractor classes

Based on the content of Figure 4.1, we are going to describe what individual classes of the extractor do or represent.

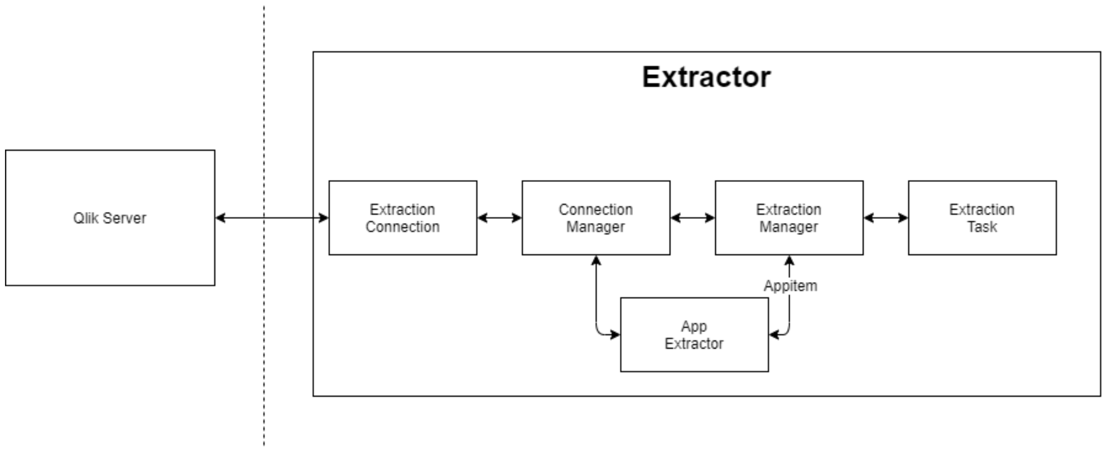


Figure 4.1: Extractor communication with a Qlik Sense server.

Extraction Task and Extraction Scenario

In order for MANTA to be easily extensible with new scanner modules, the platform uses its own scenarios and tasks. A scenario executes independent tasks and a task is a sequence of commands that returns certain output based on the input provided.

Extraction Manager

The Extraction Manager class is responsible for executing commands defined in the Extraction Task. It can either be retrieving a list of all applications located on the Qlik server or extracting a specific application from the server. It deals with the top-level issues such as server timeout expiry and it is the top-level class where all exception propagation ends.

When an application is extracted, this class is responsible for preparation of the folder structure for saving the extracted Qlik application metadata into.

App Extractor

Performs extraction of an application. Instructs the Connection Manager with the requests that are to be sent to the server, processes response messages and, if they are valid, saves them.

Connection manager

Manages the communication between the Qlik Server and the extractor. It also deals with latches used for response timeouts and checking if the responses coming from the server are valid or not.

Extraction Connection

The web socket client class which connects to the Qlik Server. It only sends and receives messages without any processing or checking.

App Item

A data container for exchanging Qlik-application-specifying information between the Extraction Manager and the App Extractor. The data fields it contains are:

- Application ID
- Application Title
- Path to the application on the server

4.2.2 Testing

Since testing of the extractor is heavily focused on dealing with various connection situations (several different errors and problems can occur), the tests usually rely on a successful connection to the server. If no connection is established and is required, these tests are skipped. In case the server runs, it needs to have 2 testing applications imported (files *TestApp1.qvf* and *Tutorial 1 App.qvf*, (Qlik-displayed name is *TestApp2*) found in test resources). Without this testing environment set up, testing is usually useless and most of tests are skipped.

4.3 Model

As mentioned above, the model artifact is only used to define the interface provided by the classes that are used in the resolver. Since the resolver classes are, after initialization, only data containers, their interfaces are usually only a set of value getters. In some special cases, some setters may be included as well.

It is very important for data flow analysis to have the model fitted precisely to the actual object structure in a Qlik application and therefore it is not possible to simply use one model object for all Qlik Sense objects. Some model objects have more properties, some are just containers for grouping children and some may even be a combination of these two.

Additionally, if we create a very well-tailored model, we can be very sure about what properties are present simply by checking the object type. This helps us to prevent Null Pointer Exception and allows us to skip checking for null or invalid values in many cases, making the code cleaner and less error-prone.

Qlik Sense's object model is fairly complicated and we are going to describe individual object groups in more detail, however, we are not going to describe each interface method as this would be unnecessary and we are will mention important properties in the resolver description later in this chapter, if needed.

4.3.1 Important interfaces

In this section, we are going to describe what important interfaces were designed to provide specific functionality needed for resolver.

QlikNode

As mentioned above, we are trying to create a hierarchy - a tree - which would start with the application node and would represent the structure of the application. Each node shall be uniquely identified by its name, type and parent node.

For this purpose, MANTA scanner modules use a base interface for all other object interfaces which defines these three properties (via methods *getID(), getType()* and *getParent()*. This way it is possible to set any other object in the hierarchy as the parent. In our project, we named this interface *QlikNode* and all object interfaces in the model implement it.

PropertyInterface and QlikProperty

As mentioned in the analysis part of this work, almost all Qlik Sense objects have got some properties which can be object-unique and which can be expressions, therefore making it possible to create data flows in the applications. In order to make it possible to store these expressions, we created the *PropertyInterface*, which provides the method to retrieve a list of object's properties.

Each property is represented by an object implementing the *QlikProperty* interface which can be seen as a standard std::pair<std::string, std::string>object known in C++. We could have used a Map<String, String>, however, it is more convenient to have one object passed around and work with it later in the data flow generator. *QlikProperty* interface also implements the *QlikNode* interface and can, therefore, be used as a child node if needed.

ColorDefinition

The coloring options of report items in Qlik Sense are a little bit more complicated than the properties we commonly work with, as we described in the previous chapter. To represent the possibility of the coloring be an expression, a library item (a shared dimension or a shared measure reference) or empty, we created a *QlikTriple* interface which, in addition to the key and value (in this case named *Expression label*), adds another retrievable value - an enum value of *QlikTripleType*, which can be the above-mentioned *EXPRESSION, LIBRARY_ITEM* or *EMPTY*.

Thanks to this we later know how to process the *Expression label* value.

ReportItem

Report items in Qlik Sense have several common properties and similar behavior. To avoid repetition in our code and to simplify the code, we created the `ReportItem` interface that lists the common methods that are to be available across all report item objects in our model. Additionally, it allows us to create more specific type definitions in containers.

ReferenceObject and ReferenceReportItem

It is very common to have objects referenced in Qlik Sense. Some references are purely ID-based and some references can even define some additional properties. To distinguish these references we have got two different objects.

The *ReferenceObject* implements, in addition to the *QlikNode* interface, the *PropertyInterface* and it can be used for references that define additional properties.

The *ReferenceReportItem* interface only stores the referenced report item's ID. It is, however, still kept as an object as it gives us, again, more flexibility in the future development in case Qlik Sense's interface changes and would make it possible to add more information to the referenced report items.

From the implementation point of view, the Model artifact is not particularly interesting mainly because no actual code is placed here except for the interface definition.

4.4 Resolver

After we analyzed what objects, relations and properties there are in Qlik Sense and determining how to find the information we need, we created a model that would provide an accessing interface for other artefacts for working with Java classes representing the Qlik Sense application structure.

With model being ready, we were able to work on the Resolver which implements all these interfaces. Resolving relies on the data extracted by the extractor and starts with the root node, *Application*, which is instantiated using the *QlikNode* values (ID, type and parent) and an additional constructor parameter - a File instance which points to the directory where the extracted application metadata is saved.

It goes over the metadata files and recursively instantiates objects which adequately represent the object. It is recursive in a way that if, for example, a sheet of an application contains several report items, a sheet's loading process instantiates the report items individually with parts of the JSON object relevant for the report item (since JSON sheet definitions contain also definitions of the report items as sub-trees). These report items can, for example, have definitions of hypercubes nested. These hypercubes are, again, instantiated with the sub-tree defining the hypercube and so on.

Thanks to this structure of metadata and the fact that we are able to work with arbitrary sub-trees of JSON objects, we are able to process only the information needed for the newly created object and all child objects take care of setting their own values themselves during their initialization.

Therefore, a typical object used in the resolver has got:

1. Private fields where individual values of the object are stored
2. A constructor with parameters *ID, type, parent* and a JSON object defining the object, optionally some special parameters
3. A private method for loading its own data which is called in the constructor
4. A private method for instantiating its own children (when needed)
5. Implemented all interface methods (usually getters) which do not change the state object (exceptions are made in some cases)

It is important to mention that it is very important to make sure that once the object is initialized, its state doesn't change. This is due to the fact that we desire that the output of the analysis is the same every time we launch resolving/data flow generating. This essentially only means that we avoid defining any setters and a field's value assignment is only possible during the object initialization (again, with some exceptions) and returning *unmodifiable* lists.

The resolver artifact is divided into several packages:

- *eu.profinit.manta.connector.qliksense.resolver*

  Classes directly in the source code root are the classes that are used on the application's top level. These classes are the ones that are either of general-purpose (*QlikNodeImpl, QlikPropertyImpl*) or they can be direct children of the Qlik application's object (*BookmarkImpl, DimensionImpl, MeasureImpl, SheetImpl, VariableImpl*). The application class AppImpl is also placed here.

- *eu.profinit.manta.connector.qliksense.resolver.datasource*

  This package contains classes that represent data sources for the application - databases, tables and fields from which Qlik Sense loads data and also classes representing the tables, fields and derived fields present in the application *after* the data-loading process is finished.

- *eu.profinit.manta.connector.qliksense.resolver.reportitem*

  Classes in the reportitem package are classes used for resolving report items. There are different classes for different report items so that it is possible to match the object as tightly to the actual object structure as possible. In the case of Qlik Sense, the one-size-fits-all approach cannot be applied. There are, however, some objects used for more than one report item. We describe them later in this section.

  All Report item classes inherit from ReportItemImpl abstract class.

  Apart from the report item items, some helping data structures are placed here as well - *QlikTripleImpl, ReferenceObjectImpl, MasterObjectImpl* (providing master-object-specific values), *ReferenceReportItemImpl* and *ReferenceLineImpl* because they are related to the visualizations and belong here.

- *eu.profinit.manta.connector.qliksense.resolver.reportitem.map*

  The MapObject class is very special as it usually contains layer objects that have a very complex structure with many optional parameters that can contain expressions. Because of this, there are many objects used when resolving maps and their layers and all these structures are placed into this package. We describe the map object structure later.

- *eu.profinit.manta.connector.qliksense.resolver.story*

  Classes representing the stories are placed here. They are not very complex or important. They form a *Story-Slide-SlideItems* hierarchy and since we are not going to analyze data flows in Stories in our project, we do not need to focus on them.

- *eu.profinit.manta.connector.qliksense.resolver.utils*

  Helper classes with purely static methods used for separation of specific tasks and making resolved classes more readable are here. *ReportItemFactory* and *SlideItemFactory* are placed here together with classes that help with navigation across JSONObjects or loading all relevant properties based on the visualization/object type.

4.4.1 Report Item classes

There are 6 classes representing standard Qlik Sense report items:

- *CalculatedObjectimpl*

  A group of simple report items using the hypercube for calculating data for displaying and visualization properties defined as expressions.

  Report item types: *gauge, KPI, pivot table, table, waterfall chart*

- *ColoredCalculatedObjectImpl*

  An extension of the *CalculatedObjectImpl*, can additionally store information about the measure- and dimension-based report item coloring in *QlikTripleImpl* objects.

Report item types: *bar chart, combo chart, distribution plot, line chart, pie chart, scatter plot, tree map.* Map report item's layers (class *MapLaterImpl*) also implement *ColorDefinition* interface that adds the coloring functionality.

- *AdvancedCalculatedObjectImpl*

  Another extension of the *CalculatedObjectImpl* which analyzes another hypercube defined in *box plot* and *histogram* JSON definitions. This hypercube defines special parts of the visualizations such as box plot whiskers and other scaling or customization elements. However, it was agreed with the MANTA stakeholders that the prototype does not have to analyze this additional hypercube yet. Therefore its analysis is not implemented in this work.

  Report item types: *box plot, histogram*

- *ContainerImpl*

  A special report item type that stores definitions of/references to its nested report items.

- *FilterPaneImpl*

  A class that stores *list box* items and shared dimension references.

- *TextImageImpl*

  Contains a hypercube and a background URL string if present. If this report item displays a calculated value as its text, it is stored as a measure in the hypercube.

A visual representation of the Report Item class hierarchy can be seen in Figure 4.2.

4.4.2 Map and Layers

Because Map's layer objects differ severely and they contain many different properties that can be set as expressions, in order to precisely capture them we had to create a class structure, which is quite difficult to understand. A Map is a standard report item object extending the abstract *ReportItemImpl* class with two additional fields - a list of 'normal' layers and a list of background layers.

The base class for 'normal' layers is the *MapLayerLocationImpl* class which is a container for the location definition of a layer. It can be:

- Based on latitude and longitude

  In this case, longitude and latitude fields are set as expressions, whose result specifies where the layer's points/charts shall be placed on the map.

- Based on administrative division

  Administrative division can be set using expressions in four fields - Location Field (a general expression that shall return location), Country and Administrative Area (twice - level 1 and 2).

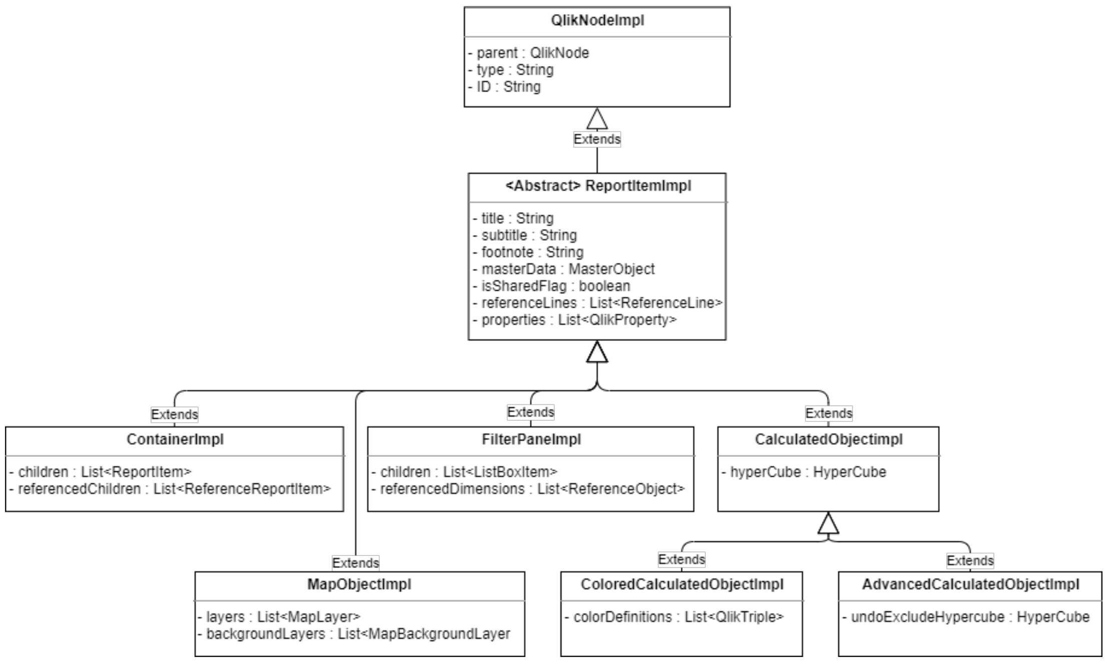

QlikNodeImpl
- parent : QlikNode
- type : String
- ID : String
Extends
<Abstract> ReportItemImpl
- title : String
- subtitle : String
- footnote : String
- masterData : MasterObject
- isSharedFlag : boolean
- referenceLines : List<ReferenceLine>
- properties : List<QlikProperty>
Extends
ContainerImpl
- children : List<ReportItem>
- referencedChildren : List<ReferenceReportItem>
Extends
FilterPaneImpl
- children : List<ListBoxItem>
- referencedDimensions : List<ReferenceObject>
Extends
CalculatedObjectimpl
- hyperCube : HyperCube
Extends
MapObjectImpl
- layers : List<MapLayer>
- backgroundLayers : List<MapBackgroundLayer
Extends
ColoredCalculatedObjectImpl
- colorDefinitions : List<QlikTriple>
Extends
AdvancedCalculatedObjectImpl
- undoExcludeHypercube : HyperCube

Figure 4.2: Inheritance relations between classes representing a standard report item objects of Qlik Sense.

Each 'normal' layer contains one location definition except for the line layer which contains two of these (for start and end points).

The *MapLayerImpl* class extends the *MapLayerLocationImpl* with properties, color definitions, a hypercube field for storing the hypercube, which is used for calculating for example pie charts at various points of the map, and other less important fields. The area layer is represented with this class.

The *MapCustomLayerImpl* class is the extending class of the *MapLayerImpl* class and adds visualization settings field which allows a user to set size or density settings which shall be used when applicable. Density, chart and point layers are represented by the class.

The most complicated is the *MapLineLayerImpl*, which extends the *MapCustomLayerImpl* class adding fields for defining the way how lines shall be calculated (defining the *Line geometry field* or by adding another location information for end points of the lines). Only the line layer is represented by this class.

The background layer is represented by the *MapBackgroundLayerImpl* class and contains property values for the 3 modes it can be in - TMS[1], WMS[2] or Image (Image URL). Each mode provides a different way of setting up the map background and different properties can be set, which is why there are three mode-based property groups. These groups are represented by the *BackgroundLayerPropertiesImpl* classes.

4.4.3 Testing

Tests are performed using resources containing JSON definitions of various objects - from small definitions of a simple dimension or a measure to large complex

[1]Tile Map Service
[2]Web Map Service

structures like sheet definitions with over 130 thousand characters.

What is usually tested is, that when an object is created using a definition, all properties load as they should. This means, that for example for a HyperCube, all dimensions and measures should be loaded, calculation conditions are omitted when empty, etc. For all the resolved objects, there is a special Test class (there are a few exceptions when tests were not needed, though).

However, when it comes to testing the property loading of objects, large test classes are created (*GeneralPropertyEmptyTest, GeneralPropertyExpressionTest* and *GeneralPropertyValueTest*). Since JSON structure varies depending on fixed-value/empty/expression contents of property fields, there are 3 main test classes.

1. Empty fields

   This class tests all built-in visualization types with only dimensions/measures set. No properties are modified, everything is default. The purpose of this test is to check if the property-loading does not fail on having some JSON items missing/empty strings.

2. Value-set fields

   Tests the property-loading of all properties that's value is a number/string, not an expression. Even though these values are useless in terms of data flow, they are (usually) saved the same way as expressions and we will only find out when parsing these values.

3. Expression-set fields

   Tests the property-loading for expression-set properties values. These are always strings, usually nested in qStringExpression/qValueExpression keys of the JSON object.

To successfully run tests, it is only needed to have all resource files present (tests/resources folder in resolver). All tests shall pass, none should be ignored nor failed.

4.5 File Reader

The input reader implements functionality defined by the Manta abstract file input reader. Collects all extracted applications and when requested, performs reading of the extracted metadata to form a tree structure replicating the object hierarchy of the read Qlik Sense application.

The *File Reader* is essential for the whole analysis because it is this artifact that provides the Java objects for analysis. Without it, we would have no mechanism for iterating over all extracted applications, resolving them and supplying our *Data Flow Generator* with. It can be said that it is the connecting link between all our other artifacts.

4.6 Data Flow Generator

The *Data Flow Generator* artifact is the part of our scanner module which produces the output which can be visualized. It analyzes the *Resolver* output and sequentially analyses all parts of the application to create as accurate data lineage graph as possible.

4.6.1 Overview

Once we have got the Connector ready and are able to get the application in the Java object structure from the File Reader, we can start analyzing the metadata and create the data lineage graph.

There are essentially two major implementation tasks in this part:

1. Creating parsers for parsing the Load Script and visualization expressions

   Parsers are needed for analysis of the semantic meaning of individual Load Script statements, which lets us understand how columns used in expressions were created (for example we can find out from which file a column was loaded, how it was renamed or derived from other columns), and for analyzing the visualization expressions. A visualization expression can contain Qlik Sense field references and we want to be able to identify these field names in an expression.

2. Implementing analyzers for individual objects

   When analyzing the Qlik Sense Java-represented objects, we need to ensure that all resolved fields are correctly input into the data lineage graph, forming accurate hierarchy and when a relation between two objects or between an object and a data field occurs, the data flow is correctly visualized. We need to pay close attention to the consistency of the analysis, so that two separate analyses output the same graph - this is useful for testing and especially for MANTA's comparing functionality. This functionality shows differences between two analyses (for example an original and a modified Qlik Sense application) and if we did not have consistent output, it could be impossible to capture all changes accurately.

4.6.2 Parsers

Overview

Since the Load Script allows the usage of visualization expressions in some of its statements, the visualization expression parser and the Load Script parser share most parts and they only differ in the 'entry point' of the parser and the Load Script parser additionally uses the *ElPrefixes* parser grammar for parsing prefixes of the *Load* and *Select* statements' prefixes.

While Load Script parser is used to parse the whole Load Script as a whole, the Expression parser can only be used for parsing a single expression. It can be, to a certain degree, said that the Expression parser is a sub-parser of the Load Script parser, since the Expression parser rules are a subset of the Load Script's rules.

There are some cases when the Qlik Sense's Load Script processor behaves rather strangely. For example, it sometimes allows a forgotten semicolon without throwing an exception. Other times, there are some cases when an expression evaluates to null/empty string during an invalid expression, but there is no exception thrown. Instead, the script/expression processor finishes the script with this 'invalid' value - a boolean expression $1 < 2 < 3$ returns no value/null, while expected the behavior would be to crash (more than one comparison operator) or to return -1 (Qlik Sense value for 'true').

For this case, there are some expression evaluation 'anomalies' which had to be taken into account, because even if a customer runs this semi-valid script, the parser should not crash and therefore the rules are defined in the parser a little bit loosely.

We used MANTA's *IMantaAstNode* interface for our nodes to follow the convention used in other company's parsers and to avoid inventing the wheel. All resulting nodes in the parsing tree are therefore implementing this interface and can be easily used. It was not necessary to add any additional functionality to the tree nodes as the methods provided by the interface were sufficient for us during the analysis.

Due to the number of rules needed to cover almost whole syntax (it is probably never going to be complete since documentation is not 100% up-to-date nor lists all corner cases), parsers are split into several shared files and the final structure can be seen in Figure 4.3.

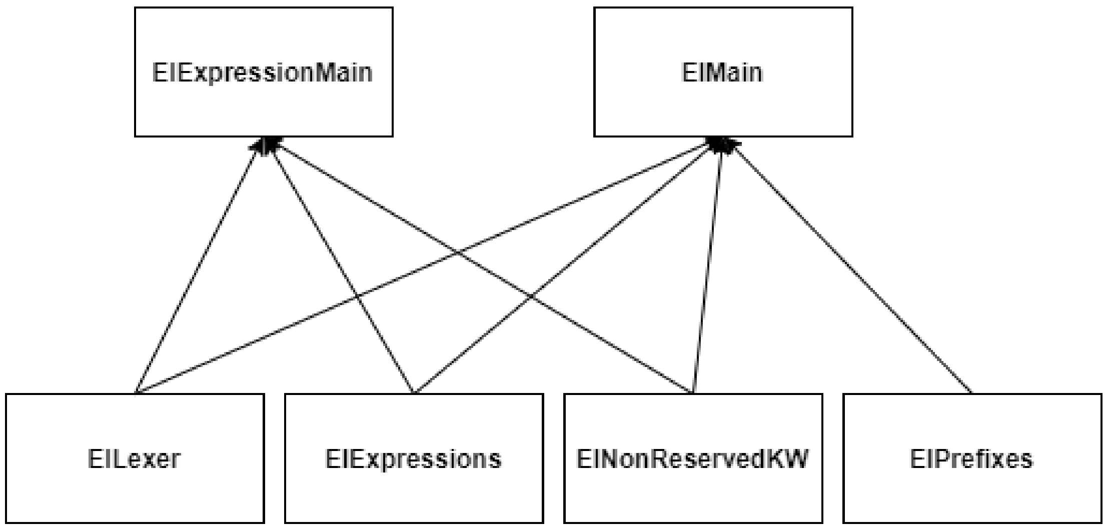


Figure 4.3: Grammars used (imported) in the Load Script parser (*ElMain*) and the Expression parser (*ElExpressionMain*).

4.6.3 Parser grammar files

In this subsection, we are going to briefly describe what can be found in each grammar file we created.

ElMain

- Serves as the main grammar for the Load Script parsing.
- Contains rules to recognize all documented control and regular statements.

- The DIRECT_QUERY statement has got a rather undocumented syntax and only elementary rules are implemented for this statement.
- Always returns the tree with the root node of type ElLexer.AST_SCRIPT.
- **INPUT**: A Load Script complying with the rules of the Qlik Sense Script Syntax.
- **OUTPUT**: An AST tree with the root of type ElLexer.AST_SCRIPT containing AST_BLOCK nodes with individual statement sub-trees.

ElLexer

- Defines:
  - AST tree node types.
  - Around 400 non-reserved keywords for functions used in Qlik Sense.
  - Character tokens such as bitwise shift (<<and >>) or curly brackets.
  - Fragments and token definitions for parsing various identifiers, quotes-enclosed strings or numbers as a single token.
  - Rules for ignoring single- and multi-line comments.
- Hashtag-containing function names (such as date# or money#) are not checked keywords since these keywords need to have KW_<name> as token name, but hashtag cannot be in the token name.
- Identifier token is only implemented for the ASCII characters accepted in Qlik Sense Load Script (which we mentioned in Section 3.4.3).
- Starts with characters:

  ```
  ( LETTER | '#' | '\$' | '%' | '?' | '@' | '\' | ']' | '^'
  | '_' | '|')
  ```

  and continues with start characters or:

  ```
  ( DIGIT | DOT )
  ```

- Recognises 4 types of strings: apostrophe-, bracket-, double-quotation- and grave-accent-enclosed.
- Apostrophe-enclosed strings have apostrophe escape character defined as ” (two apostrophes).
- Double-quotation-enclosed strings have quote escape character defined as ““ (two quotation marks).
- Brackets-enclosed strings have right bracket escape character defined as]] (two right brackets).
- These escape characters are then rewritten into a single character (implementation of the function in *ElNonReservedKW*).

ElNonReservedKW Defines:

- Rules mostly used in both *ElMain* and *ElExpressions* rules.
- Sets of non-reserved keywords to be used as identifiers in different contexts (eg. SELECT statement's *table* identifier cannot be the keyword *WHERE*).
- Rules for dollar-expansion and path parsing.
- Helper rules for distinguishing meaning in complex situations (statements `REM, SQL, SELECT` or conditions for parsing acceptable *table_identifier*).

ElPrefixes Defines rules for the prefixes used before LOAD/SELECT/MAP statements (not all prefixes can be used everywhere). These prefixes are not in the requirements of our scanner module, however, it is necessary to parse them to avoid parser recognition exceptions.

ElExpressions

- Contains rules for expression parsing.
- Unary/Binary operations with precedence.
- Function recognition.
- Aggregate function recognition.
- Always returns a tree with the root node of type ElLexer.AST_EXPRESSION.
- The last child of the root node is the original, complete expression represented as the text from the script - this is useful when providing the expression itself as an attribute in original formatting.

ElExpressionMain

- Imports *ElLexer, ElExpressions* and *ElNonReservedKW*.
- Has only one rule which directly returns the output of *ElExpressions* (therefore a tree with the root node of type ElLexer.AST_EXPRESSION).
- **INPUT**: A valid expression complying with the Qlik Sense expression syntax.
- **OUTPUT**: An AST tree with the root of type ElLexer.AST_EXPRESSION and sub-elements matching the expression. The last child node is of type AST_WHOLE_TEXT containing the original string representation of the expression to be parsed.

Load Script Parser's special cases

It would not make sense to comment on all parser rules defined for parsing the Load Script since they are largely written as an interpretation of the documentation with some minor changes when it turned out that the documentation was incorrect. Instead, we decided to point out three special cases when we had to create some workarounds.

Rule rem_select_statement: The statement REM is a commenting statement, where all text between the REM keyword and the next semicolon is understood as a comment (it is the same as putting the text in between the /* and */ marks. However, the SELECT and DECLARE statements support naming their output tables in the format:

```
<new_table_name>: SELECT/DECLARE statement
```

The *new_table_name* can, however, be value 'REM' or 'SELECT' and then a conflict occurred:

```
// a statement REM with content ' : some-useless-text'
REM : some-useless-text;

// creates a table 'REM' from the data returned by the SELECT
REM : SELECT * FROM <connection>;
```

We managed to solve this issue by looking farther ahead and checking what content is behind the colon to be able to correctly evaluate what is a REM statement and what is just a statement creating a table with an inconvenient name.

Rules sql_statement and select_block: Because SQL and SELECT statements are not evaluated by Qlik Sense, but instead by the ODBC/OLE DB/-Custom connection providers, we only decided to store the value of the whole statement into the AST node that represents the statement (in case of the SQL statement we removed the 'SQL' keyword).

We are going to use these statements' value in MANTA's Query Service (which analyzes SQL statements and returns data flow graphs) during the semantic analysis of the Load Script later.

Rule new_line: Is used to check for the new line character. This is used for control statements, since they can end with a new line instead of a semicolon/EOF and not checking the new line would result in a very ambiguous grammar.

The checking for a new line is performed by looking through the ignored characters (white spaces and new lines) and checking whether there is a new line. If there is no new line character, the rule is not fulfilled and some other control statement alternative needs to be used.

Expression Parser

The Expression parser is done based on the documentation and analysis. The expression syntax is at the first sight very well-described, however, there always can be some cases when the syntax exception can be undocumented or the syntax information may be lost/forgotten due to a huge scope of the expression syntax.

4.6.4 Expression Evaluator

Before we were able to analyze the data flows in the objects, we needed to find a way how to determine what fields are used in visualization expressions. The AST trees output from the Visualization expression parser can be very complex and the same node can have a different meaning in a different context (for example different string variants have different meaning in the LOAD statement and elsewhere, as mentioned in the previous chapter). To be able to crawl across the tree and get this data flow information from the expressions, we implemented a helper class named *ExpressionEvaluator*.

Interface

The *ExpressionEvaluator* class only contains a single public static method as it only has one purpose:

```
public static FlowContainer getFieldsUsed(IMantaAstNode, boolean)
```

The first parameter is the root of the expression tree, always has to be of type AST_EXPRESSION, since this is the tree root node type that is returned from the expression parser.

The second parameter is useful only in some cases when we have double quotes used - Qlik Sense sees differences in meaning when used in a LOAD statement and outside (plus SELECT statement, which is, however, evaluated by the OLEDB/ODBC driver, not Qlik Sense). If set to true, LOAD statement string interpretation is used, otherwise general interpretation is used.

The *FlowContainer* is a container of Set<String> that sorts data sources from the expression, based on the context of the expression, into one of five categories - direct/indirect field flows, variable references, include file paths (using dollar-expansion syntax) and bookmarks (can be both IDs and user-assigned names). Strings are returned in their normal form - without the quotation marks around them. Nodes are not looked up directly to leave space for the processing method to decide what to do with the data - sometimes it is not necessary to map Strings to Nodes, but perhaps only storing the information as context for some later analysis is sufficient.

Invalid input

When a null is input, an empty container is returned, however, this shall never happen.

Testing

The testing is performed by calling the public static method with different AST trees and asserting that the output (the *FlowContainer* object) contains the expected values and nothing else.

Graph Helper

In order to be able to work with the output graph during the analysis, MANTA scanners use scanner-specific *GraphHelper* classes which provide interface for working with the graph and making the individual analyzers simpler by grouping the common graph methods in this class.

All Graph Helpers extends MANTA's *AbstractGraphHelper* which provides methods for adding direct/indirect data flows into the graph or for returning the *Graph* object instance.

In our case, the *QlikSenseGraphHelper* class helps us with the following tasks:

- Mapping of shared dimension graph nodes by their ID (Dimension ID -> graph node)
- Mapping of shared measure graph nodes by their ID (Measure ID -> graph node)
- Mapping of shared report items (master objects) graph nodes by their ID (Report Item ID -> graph node)
- Mapping of loaded Qlik field graph nodes by their name (field name -> graph node)
- Providing the Application-representing and the Load-Script-representing nodes to analyzers
- Building nodes with various parameters
- Copying nodes merging other graphs into our graph (for example when we use MANTA's QueryService to analyze SQL statements of the Load Script)

4.6.5 Analyzers

Analyzers are the most important part of the Data Flow Generator since they are responsible for creating the graph which is the output of the whole analysis. Analyzers are required to be stateless so that they produce the same output every time they are provided the same data. Because of this, most analyzers only contain static methods and no instantiation is possible.

The entry point of each Qlik Sense application analysis is the *AppAnalyzer* which launches analyses of (in order):

1. The Load Script
2. Shared dimensions
3. Shared measures

4. Shared report items (master objects)
5. Sheets (with nested single-use report items, dimensions, measures etc.)

Graph Construction

The output graph is created sequentially, which means that at the beginning we start with an empty graph and each analyzer adds its own graph nodes (vertices) and edges according to its analysis output. The node-building, data-flow-adding, node-copying and node-mapping is, of course, done by the *GraphHelper* which we described earlier.

If we, for example, find out that a Report Item's *Dimension property* uses a field *ABC*, then we have to:

1. Check if there is already a node representing field ABC.
2. If it is not found, we have to create the node manually.
3. Build the node representing the Report Item (if it was not created before).
4. Build the node representing the Dimension and the *Dimension Definition* (see the end of this section where we describe the graph node structure which we use).
5. Create a new data flow from the field *ABC* to the Dimension Definition node.

By adding new nodes (vertices) and data flows (edges) we produce a data lineage graph which describes the data flows in the Qlik Sense application as accurately as possible. It is therefore very important to implement the analyzers correctly, so that the semantic meaning of each object and *Load Script* statement is analyzed and projected into the graph properly.

Script Analyzer

The *ScriptAnalyzer* is used for the analysis of how data is loaded into a Qlik Sense application. Unlike other reporting tools, which usually use 'live' data, Qlik Sense performs data loading on demand and saves the fetched data locally. This means that connection to the databases and other data sources is only performed when a user wants it to happen and another data transfer happens again on demand.

The analyzer iterates over the Load-Script-representing AST tree and analyses individual sub-trees representing the Load Script's statements. For each non-trivial statement (LOAD, SELECT, SQL, DROP, RENAME and STORE), there is a separate statement analyzer and the minor statements are analyzed directly in the *ScriptAnalyzer* class.

Each analysis is performed in a certain context - this context contains information about the state of Qlik Sense at a certain point (loaded/available tables and fields at the current point in the script, counters of 'unknown' tables used in the output graph node names, etc.). This context is passed around in the *AppContext* object, which serves as a large container for all this information.

The *ScriptAnalyzer* is the only instantiated analyzer because to analyze the SQL scripts we use the MANTA's Query Service, of which we need to keep the instance reference and also we use the MANTA's Node Creator, which helps us with creating a folder-like structure in the output graph. These two objects are then passed around as parameters to statement analyzers when needed.

The outcome of each statement analysis is projected into the GraphHelper's graph instance based on the information contained in the statement. Each statement analyzer, therefore, performs a semantic analysis of the statement using the statement's AST tree and the result of this analysis is projected in the output graph (new nodes and/or data flows) and the *AppContext* - for example by removing a table from the context when a DROP statement drops it.

It is important to note that since the requirements do not include a complete statement analysis, statement forms where deduction would be required (asterisk instead of listing fields for loading, using variables instead of parts of statements) are ignored. However, code includes some preparation for this analysis which can be implemented later.

QlikDataModel Analyzer

This is a 'Load Script post-processor'. It takes the output of the load script analysis (*ContextTable* objects in the *AppContext*) and compares them with the list of tables and fields that are supposed to be the outcome (in this section referenced as the final table and the final field) - this list contains all tables and fields that can be later referenced in expressions in report items and/or stories.

The analyzer iterates over the list of final tables and 2 options can happen:

1. There is a Context table with the same name
    - For each final field in the final table, the analyzer checks if the context table contains such field - if it does, this field becomes the source of the final field. Otherwise, a special Load Script's UNKNOWN_TABLE_SOURCE node is the source of the final field.
    - If there are some fields in the Context Table which are not matched by any final field, they are ignored - it is probably caused by a wrong deduction during the Load Script's semantic analysis.
2. There is NOT a Context table with the same name
    - For each final field, a new node is created with one of two sources:
        - UNKNOWN_TABLE_SOURCE - for tables which we don't know how they were created
        - SYSTEM_SOURCE - for fields that are Qlik-Sense-provided (such as $Field or $Table). These fields are generated 'statistics fields' and can be used in visualization expressions, but they are not loaded from anywhere, they are provided by the server.

After each final field's source-assignment, a derived fields analysis is launched, which iterates over all of the final field's *derivedFields* field list and creates a new derived field node in the same table with the final field as its source (a new direct data flow is added from the source field to the derived field).

ExpressionAnalyzer

The below-mentioned analyzers of the Qlik Sense objects often need to analyze visualization expressions in form of *QlikProperty*, *QlikTriple* or String objects. Since the interfaces they use are largely shared, we implemented a helping abstract class *ExpressionAnalyzer* which analyzes the objects mentioned earlier. This made it easier to analyze properties of the objects more easily and improved the code readability.

Dimension and Measure Analyzers

The implementation of these analyzers is fairly simple, these analyzers simply go through dimension/measure definitions (expressions) and create data flows starting from columns created by the *QlikDataModelAnalyzer* ending in the definitions themselves.

As we mentioned earlier, each dimension/measure can have some additional properties set. These properties are put into a single *Properties* node since those are usually not the data-providing properties (like definitions), but rather some visualization properties (expression-defined labels, expression-based coloring, . . .).

Names for dimensions and measures are generated in a specific manner to provide both readability and uniqueness. All elements on the Presentation layer are uniquely identified by their IDs, however, a few-letter-long alphanumeric string does not provide any information for the user. The node names are therefore created as a combination of a dimension's/measure's label expression or label (static string) and the measure's/dimension's ID. This way both readability and uniqueness are ensured.

If a dimension/measure is shared, it is put into a map in the *QlikSenseGraphHelper* for either shared measures or shared dimensions with the key being its ID and value is the graph node of the dimension/measure.

HyperCube Analyzer

The HyperCube Analyzer simply iterates over all its measures and dimensions and creates single-use dimensions/measures directly under the report item node or a map layer node (since a hypercube always belongs to a calculated report item or a map layer) and copies referenced shared dimensions/measures here (a direct flow is created from the shared dimension/measure to the copied one).

Sheet Analyzer

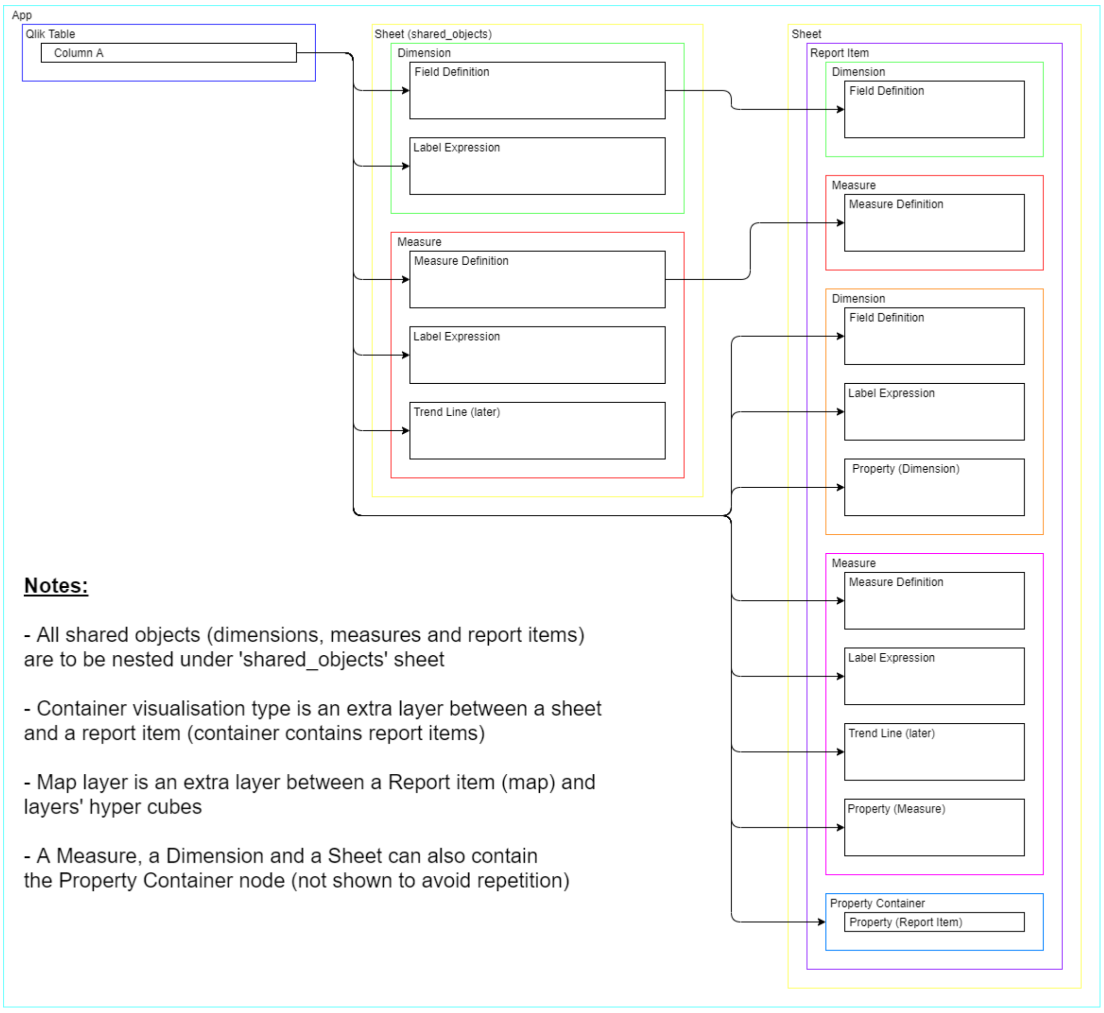
The Sheet Analyzer is basically just a container for report items and the analyzer does exactly the same thing. Creates the single-use report items in itself using the Report Item Analyzer (see below) and copies the whole shared report item nodes into itself.

A sheet has got two properties which can be expressions - label expression and description, this analyzer checks them both to see if they contain expressions and if yes, creates data flows for these properties.

The name is generated in a similar manner as in the case of dimensions/measures.

Report Item Analyzer

A report item is by far the most complex structure on the Presentation layer. The expected report item structure in relation to columns/shared objects is depicted in Figure 4.4.



<u>Notes:</u>

- All shared objects (dimensions, measures and report items) are to be nested under 'shared_objects' sheet

- Container visualisation type is an extra layer between a sheet and a report item (container contains report items)

- Map layer is an extra layer between a Report item (map) and layers' hyper cubes

- A Measure, a Dimension and a Sheet can also contain the Property Container node (not shown to avoid repetition)

Figure 4.4: Node structure of dimensions, measures, report items and sheets in the output graph.

Node names of report items are generated in a similar manner to dimensions, measures and sheets.

The common *ReportItemAnalyzer* analyzes the fields of the *ReportItemImpl* class (the common parent class for all report item objects), object-specific properties are analyzed separately. Because of this, report item analyzing is split into several parts:

Calculated Object A calculated object extends the report item with a *Hypercube* object. This is done by *HyperCubeAnalyzer* (see above).

In case a *CalculatedObject* is implementing the *ColorDefinition* interface, color definition analysis is performed. Each color definition can either use an expression or a shared dimension/measure. In the case of the latter, a given dimension/measure definition(s) only are flow-connected with the property, properties of the dimension/measure are omitted as they have no effect in coloring.

Container The analysis of the *Container* is fairly simple as the *Container* report item only forms an additional layer between a Sheet and a Report Item (container is grouping several report items). Therefore it only creates single-use report items as its child nodes and copies referenced report items as its children as well.

Filter Pane The *FilterPane* is a collection of *ListBox* items that's structure is very similar to those of dimensions and therefore the projection of a *Filter Pane* into the graph is very similar to *Container* (*ListBox* is in this relation in the position of a report item).

Map Object Since a map object is a rather large and complicated structure, it has got its own analyzer class - *MapReportItemAnalyzer*.

This analyzer is used exclusively by *ReportItemAnalyzer* class and it analyzes map's layers since *MapObject*'s inherited properties are analyzed in the *ReportItemAnalyzer* (a common method for all *ReportItemImpl*-extending objects) and *MapObject* only has two own fields - a list of layers and a list of background layers.

Layers have got a complicated inheritance structure, so their analysis is performed by analyzing predecessor classes from the base class through intermediate classes till derived (final) class (as described above when we talked about the Resolver). The code is not too easy to read due to the way Qlik Sense stores data - some expressions have got equals sign ('=') at the beginning of the expression to distinguish between an expression and a static text, some expression are saved without the sign at all.

In some cases (*QlikTriple* object), the key and the label values are the same - depending on the property, it can be impossible to change the label, which is then copied from the key, and the label analysis is skipped.

Text Image The *TextImage* analysis implementation is fairly simple since this class only has 2 fields - a hypercube and a background image relative path. Hypercube analysis is performed by the *HyperCubeAnalyzer* and path analysis is skipped since only Qlik Sense media images can be used as background images, URL or other forms of setting this is not possible. This may be analyzed in later versions of the scanner module.

Node structure

The node structure created in the implemented part of the generator is the following:

- Qlik Sense Application
  - Load Script
    - Load Script Statement
      - Load Script Statement Expression

  - Load Script Table
    - Load Script Field
- Table
  - Field
- Sheet
  - Report Item
    - Dimension
      - Dimension Definition
      - Property Container
        - Property
    - Measure
      - Measure Definition
      - Property Container
        - Property
    - Map Layer
      - Dimension
        - Dimension Definition
        - Property Container
          - Property
      - Measure
        - Measure Definition
        - Property Container
          - Property
      - Property Container
        - Property
    - Property Container
      - Property
  - Dimension
    - Dimension Definition
    - Property Container
      - Property
  - Measure
    - Measure Definition
    - Property Container
      - Property
  - Property Container
    - Property

4.7 Finishing Analysis

After all the analyzers finish generating the output data lineage graph, the data flow analysis is finished and the resulting graph is post-processed and saved by the MANTA Flow platform's server and our scanner module's task is finished.

Platform's *Viewer* or *Export* tools can then be used to see the visualized interactive graph or to create and download a dump file which allows users to share analysis outputs on different machines.

5. Case Study and Evaluation

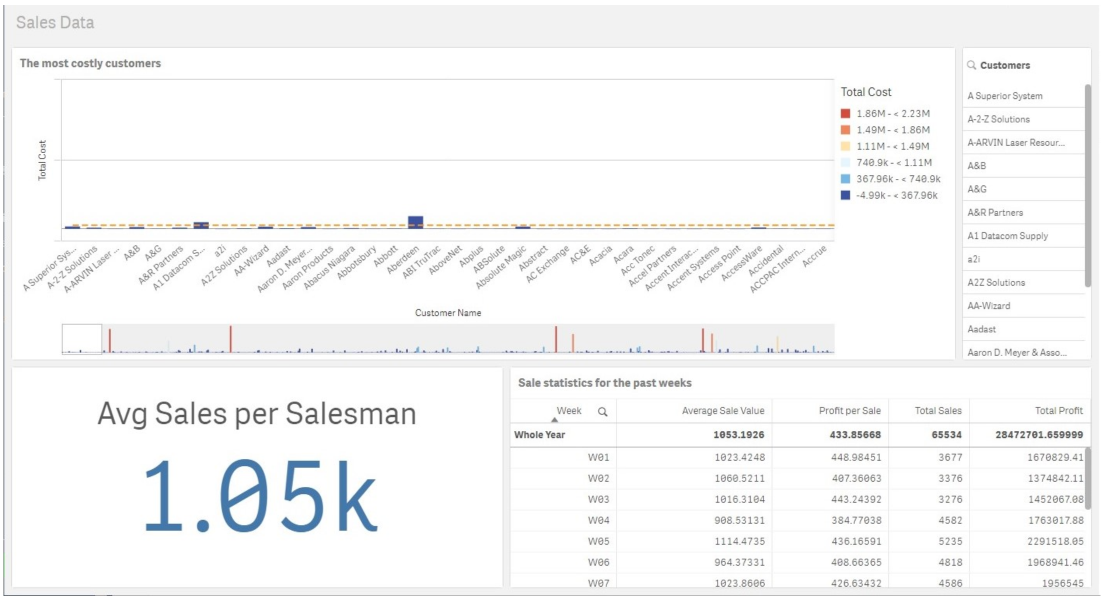
In this chapter, we would like to explain how our scanner module works if we run the extraction and data flow analysis on a real application placed on a Qlik Sense Desktop server. We will show you how the application looks like in Qlik Sense and how the data flow graph looks like when it is visualized in the MANTA Flow platform.

To demonstrate the functionality, we have created an application of medium size which loads some data from Excel Spreadsheet files and some data from database tables and uses some of these fields in its report items. We have created some single-use and some shared report items and dimension, so that the analysis for data flow generator is not trivial.

5.1 Example application

Our application for testing is called *my first app* and to fill it with data we used Qlik Sense sample data provided in a tutorial. Even though the data include actual companies, entries are not based on real-world data.

One of the sheets is shown in Figure 5.1. It contains several different report items - a bar chart, a table, a KPI[1] and a filter pane. The *Total Profit* measure, which can be seen as the last column of the table report item, is a shared measure and therefore when we will visualize it, it will be first shown in the sheet *Shared objects*, where we group shared objects, and then a direct flow will lead to its copy in the regular sheet.



Sale statistics for the past weeks

Week	Average Sale Value	Profit per Sale	Total Sales	Total Profit
Whole Year	**1053.1926**	**433.85668**	**65534**	**28472701.659999**
W01	1023.4248	448.98451	3677	1670829.41
W02	1060.5211	407.36063	3376	1374842.11
W03	1016.3104	443.24392	3276	1452067.08
W04	908.53131	384.77038	4582	1763017.88
W05	1114.4735	436.16591	5235	2291518.05
W06	964.37331	408.66365	4818	1968941.46
W07	1023.8606	426.63432	4586	1956545

Figure 5.1: A sheet of the demonstrated application visualized in Qlik Sense.

We have included different data sources in our application to demonstrate the capability of analyzing data flows from different data sources, such as the file

[1]Key Performance Indicator

system (Excel files), MS SQL and Oracle databases. To show how files are loaded into the application, we are going to show parts of the Load Script.

```
LIB CONNECT TO [Manta_Oracle];

[OL$]:
SELECT "OL_NAME",
   "SQL_TEXT",
   "TEXTLEN",
   "SIGNATURE",
   "HASH_VALUE",
   "HASH_VALUE2",
   "CATEGORY",
   "VERSION",
   "CREATOR",
   "TIMESTAMP",
   "FLAGS",
   "HINTCOUNT",
   "SPARE1",
   "SPARE2"
FROM "OUTLN"."OL$";

LIB CONNECT TO [OLE_DB_Manta_MS_SQL];

[numbers]:
SQL SELECT "n"
FROM "master"."dbo"."numbers";

[spt_monitor]:
LOAD
   [lastrun],
   [cpu_busy],
   [io_busy] AS [io_busy-Region Code],
   [idle],
   [pack_received],
   [pack_sent],
   [connections],
   [pack_errors],
   [total_read],
   [total_write],
   [total_errors];
SQL SELECT "lastrun",
   "cpu_busy",
   "io_busy",
   "idle",
   "pack_received",
   "pack_sent",
   "connections",
   "pack_errors",
   "total_read",
   "total_write",
   "total_errors"
FROM "master"."dbo"."spt_monitor";

[Sheet1]:
LOAD
   [Customer],
   [Customer Number],
   [Line of Business],
57     [Region],
58     [Region Code] AS [io_busy-Region Code]
59   FROM [lib://AttachedFiles/Customers.xlsx]
60  (ooxml, embedded labels, table is Sheet1);
61
62  [Sales data]:
63  LOAD
64     [# of Days Late],
65     [# of Days to Ship],
66     [BackOrder],
67     [Cost],
68     [Customer Number],
69     Date(Date#([Date], 'MM/DD/YYYY') ) AS [Date],
70     [GrossSales],
71     Date(Date#([Invoice Date], 'MM/DD/YYYY') ) AS [Invoice Date],
72     [Invoice Number],
73     [Item Desc],
74     [Item Number],
75     [Margin],
76     [Open Qty],
77     [OpenOrder],
78     [Order Number],
79     Date(Date#([Promised Delivery Date], 'MM/DD/YYYY') ) AS [Promised
           Delivery Date],
80     [Sales],
81     [Sales Qty],
82     [Sales Rep Number],
83     [SalesKey]
84   FROM [lib://AttachedFiles/Sales.xlsx]
85  (ooxml, embedded labels, table is [Sales data]);
```

The code snippet shows some important constructs of the Qlik's Script language which we described in the *Analysis* part of this work.

On lines 1-18, we connect to a previously defined database server. We named this connection *Manta_Oracle* and to this server, we send our SELECT SQL query. The data returned is then stored in the table *$OL* (line 3).

We proceed similarly on lines 20-50, which load 2 tables, *numbers* and *spt_-monitor*. The second statement (lines 26-50) loads the table using the succeeding LOAD statement, where the source table for the LOAD statement is the table that is the result of the SQL SELECT statement under it. This is one of the ways how fields can be renamed (using alias naming in the LOAD statement), the other being a common RENAME statement.

Lines 52-60 show how a table *Sheet1* can be loaded from a local Excel file, where *AttachedFiles* is a local file system connection (a path to a folder on the local system). The last statement on lines 62-85 shows how a field can be defined using an expression (line 79), where we modify the *Promised Delivery Date* column to match our desired date format.

5.2 Extraction

Now that we have some idea of how our example application looks, we can have a look at how does the extracted file structure look like. When we run extraction of *my first app* application, the following file hierarchy is created (with relevant

JSON objects stored in the files:

- my first app
  - bookmarks
  - dimensions
    - SpjeHpu.json
  - masterObjects
    - 87768ea5-bb38-4aa0-9610-477c4f0c69ad.json
  - measures
    - uYfRu.json
  - sheets
    - 56cdd0f9-6366-4090-891a-691f23cfc4a6.json
    - ff33d8cb-00e5-4fbd-a0d1-0dafee2de72f.json
  - snapshots
  - stories
  - associations.json
  - connections.json
  - defaultFolder.json
  - innerUserTables.json
  - loadScript.json
  - media.json
  - systemFields.json
  - tables.json
  - variables.json

In the extracted files, we can find:

- No saved bookmarks
- 1 shared dimension

  In the *SpjeHpu.json* file, we can find a JSON-formatted text which defines the dimension with the ID *SpjeHpu*. This dimension can then be used in visualizations without any need to define them again each time it is used. In the report item, this object is then only referenced by its ID.

- 1 shared report item (a master object)

  The file in the *masterObjects* directory defines an entire visualization (report item) with all its properties, dimensions and measures. This visualization definition can be then used in any sheet without any further configuration.

- 1 shared measure

  Its usage is practically the same as the shared dimension's.

- 2 sheet definitions

  We store each sheet definition in a separate file. This way it is very easy for us to distinguish what belongs to which sheet. Moreover, the *Qlik Engine API* only provides detailed information on one sheet at the time.

- No snapshots and stories

  Data analysis of the *Stories* functionality was not included in the requirements and therefore we did not add any stories into the applications for the sake of simplicity.

- associations.json

  A file that contains all field associations between tables that are set up by the user. We do not use this in our analysis, however, it may be useful later.

- connections.json

  A list of all connections within the Qlik Sense Desktop application.

- defaultFolder.json

  The default folder property. When a relative file address is used without using the *lib://connection-name/relative-path* format, the relative address is used starting in the default folder location. This can then help us when reconstructing the absolute path.

- innerUserTables.json

  A list of all tables and fields loaded by the *Load Script* from defined sources. Also contains a list of derived fields. This is particularly helpful when we are comparing the result of the *Load Script* analysis with what is the expected output. Thanks to this we are sure that the list of fields we are working with during the expression evaluation is complete.

- loadScript.json

  The complete *Load Script*.

- media.json

  A list of all media placed on the server. In Qlik Sense, all media (images or thumbnails) can only be used if they are included in the Qlik Sense media library. This file lists all media files and their paths. We do not use this information yet, but it may come handy later.

- systemFields.json

  Supplements the *innerUserTables.json* file - lists all tables and their fields. When compared to the *innerUserTables.json*, it does not list the derived fields, however, it lists all system tables and system fields, which may also be used in a visualization. If we combine these two files, we get a complete list of all loaded or system tables and fields and their derived fields.

- tables.json

  Provides detailed information about the origin of tables. This means that we know from which connection the data was loaded, whether it was from a file or a database etc.

- variables.json

  Lists the key-expression pairs of variables defined in the application.

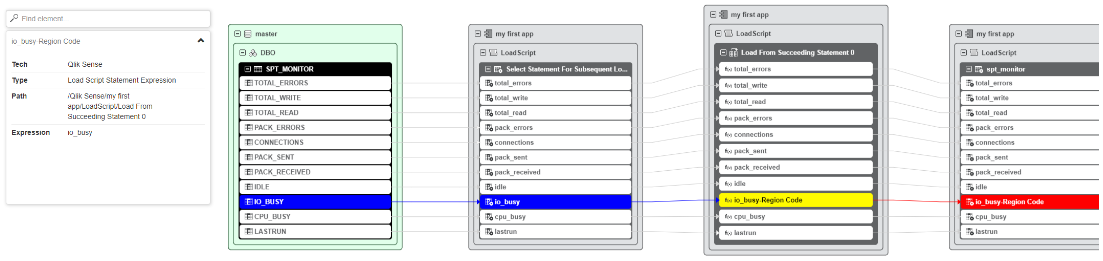
5.3 Analysis and visualization

Now that we have got the data extracted and know what data we have retrieved from the server, we can have a look at how is the data visualized. We are going to show small parts of the output graph because the whole graph spans over around 300 nodes and it would be impossible to read it in the printed version.

The figures used below are the actual output produced by our scanner module and visualized in the *Manta Flow Viewer*.

5.3.1 Loading data from a data source

To show how data loading with the load script is visualized in the graph, we have decided to show the part of the graph where the table *spt_monitor* is loaded. You can find the *Load Script* statements used for it on lines 26-50 of the script shown at the beginning of this chapter. The visualization of the data flow is shown in Figure 5.2.


Find element...

io_busy-Region Code

Tech	Qlik Sense
Type	Load Script Statement Expression
Path	/Qlik Sense/my first app/LoadScript/Load From Succeeding Statement 0
Expression	io_busy

Figure 5.2: Visualization of loading a table from a database.

As you can see in Figure 5.2, first we have a (MS SQL) database with a *DBO* schema and a *SPT_MONITOR* table. From this table, we load the columns using the SQL statement of the *Load Script*. Since this is a proper SQL query, we use MANTA's Query Service to analyze the query and create this part of the graph for us. We then connect it to the *Select Statement For Subsequent Load 0* table, which represents the table that results from the SQL query.

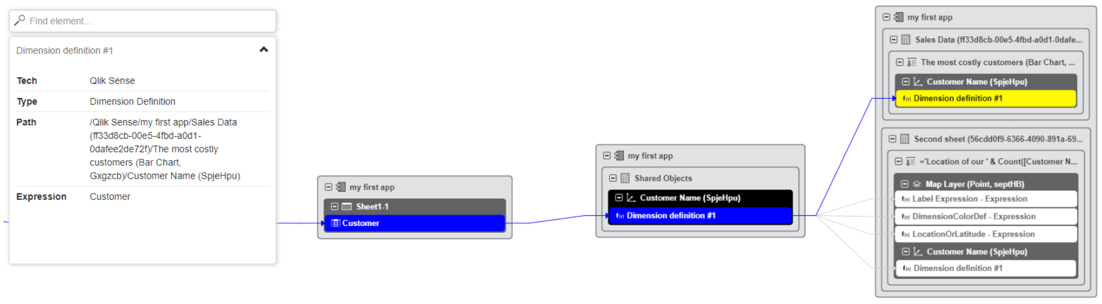
This SQL statement then provides the source data to the preceding LOAD statement (as you can see in the script). The output of the LOAD statement is shown in the graph node named *Load From Succeeding Statement 0*. This statement may have its expressions different from the output field names (see node *io_busy-Region Code*, that's expression is *io_busy*, effectively renaming the field using an alias in this statement).

Because the LOAD statement is preceded by a table name tag (*spt_monitor*), the output of the statement is to be stored into a table named so. At this point, the table is nested under the *Load Script* node, because it may be further modified in the *Load Script* and in the end, it may not even be loaded into the application at all. We will only find this information once the script's analysis is over and we compare our output with the expected output using the *QlikDataModel Analyzer* described in the previous chapter.

5.3.2 Shared dimension

Next, we are going to focus on how shared dimensions and/or measures are visualized and how it looks like when they are actually used in a report item.

We decided to show how the shared dimension *SpjeHpu* is visualized, as you can see in Figure 5.3.



Find element...

Dimension definition #1

Tech	Qlik Sense
Type	Dimension Definition
Path	/Qlik Sense/my first app/Sales Data (ff33d8cb-00e5-4fbd-a0d1-0dafee2de72f)/The most costly customers (Bar Chart, Gxgzcb)/Customer Name (SpjeHpu)
Expression	Customer

my first app
Sheet1-1
Customer
my first app
Shared Objects
Customer Name (SpjeHpu)
Dimension definition #1
my first app
Sales Data (ff33d8cb-00e5-4fbd-a0d1-0dafe...
The most costly customers (Bar Chart, ...
Customer Name (SpjeHpu)
Dimension definition #1
Second sheet (56cdd0f9-6366-4090-891a-69...
='Location of our ' & Count([Customer N...
Map Layer (Point, septHB)
Label Expression - Expression
DimensionColorDef - Expression
LocationOrLatitude - Expression
Customer Name (SpjeHpu)
Dimension definition #1

Figure 5.3: Visualization of a shared dimension *SpjeHpu* and its usage in a report item.

The dimension definition only contains one field-using definition, the *Dimension definition #1*. This shared dimension is used twice, the first time in the bar chart report item (top usage) and the second time in the *Second sheet* sheet, in a map report item's layer. In all these five places, the dimension is referenced by its ID.

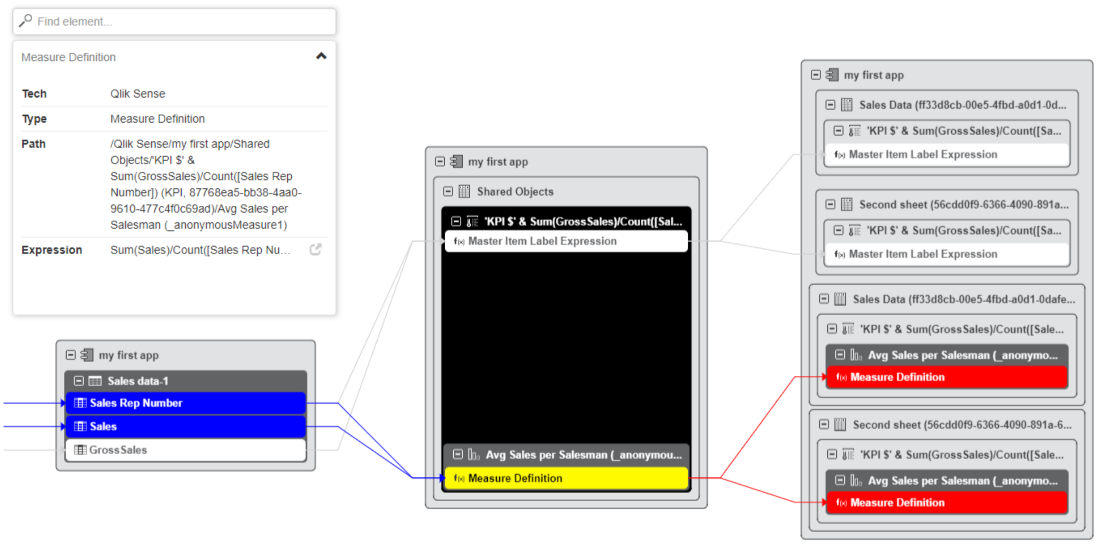
When it is used in a property expression (*Map Layer* properties), it is visualized as a simple data flow with no more context provided (as there is not really anything to say). When a dimension is referenced in a hypercube, we copy the dimension definition nodes (in this case only one definition) and generate data flows from the original node (which belongs to the shared dimension) to its copy in the *Map Layer*.

This way the user knows that the shared dimension is used in these places and if the shared object changes, all of these shared-object-using objects will change as well.

5.3.3 Shared report item

As we have mentioned in the analysis, a shared dimension or a measure does not have its other properties fixed and that is why we only copy the dimension/measure definitions when they are referenced elsewhere.

However, when it comes to the report items, no property can be changed additionally. Because of this, we have to copy all properties of the object (all report item child nodes). As you can see in Figure 5.4, we have got a shared report item, a KPI, which only has a measure defined using an expression, which is used in both *Sales Data* and *Second sheet* sheets. Because of this, the KPI report item is completely copied into both sheets and data flows are generated from the source to the copied nodes.


Find element...
Measure Definition

Tech	Qlik Sense
Type	Measure Definition
Path	/Qlik Sense/my first app/Shared Objects/'KPI $' & Sum(GrossSales)/Count([Sales Rep Number]) (KPI, 87768ea5-bb38-4aa0-9610-477c4f0c69ad)/Avg Sales per Salesman (_anonymousMeasure1)
Expression	Sum(Sales)/Count([Sales Rep Nu...

my first app
Sales data-1
Sales Rep Number
Sales
GrossSales
my first app
Shared Objects
'KPI $' & Sum(GrossSales)/Count([Sal...
Master Item Label Expression
Avg Sales per Salesman (_anonymou...
Measure Definition
my first app
Sales Data (ff33d8cb-00e5-4fbd-a0d1-0d...
'KPI $' & Sum(GrossSales)/Count([Sa...
Master Item Label Expression
Second sheet (56cdd0f9-6366-4090-891a...
'KPI $' & Sum(GrossSales)/Count([Sa...
Master Item Label Expression
Sales Data (ff33d8cb-00e5-4fbd-a0d1-0dafe...
'KPI $' & Sum(GrossSales)/Count([Sale...
Avg Sales per Salesman (_anonymo...
Measure Definition
Second sheet (56cdd0f9-6366-4090-891a-6...
'KPI $' & Sum(GrossSales)/Count([Sale...
Avg Sales per Salesman (_anonymo...
Measure Definition

Figure 5.4: Visualization of a shared Report Item used in both application's sheets.

If we visualized it this way, it is clear that the report items in the sheets are dependant on the shared report item and if anyone changes something in the shared objects, changes shall be expected in both sheets.

Additionally, Figure 5.4 also shows how our *Expression Evaluator* works properly. It correctly evaluated that the measure definition expression *Sum(Sales)/ Count([Sales Rep Number])* uses both *Sales* and *Sales Rep Number* fields and created data flows from these two fields into the definition node.

5.3.4 Sheet visualization

The last demonstration we are going to show is the complete Sheet object visualization.

We have decided to use the *Sales Data* sheet, which we already showed in the beginning of this chapter.

We have got four report items in the sheet - a bar chart, a KPI, a filter pane and a table. In Figure 5.5, you can see that nodes representing these report items are present in the graph, nested under the correct sheet. Only properties that

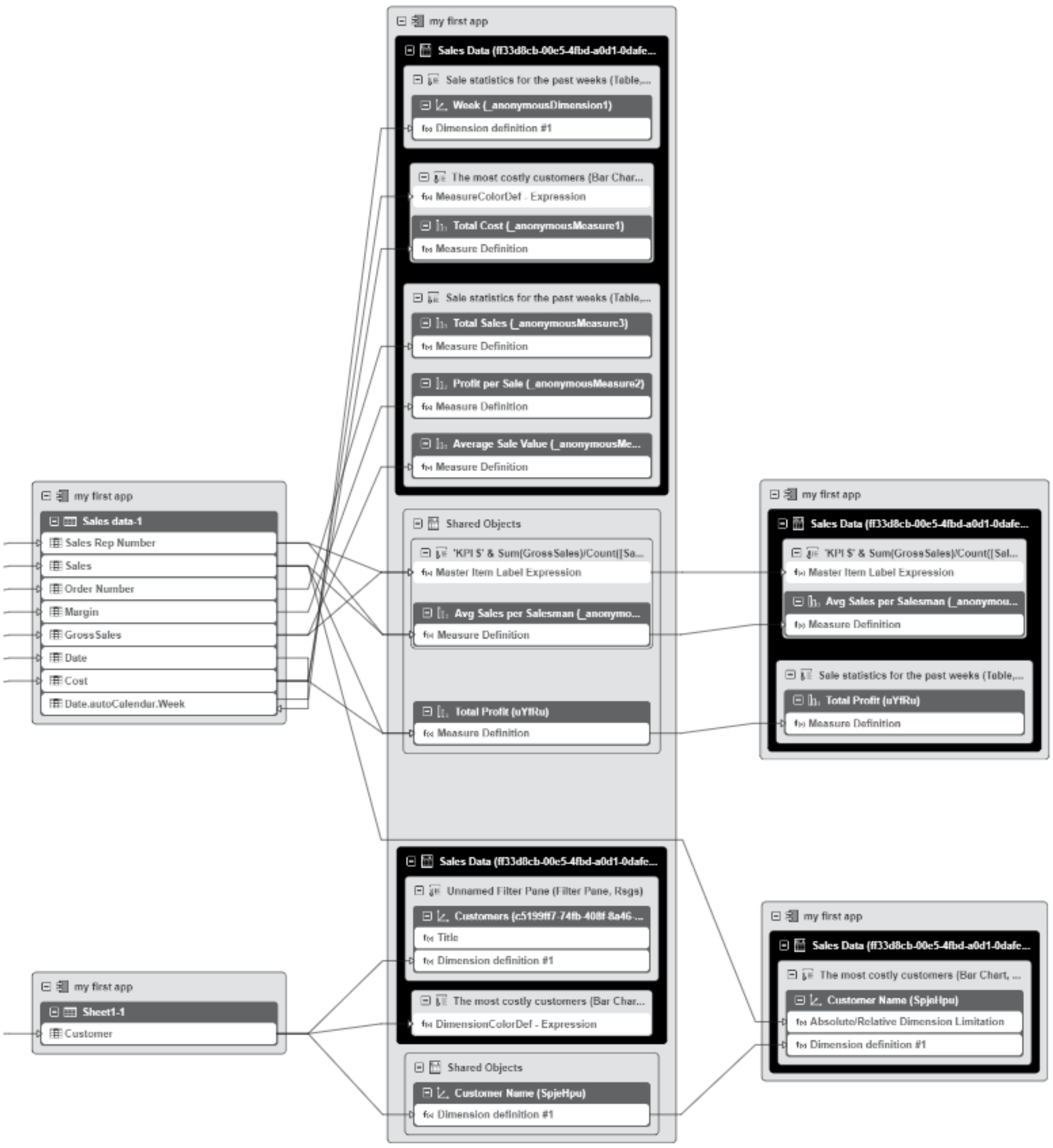

my first app
Sales Data (ff33d8cb-00e5-4fbd-a0d1-0dafe...
Sale statistics for the past weeks (Table,...
Week (_anonymousDimension1)
Dimension definition #1
The most costly customers (Bar Char...
MeasureColorDef - Expression
Total Cost (_anonymousMeasure1)
Measure Definition
Sale statistics for the past weeks (Table,...
Total Sales (_anonymousMeasure3)
Measure Definition
Profit per Sale (_anonymousMeasure2)
Measure Definition
Average Sale Value (_anonymousMe...
Measure Definition
Shared Objects
'KPI $' & Sum(GrossSales)/Count([Sa...
Master Item Label Expression
Avg Sales per Salesman (_anonymo...
Measure Definition
Total Profit (uYfRu)
Measure Definition
my first app
Sales data-1
Sales Rep Number
Sales
Order Number
Margin
GrossSales
Date
Cost
Date.autoCalendar.Week
my first app
Sales Data (ff33d8cb-00e5-4fbd-a0d1-0dafe...
'KPI $' & Sum(GrossSales)/Count([Sal...
Master Item Label Expression
Avg Sales per Salesman (_anonymou...
Measure Definition
Sale statistics for the past weeks (Table,...
Total Profit (uYfRu)
Measure Definition
Sales Data (ff33d8cb-00e5-4fbd-a0d1-0dafe...
Unnamed Filter Pane (Filter Pane, Rsgs)
Customers (c5199ff7-74fb-408f-8a46-...
Title
Dimension definition #1
The most costly customers (Bar Char...
DimensionColorDef - Expression
Shared Objects
Customer Name (SpjeHpu)
Dimension definition #1
my first app
Sheet1-1
Customer
my first app
Sales Data (ff33d8cb-00e5-4fbd-a0d1-0dafe...
The most costly customers (Bar Chart, ...
Customer Name (SpjeHpu)
Absolute/Relative Dimension Limitation
Dimension definition #1

Figure 5.5: Visualization of a sheet of our sample Qlik Sense application.

use a valid expression and contain a field usage are displayed, as other properties are irrelevant in terms of data flows.

We can see that the above mentioned shared KPI is displayed here, including its origin among the *Shared Objects*.

This visualization for example tells the user that if he or she removed all tables but *Sales Data-1* and *Sheet1-1*, nothing would change in this sheet. This could be for example useful when redundant data is present in the application file. Since Qlik Sense stores all the data locally, its size can be very big and take up a lot of disk space. If unused tables or fields were deleted, space-saving could be achieved without losing any functionality.

5.4 Space for improvement

Even though the prototype of the Qlik Sense scanner module works as expected and meets the requirements defined, there is still some space for improvement and some bugs or imperfections may occur. We list here some of the issues we are well-aware of and which we would like to fix in the nearest future.

5.4.1 Naming conventions

During the development, we have been using the property names as defined in the JSON objects. The keys used in theses objects were very often quite cryptic and not very useful for a regular customer who has never seen the Qlik Sense metadata.

As a result, there are still some properties named not very nicely, such as *DimensionColorDef* (as seen in the bottom of Figure 5.4), even though most items have already gotten much nicer titles.

Additionally, objects have usually unique, but strange names, which are a combination of a title expression and its ID, so a report item title *'KPI $' & Sum(GrossSales)/Count([Sales Rep Number]) (KPI, 87768ea5-bb38-4aa0-9610-477c4f0c69ad)* can be hard to read, even though it is necessary to ensure the uniqueness of node names in the graph.

It is, naturally, our goal to produce a graph which is easily comprehensible and this improvement would help users with navigation in the data flow graph greatly. Additionally, it would be way easier for users to match graph nodes with objects they see in their Qlik Sense application if the graph node name displayed the same string as the Qlik Sense object it represents.

5.4.2 Redundant nodes

During our analysis, there are some cases when we have to create some nodes in the data flow graph before we know if the node is going to be used for data lineage or not. As a result of this, sometimes we create nodes without any data flows (flowing in or out). These nodes should not, obviously, be present in the output graph and shall be removed.

One example is the *Title* node of the filter pane report item as seen in Figure 5.4. Another example can be seen in Figure 5.6. We create *Properties* nodes for storing properties into, but some objects do not have any data-flow-generating properties and as a result, these nodes are created and never used.

Luckily, the MANTA Flow platform contains a mechanism for filtering such nodes and it is possible to easily remove them.

5.4.3 Parsing incompleteness

Even though our parsers contain around 100 complex rules, there are always cases when it is not possible to parse the input and a parsing exception is thrown. Because the syntax is so complex and the documentation is not always complete and up-to-date, there can always be some cases when a parser comes across an unexpected token and crashes.

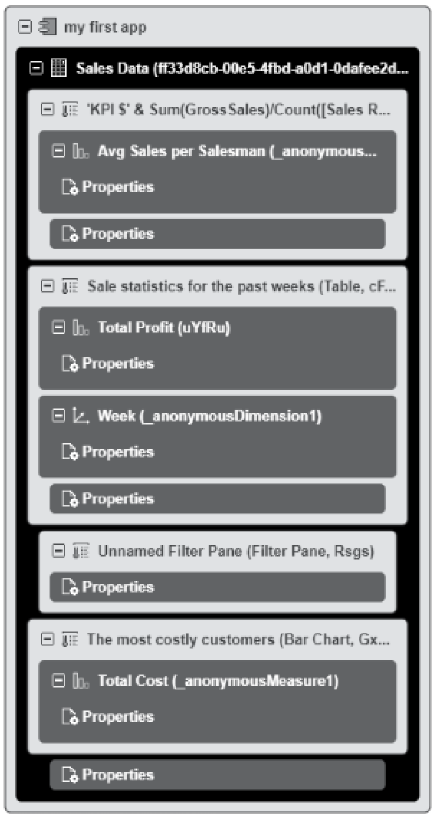

my first app
Sales Data (ff33d8cb-00e5-4fbd-a0d1-0dafee2d...
'KPI $' & Sum(GrossSales)/Count([Sales R...
Avg Sales per Salesman (_anonymous...
Properties
Properties
Sale statistics for the past weeks (Table, cF...
Total Profit (uYfRu)
Properties
Week (_anonymousDimension1)
Properties
Properties
Unnamed Filter Pane (Filter Pane, Rsgs)
Properties
The most costly customers (Bar Chart, Gx...
Total Cost (_anonymousMeasure1)
Properties
Properties

Figure 5.6: A part of the output graph with redundant orphan nodes.

There is, unfortunately, nothing we can do to avoid this except for extensive testing of all *Load Scripts* we can find and checking that the parsers work properly no matter what the input is.

5.4.4 Color scheme

Coloring plays a vital role for the user when trying to distinguish between different technologies in the MANTA Flow Viewer. Since one graph can contain several technologies (such as the *green* MS SQL database nodes in Figure 5.2 combined with default *black-gray* Qlik Sense nodes), it is important to make it as easy as possible for the user to navigate in often huge graphs.

For this purpose, our goal is to configure a coloring scheme that would be Qlik-Sense-exclusive. An obvious choice would be a shade of green similar to the one used in Qlik's logo.

5.4.5 Extraction from Desktop distribution only

At the moment, the extraction is only possible from a local Qlik Sense server (Qlik Sense Desktop distribution). However, in the commercial sphere, it is more common to use the Qlik Sense Enterprise distribution, which requires advanced authorization using either proxies or certificates.

In order for the customers to have the scanner module as useful as possible, it is necessary to have this functionality implemented before the module is released.

Even though the extracted metadata is not going to be any different than the metadata extracted from the Qlik Sense Desktop server, a different connection behavior is to be dealt with and advanced connection configuration is going to be necessary from the user.

6. Conclusion

In this project, we have developed the Qlik Sense scanner module prototype in our work, which extracts and analyzes metadata saved on a Qlik Sense server and produces a graph which visualizes data flows in the application. This module is integrated in the MANTA Flow platform and can be used together with other technology scanners already used by the platform.

To develop this module, we had to analyze the Qlik Sense thoroughly, finding out how data is structured, what objects and relations there are between them and how Qlik Sense works with all of this. Additionally, we had to find a way how to retrieve the metadata since APIs do not naturally provide the metadata in a common way and we had to retrieve it using many small server requests.

Our Data Flow Generator is now capable of analyzing some of the most important statements used in the *Load Script*, a script which is used for defining how data shall be loaded into the application, and generate a graph which very accurately visualizes what these statements do.

We have also managed to create parsers which are capable of parsing the *Load Script* and all visualization expressions used in the applications and thanks to this our Qlik Sense scanner module is able to analyze all expressions and table column references. This helps the scanner with detecting data flows mostly on the presentation layer and makes it a tool which can help Qlik Sense developers greatly when creating, modifying or optimizing Qlik Sense applications.

As we have shown in the last chapter, our scanner visualizes the data flows in such manner that all column (field) usage or transformation is clearly indicated by the graph and it is easy to see what Qlik Sense object we are visualizing and working with.

Bibliography

[1] *Business Intelligence Market Insights for 2020: What Do They Mean for You?* URL: https://www.selecthub.com/business-intelligence/business-intelligence-software-market-growing/. (accessed: May 04, 2020).

[2] *Creating a variable.* URL: https://help.qlik.com/en-US/sense/April2020/Subsystems/Hub/Content/Sense_Hub/Variables/create-variable-using-dialog.htm. (accessed: May 19, 2020).

[3] *Customer Success Stories.* URL: https://www.qlik.com/us/solutions/customers/customer-stories. (accessed: May 10, 2020).

[4] *Dimensions.* URL: https://help.qlik.com/en-US/sense/April2020/Subsystems/Hub/Content/Sense_Hub/Dimensions/dimensions.htm. (accessed: May 18, 2020).

[5] *Engines.* URL: https://help.qlik.com/en-US/sense/June2019/Subsystems/ManagementConsole/Content/Sense_QMC/engines-overview.htm. (accessed: May 16, 2020).

[6] *Exploring data lineage: Get a complete picture of your data flows.* URL: https://www.ibm.com/developerworks/data/library/techarticle/dm-1001datalineageinfosphereworkbench/. (accessed: July 21, 2020).

[7] *Functions in scripts and chart expressions.* URL: https://help.qlik.com/en-US/sense/June2020/Subsystems/Hub/Content/Sense_Hub/Scripting/functions-in-scripts-chart-expressions.htm. (accessed: July 21, 2020).

[8] *How It All Started: Patient Zero of Manta Tools.* URL: https://getmanta.com/blog/how-it-all-started-patient-zero-of-manta-tools/. (accessed: May 11, 2020).

[9] *How to Drive Data Literacy in the Enterprise.* URL: https://www.qlik.com/us/bi/-/media/08F37D711A58406E83BA8418EB1D58C9.ashx?ga-link=datlitreport_resource-library. (accessed: May 04, 2020).

[10] *Hypercube.* URL: https://help.qlik.com/en-US/sense-developer/April2020/Subsystems/Platform/Content/Sense_PlatformOverview/Concepts/Hypercubes.htm. (accessed: May 18, 2020).

[11] *Introducing Qlik Sense Enterprise.* URL: https://help.qlik.com/en-US/sense-admin/April2020/Subsystems/DeployAdministerQSE/Content/Sense_DeployAdminister/Common/qse-introduction.htm. (accessed: May 10, 2020).

[12] *JSON-RPC 2.0 Specification.* URL: https://www.jsonrpc.org/specification. (accessed: May 16, 2020).

[13] *MANTA Cases #3: (Not Always) Agile Development.* URL: https://getmanta.com/blog/manta-cases-3-not-always-agile-development/. (accessed: May 11, 2020).

[14] *Measures.* URL: https://help.qlik.com/en-US/sense/April2020/Subsystems/Hub/Content/Sense_Hub/Measures/measures.htm. (accessed: May 18, 2020).

[15] *Operators.* URL: https://help.qlik.com/en-US/sense/November2019/Subsystems/Hub/Content/Sense_Hub/Scripting/Operators/operators.htm. (accessed: May 21, 2020).

[16] *Proxies.* URL: https://help.qlik.com/en-US/sense/June2019/Subsystems/ManagementConsole/Content/Sense_QMC/proxies-overview.htm. (accessed: May 16, 2020).

[17] *Qlik Engine API.* URL: https://help.qlik.com/en-US/sense-developer/2.0/Subsystems/EngineAPI/Content/introducing-engine-API.htm. (accessed: May 16, 2020).

[18] *Qlik Engine JSON API.* URL: https://help.qlik.com/en-US/sense-developer/April2020/Subsystems/EngineAPI/Content/Sense_EngineAPI/introducing-engine-API.htm. (accessed: May 16, 2020).

[19] *Qlik Engine JSON API reference.* URL: https://help.qlik.com/en-US/sense-developer/November2019/APIs/EngineAPI/index.html. (accessed: May 18, 2020).

[20] *Qlik Sense APIs and SDKs.* URL: https://help.qlik.com/en-US/sense-developer/2.0/Content/APIs-and-SDKs.htm. (accessed: May 16, 2020).

[21] *Script syntax.* URL: https://help.qlik.com/en-US/sense/November2019/Subsystems/Hub/Content/Sense_Hub/Scripting/script-syntax.htm. (accessed: May 21, 2020).

[22] *Snapshot.* URL: https://help.qlik.com/en-US/sense-developer/April2020/Subsystems/Platform/Content/Sense_PlatformOverview/Concepts/Snapshot.htm. (accessed: May 19, 2020).

[23] *Supported Technologies.* URL: https://getmanta.com/technologies/data-integration/. (accessed: May 10, 2020).

[24] *Understanding script syntax and data structures.* URL: https://help.qlik.com/en-US/sense/April2020/Subsystems/Hub/Content/Sense_Hub/LoadData/understand-data-structures.htm. (accessed: May 21, 2020).

[25] *Visualization expressions.* URL: https://help.qlik.com/en-US/sense/November2019/Subsystems/Hub/Content/Sense_Hub/ChartFunctions/visualization-expressions.htm. (accessed: May 21, 2020).

List of Figures

2.1 Visualization of data flows of a Qlik Sense application. 8
2.2 An application overview user interface of the Qlik Sense Enterprise client. 9
2.3 A script-loaded table visualized in Qlik Sense's *Data model viewer*. 10
2.4 Visualization of a sample sheet in Qlik Sense. 11
2.5 An example of how Qlik Sense user interface for setting visualization's dimensions and measures looks. 12
2.6 An example of how a data lineage graph can look [6]. 15
2.7 A data lineage visualization example of data flows of several technologies created by the MANTA Flow platform (green - MS SQL, orange - Oracle, blue - Informatica PowerCenter, gray - SSAS). . . 18
2.8 The MANTA Flow platform's *Viewer* interface. 19

3.1 APIs that can be used for communication between individual Qlik Sense services in case of a Qlik Sense server installation. 21
3.2 APIs that can be used for communication between individual Qlik Sense services in case of a Qlik Sense local installation [17]. 21
3.3 Diagram of data flows on the Data Layer. 26
3.4 Dimension, Measures and some of their properties in a chart. 1 - Dimension label, 2 - Dimension field labels, 3 - Two measure labels, the first one being measure expression (when no explicit label or label expression is provided), the other being *saved name* property of a *reused* measure. 29
3.5 A standard bar chart used showing average sale per customer per city. 34
3.6 A filter pane with two dimensions - City and Customer. 34
3.7 Resulting bar chart after filter pane was used. 35
3.8 A problem of field origin. 46

4.1 Extractor communication with a Qlik Sense server. 56
4.2 Inheritance relations between classes representing a standard report item objects of Qlik Sense. 63
4.3 Grammars used (imported) in the Load Script parser (*ElMain*) and the Expression parser (*ElExpressionMain*). 66
4.4 Node structure of dimensions, measures, report items and sheets in the output graph. 75

5.1 A sheet of the demonstrated application visualized in Qlik Sense. 79
5.2 Visualization of loading a table from a database. 84
5.3 Visualization of a shared dimension *SpjeHpu* and its usage in a report item. 85
5.4 Visualization of a shared Report Item used in both application's sheets. 86
5.5 Visualization of a sheet of our sample Qlik Sense application. . . . 87
5.6 A part of the output graph with redundant orphan nodes. 89

List of Tables

3.1 Example of the effect of TOTAL modifier (A is short for 'Amount'). 43

A. Attachments

A.1 User Documentation

To run extraction and data flow analysis of our scanner module, several environment requirements need to be fulfilled:

- Java 8 needs to be installed on your computer.
- You need to have an access to a Qlik Sense server with sufficient privileges to read and extract Qlik Sense application metadata.
- MANTA Flow has to be installed on your computer. This can be a major problem since only customers and developers of MANTA usually have access to this program.

A.1.1 Building the project

As we have mentioned earlier, our code is split into two main parts - the Extractor and the Data Flow Generator. It is necessary to build both these parts separately by running the *mvn clean install* command in the top-level directory of each component. However, because dependencies of our project include libraries which are an exclusive property of MANTA, it is necessary to have access to these libraries before the building process begins, otherwise it is not possible to build these scanner module parts.

A.1.2 Running the Scanner Module

Once these module parts are built, they can be included in the Manta Flow platform and after filling in relevant fields in the *.properties* file used by our extractor (which defines the URL and the port of the server or user credentials), an extraction and data flow analysis can be launched by running either *_run.sh* (UNIX-like systems) or *_run.bat* (Windows systems) scripts.

The output of the analysis can then be visualized and examined by the Manta Flow Viewer, as we have shown in some of the figures used in this thesis work.

Note that none of the configuration files used for integration with the Manta Flow platform are included in the attachment as parts of these files are not entirely created by us and also because this is a proprietary piece of software belonging to MANTA.

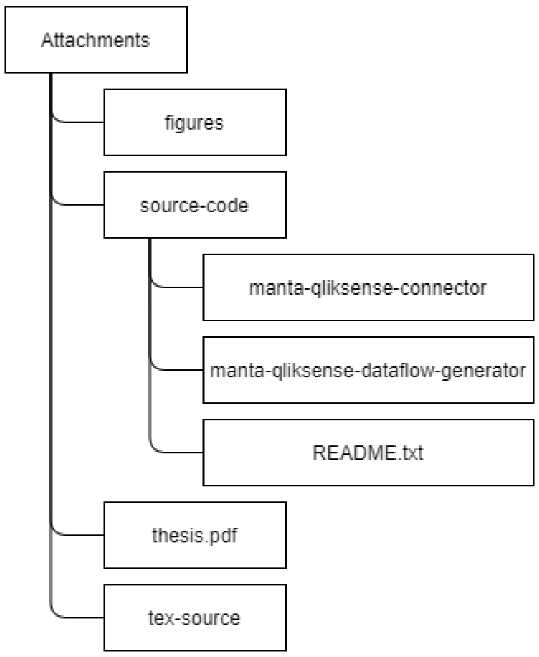
A.2 Contents of the Attachment

The attachment comprises of the following items:

The *figures* folder, which contains figures used in this work so that the reader can examine them in higher detail.

The *source-code* folder, which contains the source code of our two scanner module parts.

The *thesis.pdf*, which is the PDF version of this text.



The *tex-source* folder, which contains the TeX source code used for compiling this text.