Data Lineage Editor

By

Nick Belzer
Buster Bernstein
Jasper Geurtz
Rens Hijdra
Philippe Lammerts
Henk-Jan Wermelink

To obtain the degree of Bachelor of Science at the Delft University of Technology, to be presented and defended publicly on Tuesday July 2, 2019 at 11:00 AM.

Project duration:	April 23, 2019 - July 5, 2019	
Bachelor Project Committee:	D. Kraai, MSc	Senior Consultant, Client
	Dr. C. Lofi	Coach, TU Delft
	Ir. O.W. Visser	Bachelor Project Coordinator, TU Delft
	Dr. H. Wang	Bachelor Project Coordinator, TU Delft

Foreword

This report is the result of the Bachelor Project for the Bachelor Computer Science & Engineering at Delft University of Technology. This project aims to use the knowledge we have gained throughout our bachelors and apply it in a real-world project. For ten weeks, we have been working, in collaboration with the client, on an application that eases the process of creating data lineage diagrams. The report covers the process of the project, the implementation, the evaluation, and discusses future work and limitations.

We want to thank our supervisor, Dr. Christoph Lofi, for all his insights and coaching on how to approach this project. We also want to thank the client for their invested time and ongoing interest and support throughout the project. Moreover, we want to thank the manager of the client for the opportunity to do a software project within a business perspective. Learning about business needs and the value of software in business has been a great learning experience.

Contents

1 Summary

Mapping out the data landscape of a company or organization has become a necessity with recent government regulations such as the GDPR[1]. These regulations require companies and organizations to account for their data. It is necessary to show what the source of certain data is, and what places use certain values. Data lineage can give insight into this; however, creating this data lineage can be a tedious and costly task. Our client deems current solutions for this inappropriate or insufficient. To this end, we have created a 'Data Lineage Editor' for our Bachelor Project.

The 'Data Lineage Editor', which is a web application, can not only visualize and edit the current data lineage but, among others, also view the lineage on multiple layers of abstraction; highlight lineage flows; seamlessly integrate with the client's infrastructure, and automatically generate a practical layout. One of the unique aspects of this project has been designing an application that visualises data lineage, while maintaining a low learning curve which allows more people to easily create and understand how data flows through a data landscape. We have conducted a usability and utility test at the end of the project; the results of these tests give us confidence that our application fulfils the needs of the client.

[1]https://gdpr-info.eu/

2 Introduction

Data responsibility is becoming a more significant issue in the world at this moment. Governments require companies and organizations to be able to explain where their data is coming and going. Companies often have no clear overview of how the data flows through their data landscape, and have to figure this out. To do this, companies have to provide maps of their data landscape. Our client, part of an auditing firm, is developing and running a suite of software as a service (SaaS) applications, for these companies, that help address business issues in Finance, Tax, GRC, Data Management, and Operations. They asked us to develop an application that can help companies to keep track of the different data sources and the connections between them.

Various issues arise when figuring out and building these maps. Thus, this report describes the design and the implementation of a web application that alleviates some of these issues.

In the problem analysis (section 4) and conceptual solution (section 5) an initial approach is devised to a set of problems defined in the problem definition (section 3).

The design and implementation (section 6) builds upon the conceptual solution and explains our final solution to the problem. In section 7, we describe the approach we took towards the entire project and how the team cooperated with the client. Further, it explains the software development methods and tools adopted.

Due to effective planning and communication, the team was able to evaluate the product after implementation, which is discussed in section 8. This section places emphasis on whether our application solves the initial problems posed by the client, and whether the end product is what the client wanted. It also investigates whether our project addresses the right issues, our application can provide a better way to model lineage in the industry and help businesses in understanding their data landscapes by providing evidence and clarity in regulatory procedures.

Finally, in section 9 we reflect on the evaluation, the final product, and the entire project. We also explore the ethical implications of our project, current limitations, possible extensions, and future recommendations.

3 Problem Definition

This section will introduce the context of the problem and then explain why there is an actual problem.

3.1 Context

In the last decade, various government regulations and laws have emerged that mandate companies to be more open about the data they have, and how it is used. For example, the government obliges insurance companies to prove where the data in the reports they deliver comes from (Solvency II). Other examples include regulatory laws like GDPR (Council of European Union, 2016), HIPAA (The U.S. Department of Health and Human Services HHS, 2013), and BCBS 239 (Basel Committee on Banking Supervision BCBS, 2013), which require organizations to be able to prove that they are compliant.

To do this, companies have to provide maps of their data landscape. These maps should indicate where data originates from, what kind of transformations are being applied to it and where it ends up. In a business context, this is referred to as data lineage.

One of the problems is that organizations often do not have a complete overview of how the data flows through their data landscape. This is usually because there are various divisions in a company that each manage different processes on the data landscape. For instance, a security department is focused on very different data than a division that manages clients personal data.

Because of the lack of overview, the client explained that often, the data landscape is divided into separate data lineage diagrams (that operate on the same data). Such data lineage diagrams are diagrams that display a visual representation of how data flows through the data landscape with information about the objects involved in that data landscape.

3.2 Problem

Figuring out data lineage diagrams is a tedious process. Since there is no standardized application that is flexible enough, organizations have to resort to external applications like Visio[2] or Excel[3] to work out the data lineage in their data landscape. The disadvantage of using external tools is that it is hard to have one centralized place to store resulting lineage maps (for example multiple Visio or Excel documents on multiple PCs with slight differences).

An example of creating a lineage diagram via excel can be found in Figure 1.

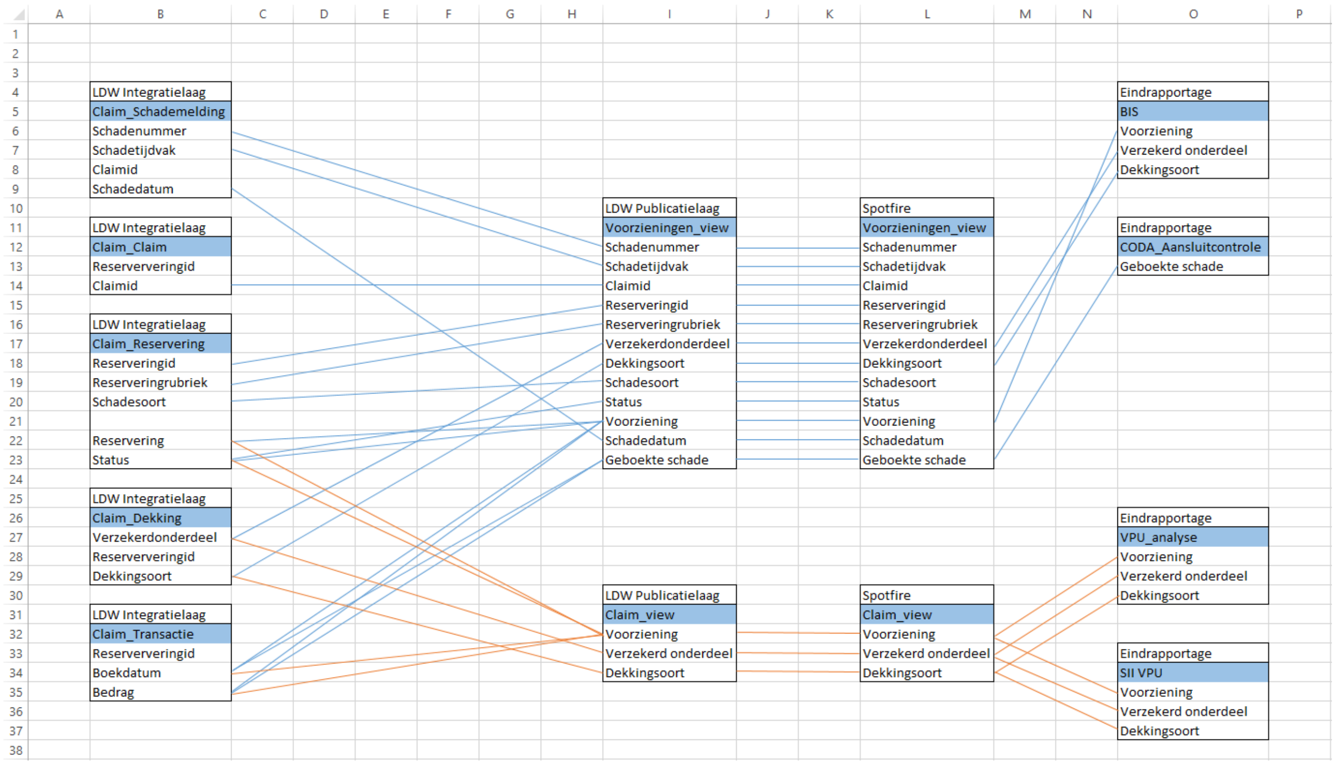
The client, an accounting and auditing firm, noticed that creating these lineage maps using these external tools is tedious and has various downsides:

- Slow and expensive

  When using an external application, the user has to copy over the whole data landscape (or at least the parts that are relevant) it wants to display in this application. This is labor-intensive, which is costly timewise and therefore also financially.

[2] `https://products.office.com/en-ww/visio/flowchart-software`
[3] `https://products.office.com/en/excel`

Figure 1: Manually creating a lineage diagram using excel



- Have to design layout manually

  In the existing applications, the user has to design the layout of the diagram manually. For small diagrams this is very doable, however, for a bigger diagram, this gets more labor-intensive.

- Prone to errors

  Because parts of the data landscape have to be copied to the external application, it can lead to inconsistencies and errors due to the human nature of doing repetitive tasks.

- Version issues

  When some information in the data landscape changes in the existing solutions, the user has to manually edit this change in each of the diagrams in which the change would occur. This is hard to do unless precise mappings exist about what each external diagrams contain.

The client approached us to help them out with these problems. What the client wanted us to do was to build an application (a visualizer and editor to help them make data lineage diagrams). The purpose being that this application can then, for instance, be used to show the government these (data lineage) diagrams, so that they can prove that they are compliant (as mentioned in the context in 3.1).

Additionally, the client wants us to try and solve the challenge of automatically generating data lineage. Steps are being taken towards finding (partial) solutions to automatically generating lineage, for example in the scientific literature some approaches are developed (Buneman et al. (2000), Benjelloun et al.

(2006), and Jang et al. (2013)). Also businesses such as SQLDep[4] or MANTA[5] provide automatic lineage through a data provenance concept called Query Inversion (Buneman et al., 2000). However, generalizing this remains an open problem.

In the future these research and/or technology advancements might be capable of dealing with the complexity of the data lineage process that organizations have to face, but until that time 'figuring out data lineage' will have to be done manually.

In our research report from week 3, we analyzed the feasibility of automatically detecting generating data lineage (under some given constraints) but decided that this was not possible in the given timespan. However, we would consider this wish so that the application can stay relevant in the future when automatic data lineage detection might be (partially) solved.

[4] `https://app.sqldep.com/queryflow/demo/`
[5] `https://getmanta.com/`

4 Problem Analysis

This project is about making some kind of tool that can visualize and edit data lineage diagrams. The purpose of this tool is to address the problems that arise during the process of creating data lineage diagrams. In this section we further analyze what the client wants from this tool, what the core issues are that need to be addressed, and whether no out of the box solutions exist for these issues.

4.1 What the client wants

Through conversations, meetings, and communicating with the client we learned about the general features that are expected of this tool. We have also spoken with end-users who make these diagrams and got a good indication of the general workflow by which data lineage diagrams are created.

First of all, mapping out **all** data lineage in the data landscape is **not** the intention. This is because there exist various kinds of data flows on the same data landscape. For instance, the *Finance Department* might hold a data lineage diagram that contains the relevant data flow related to finance. Then another department like the *Security Department* may have a different data lineage diagram related to all security-related data flows. It is very well possible that these two data lineage diagrams use the same tables or attributes, and so it is necessary that a solution should keep these different diagrams separate.

The client wants to be able to view a data lineage diagram on their screen to get some kind of overview of what that diagram looks like. This means that on their screen they wish to see one or more system(s), tables, or attributes, that are connected with lines indicating that there exists some kind of relation between these items. It is also essential that users can change the layout of such a diagram so that they can make the diagram visually clear. Because not everyone is interested in the full details of a diagram, another important wish is that these diagrams can be displayed on different levels. These levels should collapse part(s) of the diagram (for instance all attributes in a table) so that the diagram can also be viewed in more compact forms.

The client also expresses a need to be able to build these diagrams from scratch. This means being able to make new diagram instances that can then be edited. Here systems, tables, and attributes from the data landscape should be available to be selected. Once selected and added to the diagram, these should be able to be linked to other parts of the diagram to indicate that there is some kind of lineage here.

The client wants this tool to be (eventually) hosted on their data analytics platform, so that users can visualize and edit these diagrams from their platform, and that these diagrams are stored in a central location. Furthermore, this integration is useful since the client's analytics platform already provides all the systems, tables, and attributes from a data landscape. This means that the tool should also use the information available in this data landscape when working with these diagrams.

4.2 Goal of the project

To be able to visualize and edit diagrams, the first necessity is defining some model that describes what a data lineage diagram is. Without such a model it is unknown if any conceptual solution is solving what it is that the client wants such a diagram to be. This in turn would lead to not addressing the problems faced currently, which is the primary objective of this project.

The second goal of this project is devising a solution that can be integrated into their data analytics platform. It should be fine if this starts as an independent project to focus on functionality, but eventually, if this product cannot be integrated, it will have no use for the client.

Only when these two goals are addressed, any other problems the application would resolve will be relevant for the client. This third goal is therefore some way to visualize and edit these data lineage diagrams.

For 'doing automatic lineage', which was addressed at the end of subsection 3.2, this is not a goal of the project. If time allows we will look into the possibilities of solving some corner cases related to automatic lineage. This is important to at least keep in mind during formalising requirements so that the solution at least remains extensible towards this client's wish.

To summarize, the client needs a combination of these things:

- Some model that describes a data lineage diagram.
- That the solution can be integrated into their data analytics platform.
- Some way to visualize and edit these data lineage diagrams.

4.3 Novelty of the problem

With an understanding of what the goal of solving this problem is about, we should make sure that there does not already exist a solution. Because if such a solution exists, why should we bother making this tool, we can recommend that solution to the client and be done.

For visualizing and editing, there are many successful tools on the market. Examples include applications like Microsoft Visio or draw.io. These tools are meant for visualizing and editing diagrams, and to such end, are also perfectly capable of supporting the visualization and editing of 'data lineage diagrams'. These types of applications, however, do not have all the wishes of the client as argued in 3.2.

Another group that perhaps comes closest to the solution of what the client needs are Talend[6], Informatica[7] or Collibra[8]. These tools give similar solutions, but focus on the transformations and have no easy way of separating diagrams that relate to different use cases. Moreover, the client has indicated that these tools are too big of a step to use because of the financial investment upfront and the learning curve required. They also are considerably more complex than what the client wishes for in a data lineage tool, and from analyzing the capabilities of these tools/platforms, it became clear that they did not offer the intuitive viewing modes that the client wishes for.

So while many tools exist, unfortunately, none address the correct combination that suits the client's needs.

4.4 Requirements

Based on the wishes of the client and the feasibility of lineage generation, we determined the priorities for the project with the client. Together we discussed more detailed requirements which the product should

[6]`https://www.talend.com/`

[7]`https://www.informatica.com/nl/`

[8]`https://www.collibra.com/`

fulfil and listed these requirements in order of importance using the MoSCoW method[9]. The *Must Haves* must be implemented to be able to serve a prototype application. The *Should Haves* would increase the quality of the product further. We have included several items that relate to automatic lineage generation in the *Could Haves.* If time allows we will look into the possibilities of solving some corner cases. Finally, the *Won't Haves* contain remaining features that while requested, are not feasible and thus have been explicitly excluded.

Detailed information about the MoSCoW can be found in Appendix A.

[9] `https://en.wikipedia.org/wiki/MoSCoW_method`

5 Conceptual Solution

As defined in the problem analysis (section 4), there is a need for a solution that covers the set of wished for features. In this section we define what a solution could look like.

For ease, we have defined the set of features here again and will tackle them one by one in the sections below.

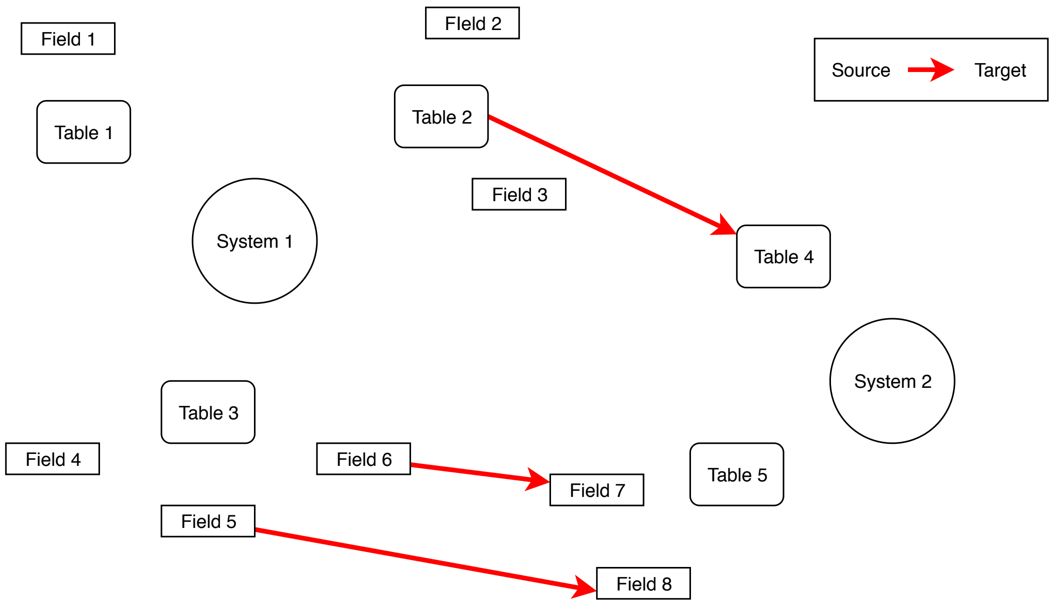
- Some model that describes a data lineage diagram.
- That the solution can be integrated into their data analytics platform.
- Some way to visualize and edit these data lineage diagrams.

5.1 Model for a data lineage diagram

To better understand how we can model data lineage from a software engineering perspective, we look at the definition given in section 3. Here data lineage is defined as the mapping of a data landscape that indicates where data comes from, how it is transformed and where it ends up. These transformations are one way between systems, tables and attributes. This means there are two types of connections which are required in a lineage diagram, the hierarchical connection between systems, tables and attributes and the actual transformations of data between them.

Based on this definition, it seems like a data lineage diagram can be modeled as a graph. So let's try to work this out. A graph consists of vertices and edges which connect the given vertices. From our definition of data lineage, we can define the systems, tables and attributes as our vertices. These vertices are connected by one-way links which we can model as our edges. An example of a data lineage diagram instance in graph form can be seen in figure 2.

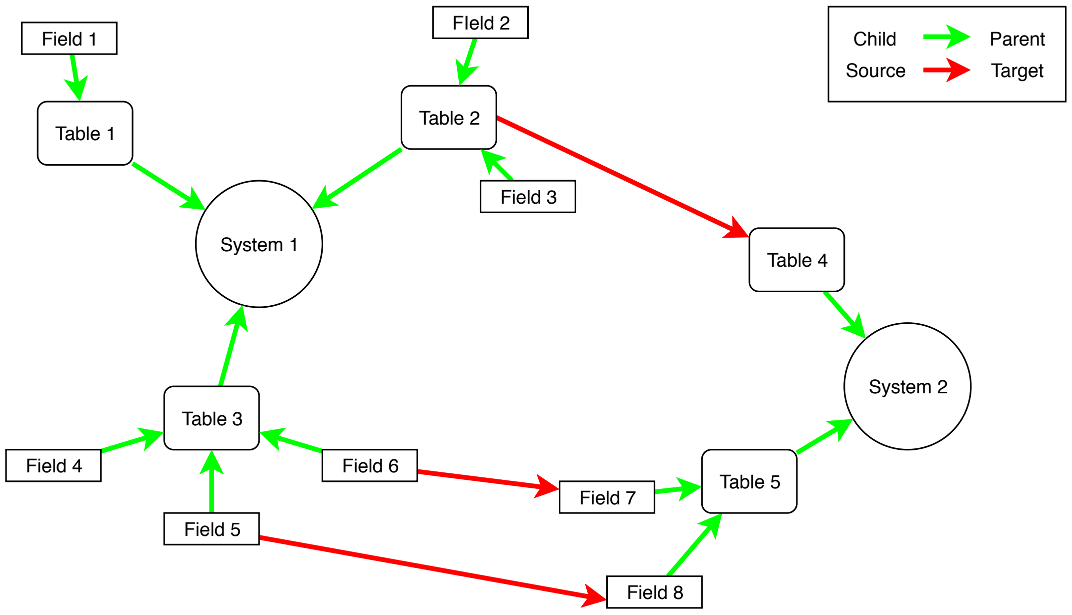
Figure 2: Lineage diagram as a graph



However, as we can see the figure, in our mapping to a directed graph, we lose the connection between the nodes themselves. As described in the analysis (section 4.1), the client wants to see the diagram on different abstraction levels. Therefore the child-parent relationship between systems, tables, and attributes should be present in the graph.

If we adjust our mapping slightly, we can also incorporate these connections into our graph. Just like the lineage edges are directed, so are the child-parent relationships between the nodes. A table has a single system as a parent. This allows us to add these relationships as directed edges between nodes. An example of how this would work can be seen in figure 3.

Figure 3: Lineage diagram as a graph with hierarchical connections



This graph-based way to view our lineage model seems to be able to store all the required information, both about the relationships between nodes and the actual lineage transformations.

5.2 Conceptual solution to integrate

As defined in the problem analysis, the client wants to use the developed application from within their analytics platform. This platform is built out of a set of web applications that each add specific functionality. As defined in the requirements, the developed application should fit into this set of applications.

The most obvious solution is to develop a web application like the other applications available in the analytics platform. This will make sure the application is easy to integrate as it requires the same process as for the existing applications, which is already familiar to the client.

This slowly guides us to a practical solution. To develop a web application that can visualize the given graph format, which represents the lineage model. The integration with the analytics platform provides an additional benefit as it can serve as a backend to the application where the lineage models can be saved or from which they can be loaded.

As defined in the problem definition (section 3) it is *hard to keep in sync with the reality of the data landscape* as currently when there exist multiple diagrams of the data landscape, they all need to be manually updated when a node in the landscape changes. The integration with the analytics platform provides a solution to this as it already contains a mapping of the data landscape. By having diagrams reference to these mapped nodes stored at this platform (instead of using a copy) a single source of truth is introduced that solves the problem of keeping different models in sync.

So building the application as a web app provides us with the capabilities to store and load lineage models and keep them up to date with the latest changes for each node.

5.3 Visualisation and Editing of the model

The primary features of the application are the ability to visualize and edit lineage diagram instances. Because the application will be developed as a web app, we have access to all JavaScript libraries. While it is possible for the team to implement a visualization library themselves, this would not be a sensible choice for multiple reasons. The primary one being that many existing libraries already solve this problem. Therefore the team would be reinventing the wheel. Next to that, it would take a significant amount of time of the project to implement a library like this. This would take time away that could otherwise be spent on implementing the requested features.

Therefore if the team would use one of the existing libraries for visualization, they could abstract away from details about rendering diagrams to the screen. Instead, they could focus on implementing the features requested by the client as defined in the list of requirements in section 4. However, using a library requires that the graph format that defines our lineage model should be converted into a format that can be understood by the chosen library. This, however, is an easier task compared to creating a new visualization library.

6 Design and Implementation

This section discusses the two main aspects of our design and implementation. It builds upon the conceptual solution (section 5), providing the details as to how the outlined solution was implemented. First, a model to describe a data lineage diagram is defined. Secondly, we discuss in more detail the design choices and how features of the application have been implemented.

6.1 Client infrastructure notes

Before everything, there is a need to define the client's infrastructure as this shaped some of the decisions made during the design and implementation process.

6.1.1 Client Data

The client has modeled their internal data as *systems*, *tables*, *attributes*, *reports* and *report fields*. These all have a hierarchical ordering; *attributes* belong to *tables*, *tables* belong to *systems* and *report fields* belong to *reports*. *Systems* can be things like a SAP[10] System, this can contain multiple databases that have *tables*, which again have *attributes*. *Reports* with their *report fields* are almost always based on (multiple) *attributes*. For example a finance report can combine data from multiple tables. Finding the source(s) of this data can be crucial as mentioned in subsection 3.1.

From this point on, for clarity and simplicity reasons, if *systems*, *tables* and/or *attributes* are used, it is implied that *reports* and *report fields* are also included as hierarchy-wise *tables* are equivalent to *reports* and *attributes* are equivalent to *report fields*.

6.1.2 Client APIs

The client allows their data about the existing *systems*, *tables* and *attributes* to be queried via REST APIs. All of these can only be queried all at once, e.g. all existing *systems* can be queried via one call which returns a list of existing *systems*.

6.2 Model for a data lineage diagram

As defined in the problem analysis (section 4), to enable the development of a tool that could solve the problems identified in the problem definition (section 3) there is a need to model lineage data in a flexible format. As described in the conceptual solution (section 5), a directed graph-like model is required with nodes and edges. In the sections below, we explain how we implemented this model.

6.2.1 Nodes and Edges

The *systems*, *tables* and *attributes* (as defined in 6.1.1) are the nodes of our graph, which are of different types. Therefore it makes sense to have a **type** property for the nodes that define whether the node is a *system*, *table* or *attribute*.

[10] `https://www.sap.com`

Tables belong to *systems*, *attributes* to *tables* etc., so a **parent** property is required, however not every type has a parent, e.g. a *system* has no parent, so this property should be optional.

For the edges, which symbolize the lineage trough the data landscape, a **source** and **target** property are required.

While designing the original format at the start of the project, there was a discussion about whether to use a *direct reference/pointer* or an identifier for the source, target and parent attributes defined above. We have decided to use a *UUID* (Universally unique identifier) which requires an extra layer of separation as there has to be a place to fetch a node or edge based on a *UUID*. However, using a *UUID* comes with a great benefit as it allows the loading of all nodes and edges in parallel: by using *UUIDs tables* do not need to wait for all *systems* to be loaded, and the loader can just set a *UUID* as a parent instead of a direct reference.

Since the external APIs (as described in 6.1.2) only let us query all the *systems*, *tables*, and *attributes* at the same time, using *UUIDs* was an obvious choice since it has allowed conversion to start without having to wait for queries to complete.

Now all the required properties to create our graph structure are discussed. However to be able to display useful information about a node or edge (some meta-data about the *system*, *table* or *attribute*) we need a different property. Examples of this type of information could be a name, an owner, a risk-classification, or any other relevant information. Therefore this property that both nodes and edges have is the **data** property. This property is a generic key-value map, which does not have a fixed size. An example map would be:

```
{
  {key: "Name", value: "Marchen"},
  {key: "Creator", value: "Bruder Grimm"},
}
```

As no assumptions are made about which information should be added this allows the nodes to have any additional data.

To support all this the following format is used: a graph consists of GraphElements which have *UUIDs* and *data*. These GraphElements can be split up in Nodes and Edges which add their *type* & *parent* and *source* & *target* respectively.

```
abstract GraphElement {
    uuid: UUID
    data: map<string,string>
}

Node extends GraphElement {
    type: string
    parent: optional<UUID>
}

Edge extends GraphElement {
    source: UUID
    target: UUID
}
```

6.2.2 Conversion to and from sheetlike formats

This format also allows for conversion from and to sheetlike formats (e.g CSV and Excel), which was one of the requirements as these are often used in a business setting.

For example take the following very small sample:

```
nodes.csv
| UUID | Type    | Parent | Name    | Owner | ...
+------+---------+--------+---------+-------+-----
| 0001 | System  |        | System1 | Jan   | ...
| 0002 | Table   | 0001   | Table1  | Piet  | ...
| 0003 | Table   | 0001   | Table2  | Joris | ...
| .... | ...     | ...    | ...     | ...   | ...

edges.csv
| UUID | Source  | Target | Creator | ...
+------+---------+--------+---------+-----
| 1001 | 0002    | 0003   | Corneel | ...
| .... | ...     | ...    | ...     | ...
```

In our application this will be mapped one-to-one for *UUID*, *type*, *parent*, *source* and *target* and all the other rows will be put into the *data* information map

```
nodes = {
  {uuid:0001,type:System,parent:None,data:{Name:System1,Owner:Jan,...}},
  {uuid:0002,type:Table,parent:0001,data:{Name:Table1,Owner:Piet,...}},
  {uuid:0003,type:Table,parent:0001,data:{Name:Table2,Owner:Joris,...}},
  ...
}

edges = {
  {uuid:1001,source:0002,target:0003,data:{Creator:Corneel,...}},
  ...
}
```

This conversion is one example of how one could convert from and to a different format. It gives confidence that the designed data format is specific enough to provide enough details about how the lineage flows and on what levels, yet generic enough to be extended to new node types and other relevant information.

6.2.3 A data lineage diagram instance

As explained in the client's wishes (section 4.1), the intention of the client is not to create one global mapping of all lineage between the systems they have. The intention is to define 'lineage diagram instances' which contain a select amount of nodes (systems, tables, and attributes) and the connections between them.

Each lineage diagram is likely to have a different purpose for the user and should, therefore, be able to show any part of the mapped system desired for that purpose. This includes the ability to show the same node or the same edge in different diagrams.

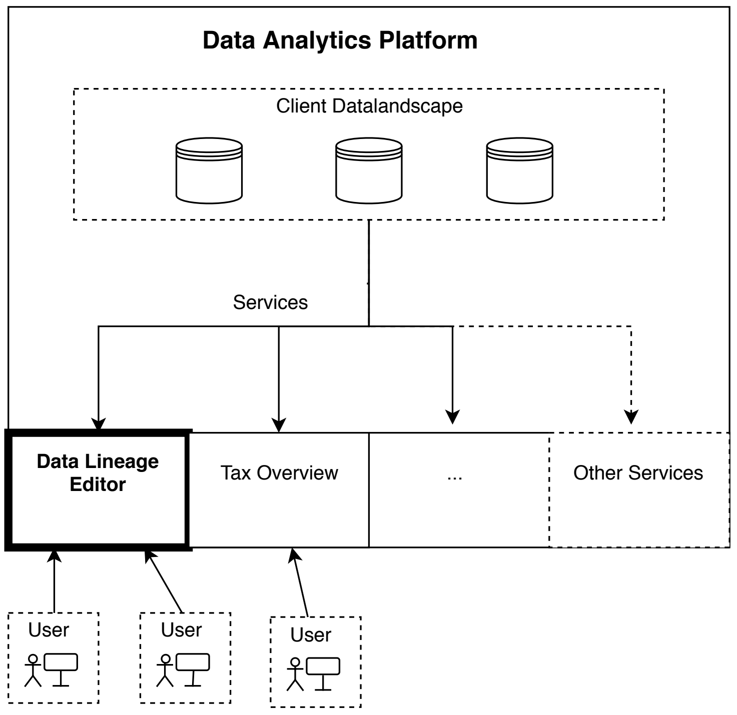
A lineage diagram is defined by the nodes visualized and the edges to be shown in between them. Next to that metadata about the nodes in a lineage diagram is stored, like the location in the diagram. An example of the format can be seen below.

```
LineageDiagram {
    uuid: UIID
    name: string
    nodes: array<UUID>
    edges: array<UUID>
}
```

6.3 Application

With a defined model it is time to dive into the design and implementation of our web application. Figure 4 shows where our application fits in the existing infrastructure.

Figure 4: Infrastructure



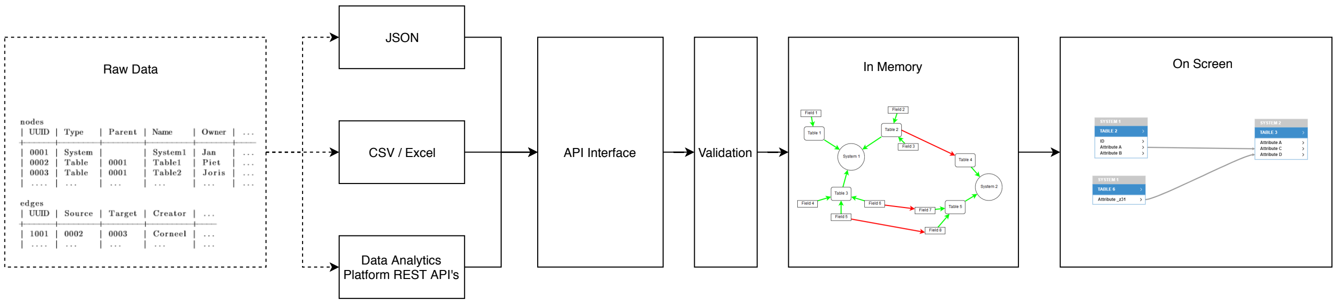
6.3.1 Approach

For the implementation of our front-end application, we have taken an object-oriented approach. We use typescript as the main programming language (this compiles to JavaScript and is thus also available in every browser). Typescript provides us with the benefit of being able to define new objects and types, like *Nodes*, *Edges*, or *Graphs*. For the visual components on the screen (HTML), we have used VueJS. This again lets us design components that can be put together like building blocks. These separate parts can then easily be tested individually. These tests give more certainty that the code is behaving as expected. Additionally, a benefit of this design choice is that it is easy to rearrange components and functionality so that we can easily redesign our layout based on client feedback.

6.3.2 Architecture

The architecture of the front-end application contains various layers between where data enters the application, and to what is visualised on the screen. Figure 5 contains the main stages in the retrieval en writing of data to the back-end.

Figure 5: Architecture



```
nodes
| UUID | Type   | Parent | Name    | Owner | ...
+------+--------+--------+---------+-------+----
| 0001 | System |        | System1 | Jan   | ...
| 0002 | Table  | 0001   | Table1  | Piet  | ...
| 0003 | Table  | 0001   | Table2  | Joris | ...
| .... | ...    | ...    | ...     | ...   | ...

edges
| UUID | Source | Target | Creator | ...
+------+--------+--------+---------+----
| 1001 | 0002   | 0003   | Corneel | ...
| .... | ...    | ...    | ...     | ...
```

When a request is made to load a data lineage diagram, it is called through the API. We have designed this stage using an interface so that it is easy to switch between local test data and real client data. This approach also enables, for instance, automated end to end tests that use sample data.

After data is retrieved from the API, it is validated in the validation stage. This stage verifies if data is in the right format and that references to other data objects are valid. For instance, if something is removed on the back-end platform, but some nodes or edges still reference this removed data, this stage can catch such as exceptions and notify the user of the application.

Post validation all the input data is stored in memory. From here this can be used from the interactive front-end components.

6.4 Visualizing and editing

A major feature of the application is the ability to visualise and edit the lineage diagrams instances. The implementation of which is described in this section.

6.4.1 Visualizing

The diagram is visualized on the screen using GoJS[11] (an Interactive JavaScript Diagram library for the Web) as it matched all our requirements, including the support for automatic layouts for graphs. For more information on why GoJS was chosen over others, refer to the research report in the appendix D.

GoJS uses its own type of data format for visualizing graphs. This model consists of two JSON arrays: *nodeDataArray* (the vertices) and *linkDataArray* (the edges). Since GoJS uses a different data format than is used in the application (as defined before) there is a need for a converter between the application data format and the GoJS data format.

This means that the data used for the GoJS rendering does not immediately reflect changes applied in the application data format, nor is it able to change that data (more on this in the next section, 6.4.2). In that section it is also explained how changes are exchanged between GoJS and the application data to have both be in the same state, always.

Another requirement was the ability to redraw the diagram on different levels of abstraction. This behavior has been implemented in the conversion from our data format to the GoJS format as the only change per level is the way nodes appear on the screen. More on this can be found in section 6.4.3.

6.4.2 Editing

Next to the visualisation of the lineage diagram the application should also allow for the editing of these diagrams. The chosen library, GoJS, provides the ability for events happening in the diagram to call functions in our code, so called hooks. As described above the data format provided to GoJS is not the same as the internal data format of the application. This means that when these hooks are called, code has to be executed to make sure the application data structure matches the change made by the user to the GoJS visualisation. Relevant hooks are called when graph objects are selected, added, removed, or redrawn. For the last three where an actual change is made to the diagram, it is required to propagate this change to the internal data format.

Changes made to the internal format are recorded in a changelog. The idea of which is to record changes, allow them to be reversed and to push them to the API. The reversal of the changes has currently not been implemented.

As noted before, the API of the client's analytics platform is slow and does not support the sending of multiple operations to the platform. This requires the application to send each recorded change to the API and wait for a confirmation, which can take a long time for many changes. Therefore an optimizer has been introduced which looks at the different changes made and in what order. Based on this, it merges or removes changes to shorten the total amount of changes that need to be sent to the API. This optimizer also eliminates the need for sending the changes sequentially, allowing the application to post changes in parallel.

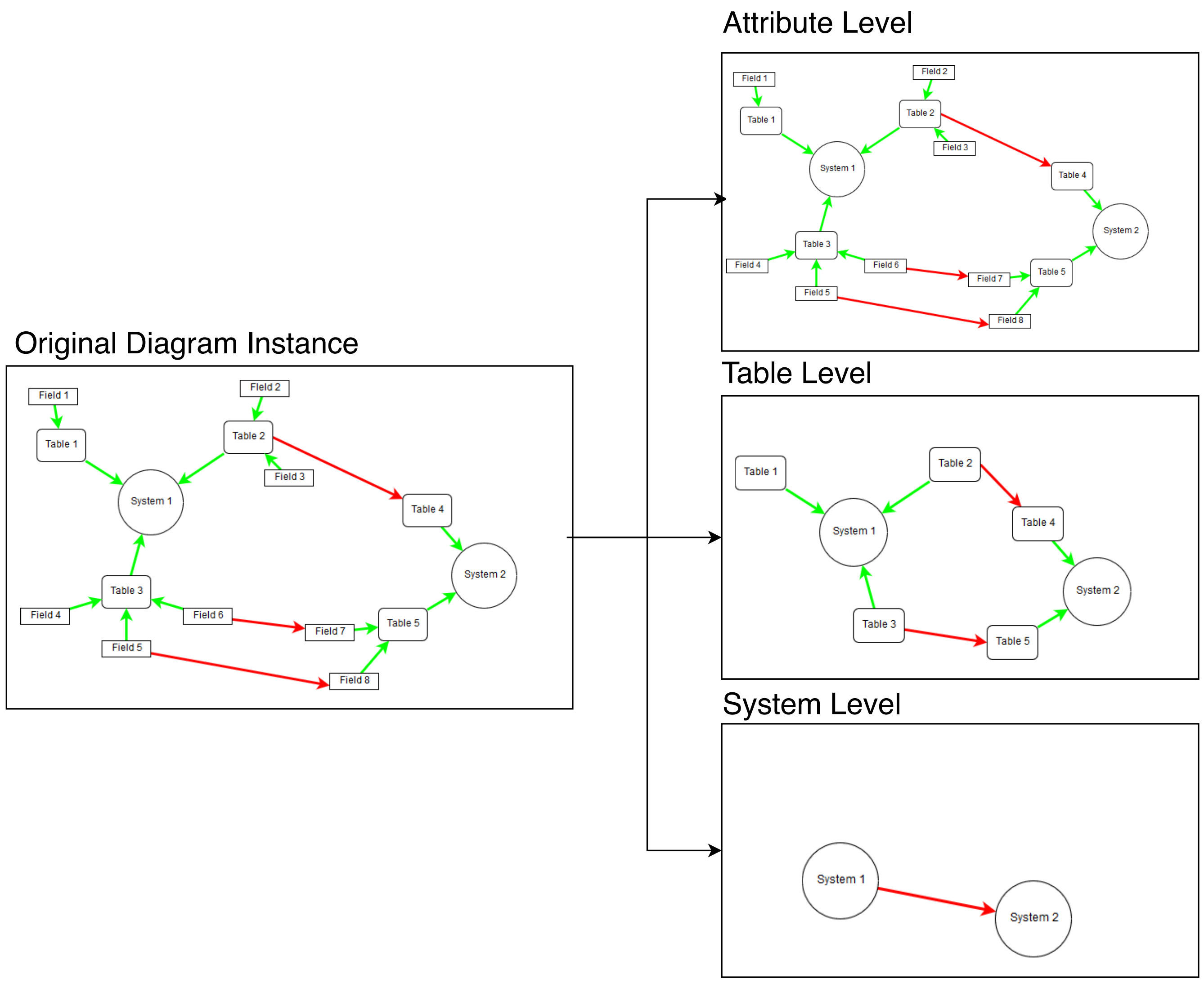
An additional feature, which is not tracked through the changelog, is the ability to save the location of all the nodes and edges (the layout). These changes are only recorded when the actual save button is pressed to optimize the number of changes that have to be written to the API.

[11] `https://gojs.net/latest/index.html`

6.4.3 Three view levels

The three levels are the **Attribute view**, the **Table view** and the **System view** as can be seen in Figure 6. The attribute view shows the most detail as it shows all the attributes of a table. All the lineage links can be shown from node to node. In the table view attributes of tables are hidden away. This means that lineage links between attributes can no longer be shown directly from and to these nodes. Therefore they are aggregated up to the table the attribute belonged to as these are still visible on this higher level. The same aggregation applies in system view where all tables are abstracted away to the systems they belong to.

Figure 6: Three view levels



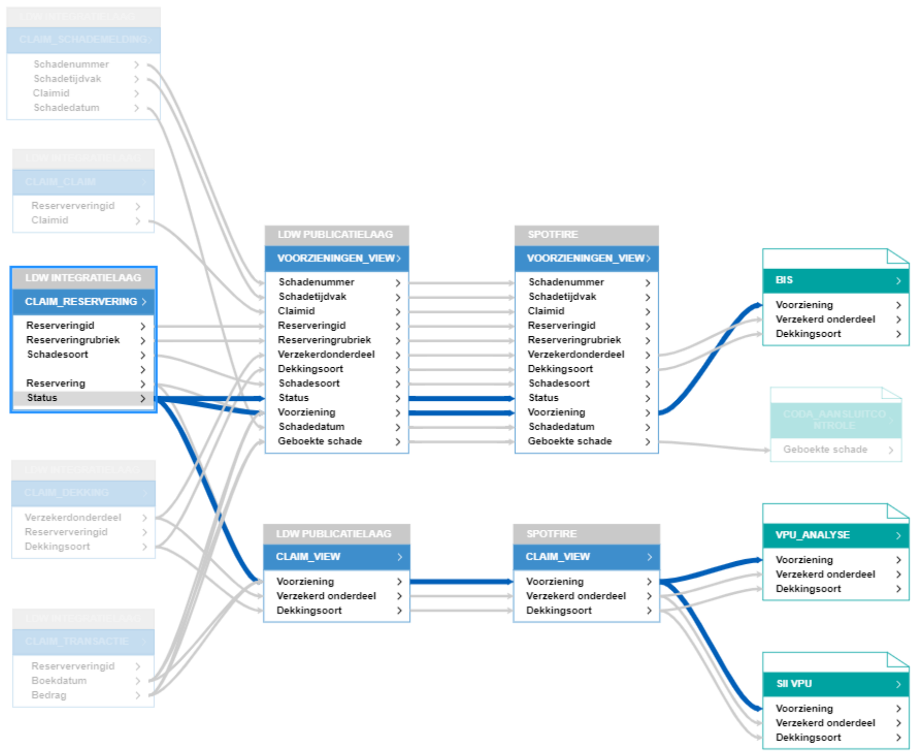
This means that when there is a link between an attribute from one (table in a) system to an attribute in another (table in a) system this link is visible from attribute to attribute in the Attribute View. In the table view, this link would show from table to table as the attributes themselves are abstracted away while in System view this would show as a link from system to system. All the while this link would remain a link between attribute and attribute, only the visualization for this link changes.

The aggregation up of these links can cause an issue the higher up we go in abstraction layers. The higher up, the more edges will mean the same. Two different lineage connections between attributes from two tables in table view would become two connections between the two tables (which would appear as the same line). To minimize the amount of rendering needed and to ease access to details about each link these links are merged during aggregation such that only one link becomes visible. Stored within this aggregated version of the links, is the information from all the links that have been aggregated. This way, the user is still able to see the details of each link.

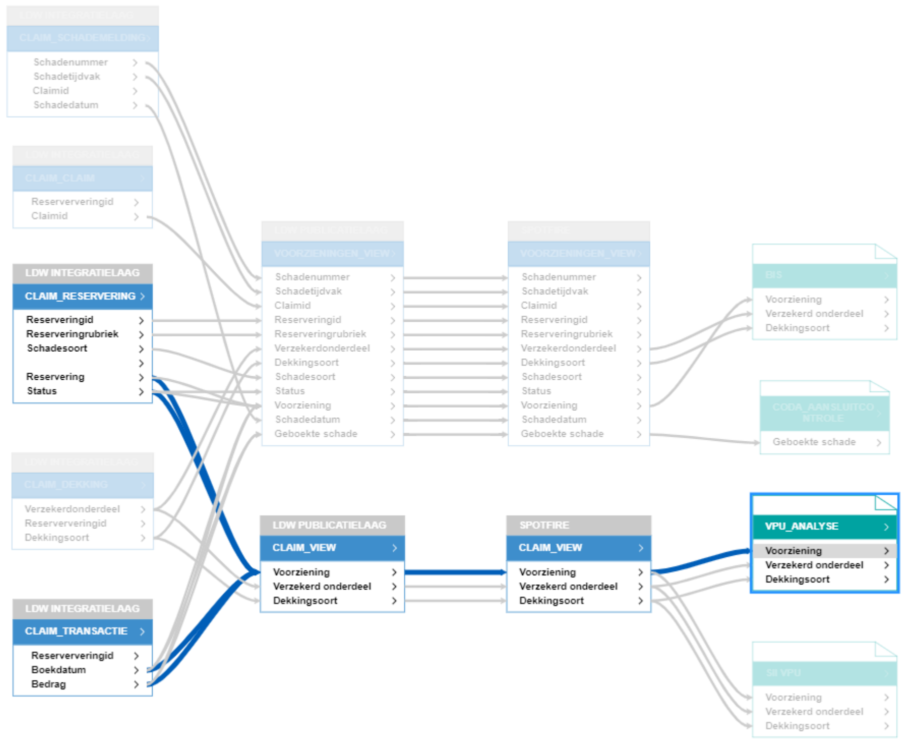
6.4.4 Trace lineage

One of the extra features of the application is being able to analyse the current diagram. The user can do this by clicking on a node to show all the connections to and from other nodes. These connections will then be highlighted in the diagram. For this to work, firstly a tracing algorithm had to be written. The first step, gathering all links connected to the clicked node, is split into two ways: tracing forwards and backwards. Then a Breadth First Search (BFS) is applied to the links going out of the target node.

When hovering on a node, all connected links are highlighted, when clicking on a node, the links stay highlighted and all nodes that are not associated with the target are dimmed. For this, it was necessary to extend the algorithm also to track the nodes that were visited during the BFS. Figure 7 shows both forward and backward lineage, as well as the dimming of irrelevant nodes.



(a) Trace of forward lineage



(b) Trace of backward lineage

Figure 7: Tracing forwards and backwards lineage on the same diagram

6.4.5 Automatically Generating a Layout

A key feature of the application is the ability to automatically lay out the currently visible lineage diagram in an orderly fashion. When designing a diagram from scratch, a user can manually add or removes nodes and edges. However, when importing large diagrams that already have large amounts of nodes and edges in an unordered fashion, this is not feasible. The GoJS library that we use already has built-in capabilities to generate layouts in different ways. For the type of graph used in this project, GoJS includes three relevant layout types:

None – Place all nodes on a grid in order of appearance and make the user align them manually.

Tree Layout – This layout positions nodes of a tree-structured graph in layers (rows or columns). Tree layout can handle a limited number of edges that would prevent the graph structure from being a tree.

Layered Digraph – This arranges nodes of directed graphs into layers (rows or columns). Due to configurations in the node packing and link straightening algorithm, these layouts become more visually pleasing than the tree layouts, however, they take quite a bit longer to generate for larger number of nodes.

All three of these layouts can be selected, and each one has its advantages. Without automatic layout, the user can customize their layout and can put nodes on arbitrary positions in the diagram. The tree layout algorithm creates a tree structure in layers from left to right and can handle nodes having multiple parents. It is also much faster for larger number of nodes (i.e. thousands). The layered digraph is similar to the tree layout, but it handles cycles in the graph much better (at the cost of speed). While this use-case will be rare, it is not unimaginable that there could be some cyclic dependencies between some systems or tables.

More information on these two algorithms can be found at the GoJS TreeLayout API and LayeredDigraphLayout API pages.

6.5 Final Product

All items merged give our final product in Figure 8. The subsections in this section correspond with the numbering on the figure to clarify all features.

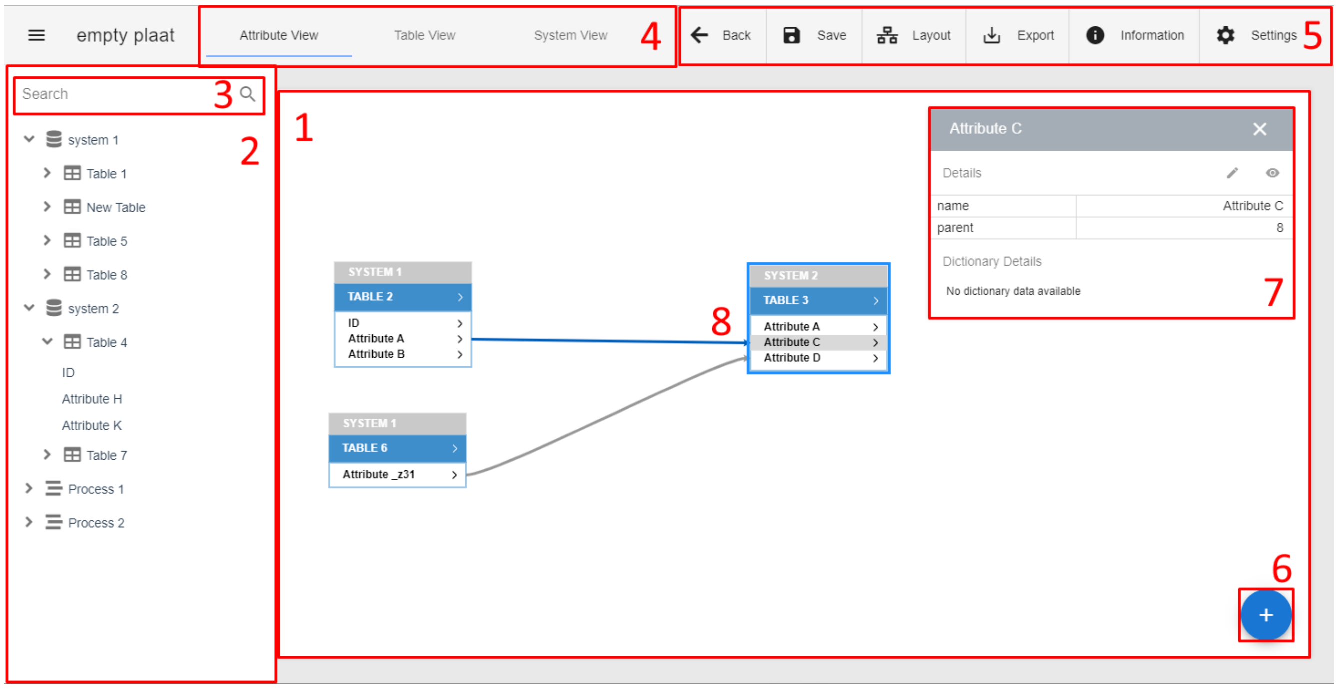


Figure 8: Overview end product

6.5.1 Canvas

One of the main requirements of the application is to be able to show systems, tables, attributes, and links between them. For this, the diagram comes into play. It is where the nodes (systems, tables, and attributes) are drawn. The main part of our application, that immediately springs into view is the canvas where all nodes and links are shown.

6.5.2 Catalog panel

To get tables to show up in the graph, one needs to be able to find them somewhere. This is what the catalog panel is used for: it shows all nodes in the system of the client in an organized tree structure. All nodes are collapsible to make the view uncluttered and only show what is wanted. From this panel, all elements are drag-and-droppable onto the canvas, which will remove them from the catalog and draw them onto the canvas.

6.5.3 Search bar

Since the catalog panel can contain many tables stored in many different systems, looking through a list of hundreds of items can be tedious. This is why we implemented a search bar. It filters the nodes in the catalog panel by a search query.

6.5.4 View switcher

Now, one has found the nodes necessary to show the lineage in question. The tables used, however, consist of tens or hundreds of attributes, and thus to get a better sense of lineage, a user might only want to view the tables or systems in the canvas. For this purpose, the view switcher comes into play. These three buttons correspond with the different levels of depth that can be visualized in the program as defined in 6.4.3.

6.5.5 Action bar

To facilitate additional functionality, the action bar contains many useful features that enables the user to interact with the diagram.

Back Go back to the diagram selection panel

Save Save the created lineage diagram

Layout Run the layout generator (section 6.4.5)

Export Download the created diagram as XLSX, SVG or PDF

Information Show statistics about the currently loaded diagram.

Settings Enable automatic layout, formatting options, grid, animations, performance settings

6.5.6 Plus button

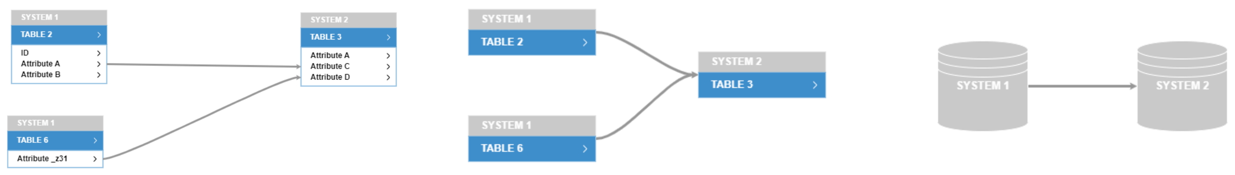
In some cases, not all data objects (nodes) are imported into the data landscape. The plus button allows for adding those nodes that don't exist yet. In this case, the user can create new systems, tables, reports, and fields to draw into the diagram. Firstly, select the type of object you want to create, then specify details such as name and parent, and the object is ready to be placed on the canvas.

6.5.7 Info panel

When all tables have been placed, it might be time for presenting or querying the created diagram. In this case, one could be interested in some metadata about objects or links. Clicking on any object in the diagram will show the information panel with all available data about the target. This could be the creator of the node, but also more specific statistics such as calculated risk.

6.5.8 Highlighting

When querying the diagram, one might be interested in finding out where data from a certain table or attribute ends up or comes from, for quality control reasons or otherwise. For this reason, hovering on systems, reports, tables, and attributes will highlight all links that originate at the target, backwards and forwards. When clicking, the links will stay highlighted, and all irrelevant nodes fade away, enabling the user to focus on the highlighted parts.



(a) Attribute level

(b) Table level

(c) System level

Figure 9: The same lineage shown on three levels

7 Process

To discover the problem and to discover what the client wants, exploratory meetings were held during the first week of the project. During the rest of the project wishes and feedback of the client have been continuously communicated through weekly meetings and other conversations to keep the focus on the right priorities. The team has made some choices such as a scrum-like workflow, a continuous integration server, and emulating office work hours. These choices and the reasoning behind them will be explained in this section.

7.1 Meetings

For this project, the team has chosen for an agile approach using sprints. After the first exploratory meetings with the client, a road-map was designed and proposed. To prevent doing unnecessary work, a weekly meeting with the client was set-up, which functioned as a *sprint review* where the team demoed the accomplishments of the past week and received feedback on that work. These sessions then shifted into a *sprint planning*. The proposed tasks of the coming week were discussed with the client, and where necessary, the client proclaimed the features they deemed of most importance for the next sprint. Using this feedback from the client, the team created the actual sprint planning and assigned issues for the coming week. Whenever other issues or bugs arose during a given week, the issue was posted on GitHub. This issue was then either assigned immediately or during the next sprint planning, depending on its importance and impact.

According to the basics of human-computer interaction (HCI), the usability of the product should be considered from the beginning of the project (Holzinger, 2005). Therefore, the team scheduled a meeting with the User Experience (UX) experts from the client every second week to focus on the usability of the product. The results of these meetings were then turned into issues at the next sprint planning.

Apart from the planned weekly general and biweekly UX meetings, the team was in contact with the client team whenever necessary. This contact was both through e-mail and Skype as well as in-person when the team was on-site at the client.

7.1.1 Team communication

Unlike previous Computer Science projects, the *Bachelor Project* is a full-time project. Because of this, it was decided to emulate office hours, i.e. working from 9 am to 5 pm. For the same reason, all team members were at work at the same location as much as possible, be it on-site with the client or in a workspace at TU Delft. Jointly, this enabled us to work together intensively, collaborate on issues, and to have easy access to each other for questions.

7.2 Maintainability

Since the team will not be working on the software after the Bachelor Project has finished, it is important that the code is maintainable. A whole new team from the client should be able to pick up the code and continue as if it is theirs. To improve maintainability, a continuous integration server (CI) has been set up. On this server we run static checks such as static type-checking and running a linter, but also dynamic checks which are compiling the code and running unit tests. The final phase of the CI is the deploy stage, where the code is compiled and deployed to a live preview server. This way, when inspecting a pull request the reviewer can immediately test the functionality in a reproducible environment, which ensures the functionality will get tested thoroughly. These steps all together will safeguard a high-quality standard in this project. The fact that this is possible is a consequence of the application being a web app, which makes this a unique and handy tool.

7.3 Code Quality

In the last part of this section, we look at the **code quality**. It provides a lightweight evaluation of the quality of the code from five different perspectives. After which an external evaluation of the code is reviewed.

Attributes and metrics from the code can be used to evaluate the code objectively. These can indicate as to whether the quality of a piece of code is good. Meaning that in general, the code is easy to extend or adjust by using proper naming, encapsulation, and separation of tasks but also by implementing proper design patterns where needed. In this report, we will take a look at the five perspectives for code quality, which is based on the quality model named SQueRE (for Standardization, 2011), defined by the Consortium for IT Software Quality. The five perspectives are: **Reliability**, **Efficiency**, **Security**, **Maintainability** and **Size**. After we have discussed these aspects, we will look at the software quality analysis done by an external entity.

7.3.1 Reliability

The reliability of software tracks the likelihood of application failures. Improving reliability equals a reduced likelihood of application downtime due to failures.

Since our application is run on the client-side, reliability for the editor means the ability to handle run time errors (mostly from invalid inputs). The team has introduced unit tests that use invalid inputs to validate what the application would do in these cases. By making sure these cases are covered by unit tests, the team reduces the likelihood of application failures.

7.3.2 Efficiency

The efficiency metric tracks the performance of the software.

The lineage editor contains a lot of parts of the application that interact with the user. To improve user experience, it is important that the application quickly responds to user input. In the editor the number of objects (both visible or in the catalog) is variable, small inefficiencies that do not show with a small input could become visible when the input scales up orders of magnitude. This means that a certain number of care has to be put into the efficiency of the code such that these inefficiencies do no show.

Aspects from the editor that could be concerning in terms of efficiency are:

Display When a lot of nodes have to be rendered to the display, this could become an issue. A practical example where the team found a solution to displaying a lot of nodes can be found in the Catalog Panel that displays all the nodes not yet visible in the diagram. These are hierarchically ordered from the high-level systems to the attributes in their respective tables. By only showing sub-nodes (for example tables under systems) once a system is opened we decrease the number of nodes on display at once and therefore the amount of time that needs to be spent on rendering.

Layout The external GoJS library provides a way to automatically layout the nodes in the graph such that the nodes and the connections between them are visible in a readable manner. This, however, is an expensive operation. Here the team is not able to directly alter the code of the external library, nor would it be trivial to implement a faster solution. To minimize the effect this has on the application. Responsiveness options have been added to the application to use different layout algorithms which are faster at the cost of clarity of the produced image.

Searching It is possible to search for nodes that are not yet visualized. A node is considered to be a match when the search query forms a subset of the name of this node. When a lot of nodes are loaded into the editor, searching through them will naturally cost more time. Right now, we use an algorithm that compares each entry to see if it matches the query. If in the future, the client decides to scale up to thousands of database systems, then the nodes could be placed into a dictionary to enable faster searching.

7.3.3 Security

The security metric tracks the risk of security breaches due to poor coding practices or architecture.

Security is not as relevant for the editor as the application is run on the client-side while authentication and storage are done by the system the application is integrated into. During the development API credential keys were used for requesting data from the analytics platform. It is important that these credentials are not published, as this would allow anyone to request private information. Therefore the team decided to design the application to load credentials from an external file, which is not on the repository.

7.3.4 Maintainability

The maintainability of software tracks how well the software can be supported in the long term. Good maintainability means that the code is easy to understand by a new developer or team and that it can easily adapt to changes in the requirements.

During the development of the application, the team has evaluated every change to the application through pull requests. During these moments of code review, other developers (from the team) went through the changes made or new code written to see if the code is well written. This means questioning whether proper design patterns are used, whether changes are really necessary, and if the applied changes are made in a manner that keeps the code easy to maintain.

To keep features of the code encapsulated, the team has abstracted different layers of the application. This starts with the API layer, which handles the retrieval of the data from an external source. This has been defined as an interface layer for which different implementations could be written. Next, this data is stored in a data store that is inspired by the Flux pattern (Inc., 2014). From here on out it can be retrieved or updated without having to think about API calls. This also allows an easy way to switch from input API to, for example, a test set.

A set of unit tests also define the expected behavior of functions and classes in the application. Part of these tests is regression tests which allow additions to the code to be made without fearing that other parts could silently have changed behavior.

7.3.5 Size

The size of the code is related to the maintainability of a piece of software. While it can be used to track progress or amount of work done it is not always the right measure to do so.

The component-based structure of Vue made it easy for the team to create simple components and therefore encapsulate behavior specific for these components. This meant that in general, the code size per component/file is kept at a minimum.

As for non-Vue files, care has been taken to encapsulate behavior or tasks into separate classes using an object-oriented approach. This has been manually checked during pull requests and has resulted in keeping the size of files and classes to a minimum.

7.3.6 SIG

During the span of the project there are two points of code review by an independent entity, the Software Improvement Group [12] (SIG). The goal of this is to advise students about the maintainability of their code and to provide points where improvements could be made.

First review The first review is the only one from which feedback was provided before the due date of this report. Therefore this is the only review of which the feedback can be discussed.

The code for the first review was sent at the beginning of week 6 of the project. Feedback was received in week 8 which was, as the team decided, the final week of implementation. In the response provided by SIG, it was noted that there exist a few methods in the code that are too long and should be split up. At the time of receiving this feedback this had already been resolved by the team and therefore no action was required.

In general the code scored 4.1 stars (out of 5) on the scale of maintainability used by the SIG. Described in their response this means that the code scores above average on maintainability. This seems in line with our expectations based on the five perspectives in section 7.3.

[12] `https://www.softwareimprovementgroup.com/`

Between the first and second review the team refactored the code to reduce interdependencies and split responsibilities up. In general the team stayed on the same course since, in general, the feedback from SIG was positive.

8 Evaluation

To determine whether the project was successful, the product has to be evaluated. This is done through three evaluations. The first is from a functional perspective and addresses whether the final product address all the requirements defined in the **project planning**. The second and third evaluations measure the application in its usefulness. Evaluating the usefulness is important since it verifies if the initial problem of the client has been solved. The usefulness of the product can be determined by analyzing whether both the **utility** and the **usability** of the product have been satisfied (Nielsen, 2003). Utility refers to the functionality of the product: "Does it do what users need?" (Nielsen, 2003). Usability refers to how easy and intuitive the user interface is to use (Nielsen, 2003).

8.1 Product Requirements Evaluation

The first evaluation analyses the completion state of the project. Does the product satisfy the customer in terms of provided features? This section reflects on all the implemented features and evaluates if the product contains the necessities that were defined together with the client at the start of the project. This is done by measuring how many features were implemented in the product based on the MoSCoW requirements as defined in appendix A.

8.1.1 Expectations

In terms of functionality, we expect that all the important features that the client wanted have been fulfilled. Based on the positive feedback from the client during the weekly demonstrations and sprints, we expect that the results from this evaluation will confirm that the product contains at least all the necessary functionality. We expect that the *could haves* have not been implemented as these did not have our focus.

8.1.2 Methodology

The MoSCoW requirements are marked by either ✓(done), P (partially done), X (not done) or $-$ (dropped in agreement with the client) to indicate whether they have been done or not.

8.1.3 Measurements

Since all MoSCoW requirements are prioritized, so *must haves* are the most important and *won't haves* are the least important. When all *must haves* is completed, it can be concluded that the product contains all the basic functionality. The amount of completed *should haves* indicate how much the client's expectations have been exceeded. *Could haves* is largely dependent on context and thus hold little weight in this evaluation.

8.1.4 Results

The results are shown in the tables below:

Must haves	**Done**	**Notes**
The application must provide support for the following types of objects: fields, tables, systems, reports, report fields and processes	✓	
The application must be able to give a visual representation of a collection of data objects (*the view*), in the form of a graph where the nodes are objects and the edges are links between these objects	✓	
The application must be able to build *the view* mentioned above from a fixed data format	✓	
The application must be extendable to new and/or unknown objects	✓	
The user must be able to view relations on *system level* (only see systems and reports), *table level* (only see systems, tables and tables) and *attribute level* to view relations on table level (see all objects)	✓	
The user must be able to create/modify/delete objects and flows in the editor such that this also happens on the client's platform	✓	
The user must be able to drag systems, tables, reports and links to different position on the screen	*P*	Done for all except links[1]
The user must be able to click on objects and links to find the information that is available about it through the given APIs	✓	
The user must be able to click on an object to highlight the lineage (highlights all connections from/to other objects from the clicked object)	✓	
The user must be able to import a file, that contains objects and links, with a fixed-format to the application	✓	

[1] After discussion with the client on this topic later, being able to relocate links was not as much of a must-have as previously thought

Should haves	**Done**	**Notes**
The user should be able to apply a filter that selects specific object types to be part of the lineage view (showing ingoing, outgoing or both directions)	✓	
The application should be able to make an automatic layout of the visualization	✓	
The application should be able to export *the view* to an image or PDF	✓	
The application should support *a view-only mode* where lineage can be displayed but not edited (e.g. based on user permissions)	*X*	
The system should log CRUD[2]edits to support version control	*P*	Changes are tracked, but version control is not supported[3]

[2] `https://en.wikipedia.org/wiki/Create,_read,_update_and_delete`
[3] When saving, every failed operation will be rolled back

Could haves	**Done**	**Notes**
The application could provide a form where the user can upload extra contextual information about the relations between the objects	X	
The application could contain a scanner that can use database interaction languages to automatically generate lineage between objects. If this lineage can be derived from the contextual information of (a) specific query language(s)	-	
The user could be able to accept suggested connections	-	
The application could be exported as a single JavaScript file, so it can be integrated as a widget	✓	
The application could export the objects and links from *the view* in CSV format	✓	

8.1.5 Discussion

As was expected, all important features have been implemented in the product. The relocation of links is the only *must have* that has been partially completed. This means that the final product contains almost all the basic functionality that had been agreed upon together with the client. Furthermore, it can be seen that some additional features (the *should haves and could haves*) have been implemented that increased the usability of the product. These results would indicate that the product should satisfy the client's needs. Throughout the project, the team collaborated closely with the client. The team had weekly discussions together with the client about the functionality of the product. One of these meetings resulted in an agreement between the team and the client to drop part of the *must have* feature that enabled draggable links. In this way, the team ensured that all the necessary features were provided to the client in the final product. Therefore, this evaluation confirms that the client is satisfied in terms of provided features on paper. However, to be completely sure whether the client is satisfied with the product, the team performed a utility study in section 8.3.

8.2 Usability Study

Besides ticking off MoSCoW requirements, there is also an important element of making sure that users can work with these finished features. Usability is important since the initial problem has difficult technical aspects attached to it. Therefore, the user interface of the product should also be intuitive for the client's employees with a less technical background. There are five usability characteristics that assess the usability of a product (Nielsen, 2003):

1. "Learnability: How easy is it for users to accomplish basic tasks the first time they encounter the design?"
2. "Efficiency: Once users have learned the design, how quickly can they perform tasks?"
3. "Memorability: When users return to the design after a period of not using it, how easily can they reestablish proficiency?"
4. "Errors: How many errors do users make, how severe are these errors, and how easily can they recover from the errors?"
5. "Satisfaction: How pleasant is it to use the design?"

In this usability study we aim to find answers to each of the five usability characteristics to verify whether the users can use the product.

8.2.1 Expectations

During each sprint review, a short demo was presented to the client. Through these demos, we were able to already receive good feedback on the usability of the product. Because of the close cooperation between the team and the client's UX experts, the team believes that the product is intuitive and easy to use. However, because the client had not been using the product in a complete real-world use case yet, it was not possible to determine the satisfaction of all the characteristics.

8.2.2 Methodology

To confirm our expectations and to find answers to the usability characteristics, a usability test was done by using a combination of a questionnaire and an observation. "Indirect usability tests, such as questionnaires or interviews, must be supplemented with direct usability tests; thinking aloud or observation would be suitable" (Holzinger, 2005). It was decided to do the observation using an experiment that simulates a hypothetical real-world use case that contains all basic tasks of the product (Ferré et al., 2001). In this use case, the user interface will be operated by an employee from the client that sits together with their customer, some company, to visualize their Data Lineage Flow. Based on the implemented features, this use case was defined by the following two scenarios:

Scenario 1:

1. Create a new data flow
2. Add two tables that exist in the companies environment
3. Look at the data flow and obtain information from the data analytics platform about the fields from the tables that have been added
4. Draw lineage between the two tables by creating a link between two fields
5. Save the current data flow
6. Delete lineage by removing the link between the two fields

Scenario 2:

1. Exit the current data flow
2. Import a spreadsheet that contains lineage data: information about the data objects and the lineage links between them
3. Apply an automatic layout to the imported data flow
4. Highlight the lineage of a report field.
5. Export the highlighted data flow

After this experiment, we followed up with a questionnaire to identify what group the participant belonged to. They had to indicate whether they were already familiar with the product or not. We also asked a couple of questions regarding their experience with user interface of the product.

8.2.3 Measurements

Learnability, *efficiency* and *errors* were measured by looking at screen recordings of the users performing each task. *Learnability* was analyzed by measuring for new users the amount of time/mouse clicks that were required to complete the basic tasks of the product. These measurements were compared to baseline measurements that quantified how an experienced user (one of the team members who was not familiar with the experiment setup) accomplishes the task. A new user is expected to complete the same tasks in at least 70% effectiveness (Rubin and Chisnell, 2008).

Efficiency was analyzed by comparing the amount of time/mouse clicks of new users with users that were already familiar with the products user interface.

Memorability could have been analyzed by comparing the results from the first usability attempt of the users with a second attempt after some time. However, because this characteristic is dependent on quite some time, it has been left out of this usability test.

Errors can be analyzed by measuring the number of mistakes that the users make during the completion of basic tasks. These mistakes can be specifically measured by counting the number of misclicks. Also, the recovery time was measured on how long it took users that made one or more errors to recover.

Satisfaction is hard to be analyzed by only looking at how the users interact with the product, but it can be analyzed by asking the users for their opinion after performing the basic tasks. Therefore the questions below were included in the questionnaire to address usability satisfaction. In the end, an analysis was done by looking at the generated statistics from the responses of the questionnaire. Each question could be answered by one of the following options: *Totally disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree or Totally agree.*

Usability questionnaire:

1. The layout of the application is clear: the interactive elements are correctly positioned on the screen.
2. The workflow of the application is intuitive. I can create what I want without having to think too much.
3. The application is visually appealing and fits the client's theme.

8.2.4 Results

In total, thirteen employees from the client participated in the usability test, of which six were already familiar with the product. The following statistics were measured based on the responses of the questionnaire and the video recordings of the experiment. The results of each scenario from the experiment are shown below in figures 10 and 11 using four different graphs. The first graph shows an overview of the average amount of time that was necessary to complete each task. These measurements were split up into the three different types of users (familiar users, new users, and a baseline user). The second graph visualizes the average amount of recovery time for each task when errors have been made by the users. The third graph visualizes the average amount of errors that were made by new users and familiar users. The last graph shows the average amount of click actions necessary to complete each task for the three different types of users. The results of the questionnaire can be seen in figure 12.

Based on the results, a couple of things can be noticed. First, we can calculate the effectiveness of new users by comparing the average total amount of time it took to complete all tasks between new users (not familiar with the product) and the baseline. The results of scenario 1 have an effectiveness rating of 64.65%, and scenario 2 has an effectiveness rating of 70.40%. Second, it can be seen from the third graph in figures 10 and 11 that new users sometimes struggle with adding tables, creating/deleting links and going to the home screen. Also, even familiar users struggle sometimes with going back home and deleting links. These errors also explain the peaks at these tasks in the time graphs. Third, the average amount of click actions seem very similar across all user types. Finally, the results from the questionnaire show that the users were very positive about the layout and the workflow of the product. Responses about the visual appearance of the application were more diverse but still positive.

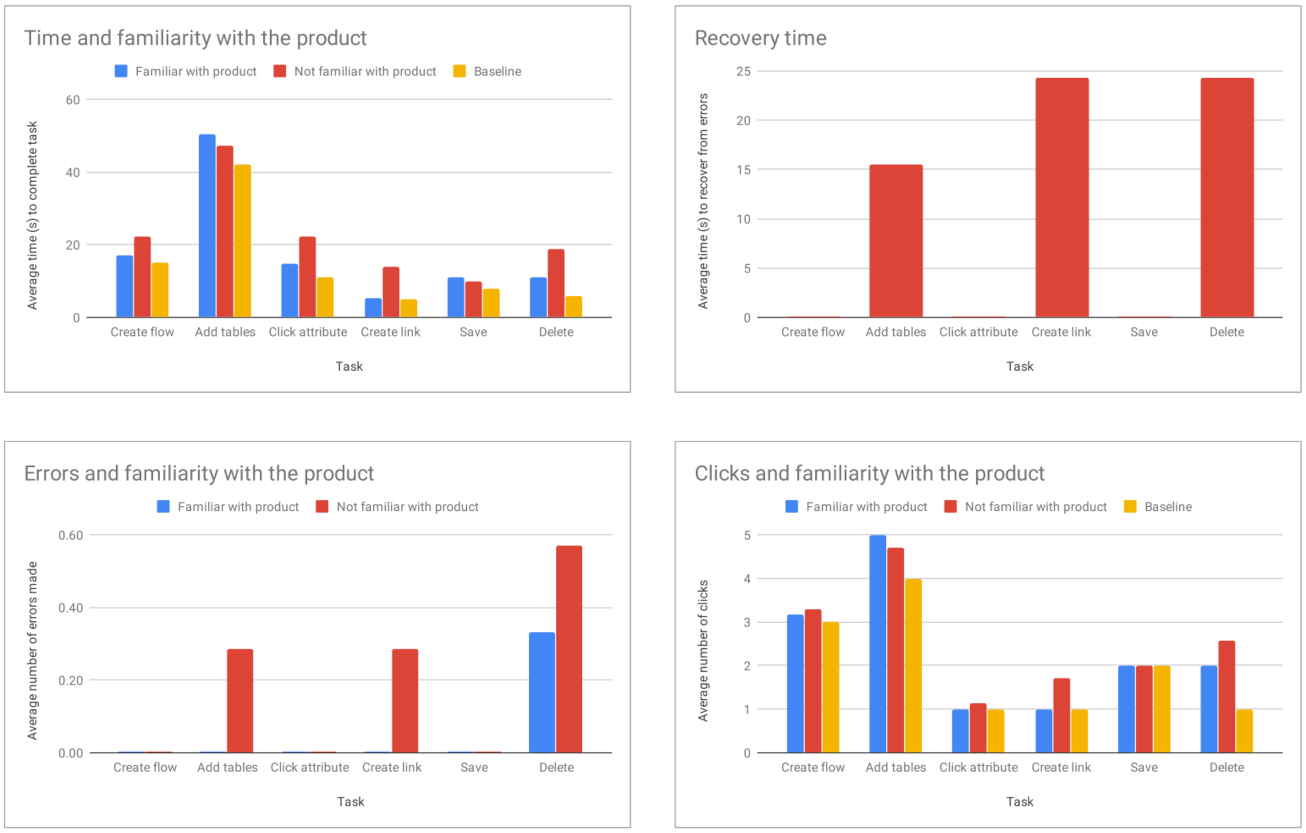


Figure 10: measurement results scenario 1

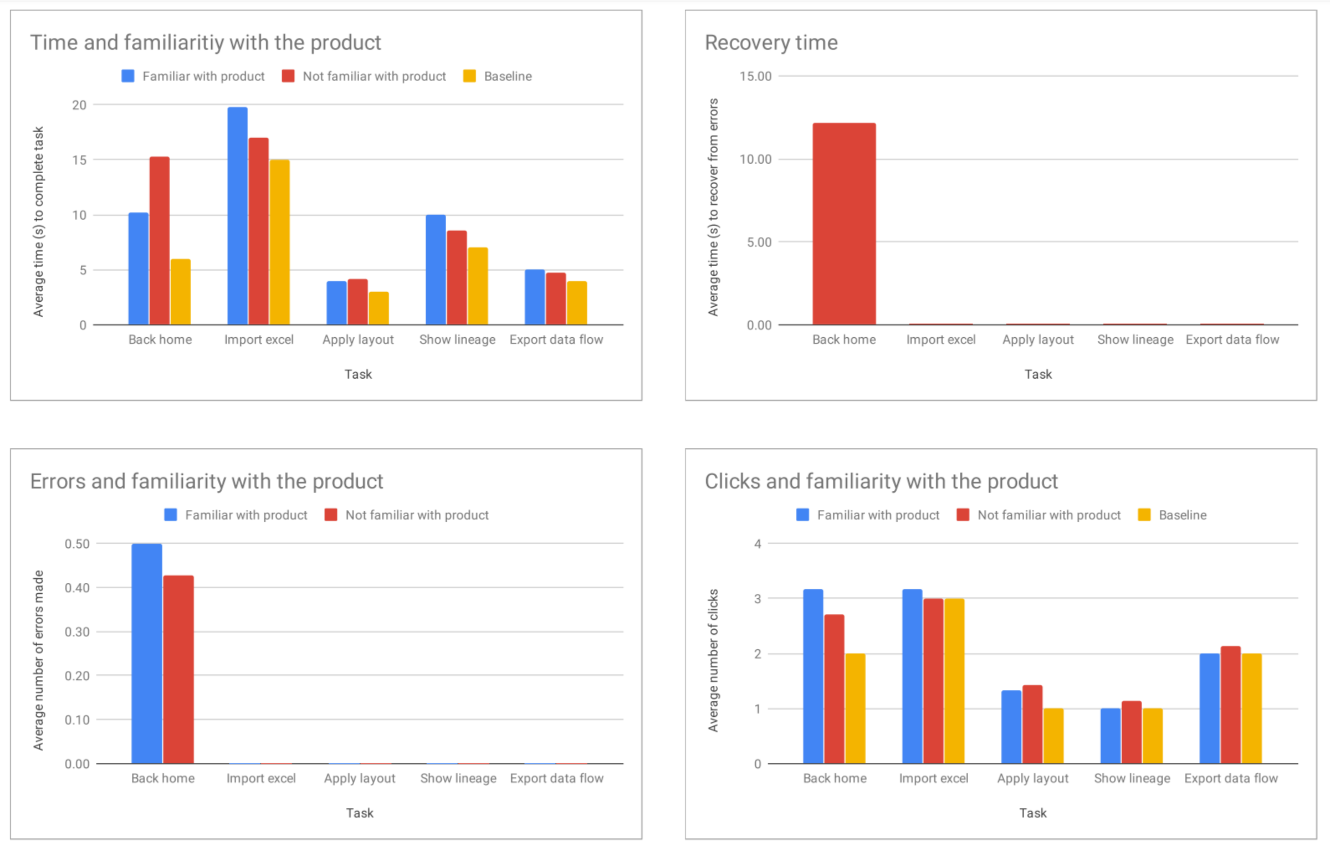
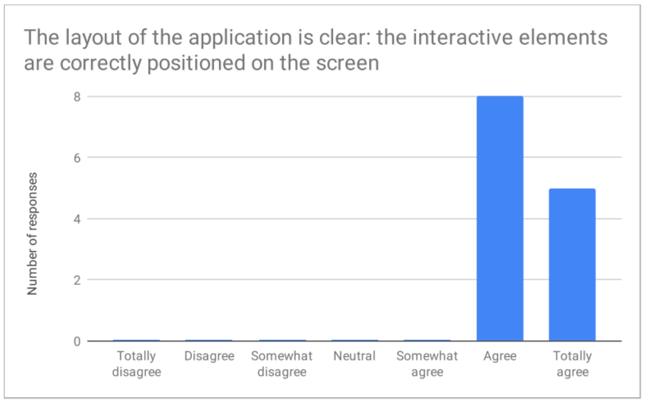
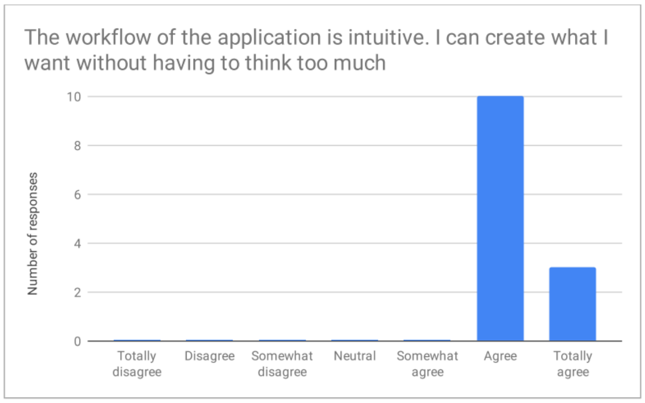


Figure 11: measurement results scenario 2





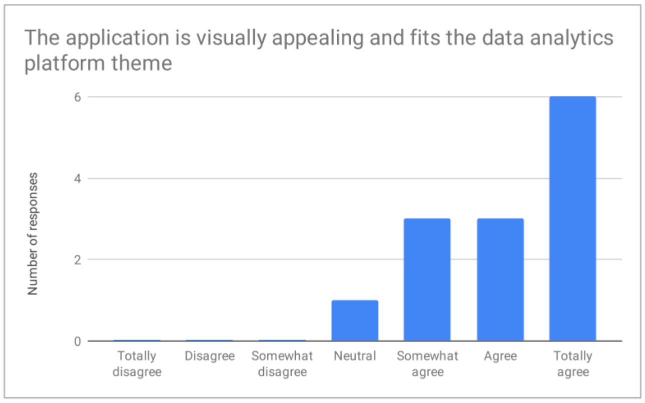


Figure 12: measurement results usability questionnaire

8.2.5 Discussion

We were not able to get a statistically significant large enough test group (n=13), so our results are indicative and should not be seen as conclusive.

The effectiveness rate of scenario 1 resulted in 64.65%. Since this value is lower than the desired 70% (Rubin and Chisnell, 2008), this would indicate that the product should have been more intuitive to use for new users. Creating and deleting links caused the most errors and the highest recovery time 10). We observed through the task of deleting links that people did not think of pressing the delete key. Adding tables involved some searching for tables in the catalog that resulted in errors for some users, but the recovery time remained low. The effectiveness rate of scenario 2 resulted in 70.40%, which is higher than the required 70%. The location of the 'back home' button was not intuitive for both new users and familiar users as it is located in the middle of the toolbar of the product. This issue should be resolved when integration in the data analytics platform is done as a widget after the usability test because its design is based on that of existing widgets. All together, we can conclude that *learnability* is partially solved. It can be fully solved, if creating/deleting links and going back home are made more intuitive.

It appeared that most tasks were performed more quickly by familiar users, which verifies the *efficiency* characteristic. However, we could also see that familiar users performed slower than new users for some tasks. By looking at the video recordings, we could see these users were more eager to explore other functionality during the experiment.

Overall only a couple of errors were made during the experiment. Of course, these are still very useful for improving the usability of the product. Even though some users got stuck for a minute on a task, all users were able to complete all tasks. Therefore we can conclude that the *errors* characteristic has been verified.

There is not any large spread in answers to the questionnaire, and generally, the answers vary between agree and totally agree. So we are quite certain that this speaks in favor of the good usability of the application. Therefore we can argue that the *satisfaction* characteristic has been verified by the questionnaire.

All together, all characteristics have been verified by the usability test except the *memorability* due to a lack of time. Only *learnability* could be more improved using the feedback from the usability test. Nevertheless, this usability study confirms our expectations and indicates that our product, in general, is intuitive and easy to use.

8.3 Utility study

Besides fulfilling requirements and making a product that is intuitive for people, perhaps the most important part of the evaluation is if our application solves the initial problem the client was having. Logically it would seem so as the requirements were based on the initial problem and these have been implemented. However, it is up to the client to answer this question. This section holds a utility study to find out whether the product is what they wanted. With this utility study, we intend to find out which reactions and visions the users have about the product and the problem it is meant to solve.

8.3.1 Expectations

During the final stage of the project, our product has been presented to multiple departments at the client. Additionally, our application was even presented to some (potential) customers of the client. This makes the team believe that the final product is useful and that we support all the short term goals and have support for the long term vision of the client.

8.3.2 Measurements

After the experiment described in subsection 8.2, the users had real experience with our product that had been developed according to their needs defined at the beginning of the project. With that experiment in mind, we added a questionnaire at the end, where we asked them to confirm whether the resulting product fulfills their needs. This questionnaire contained some multiple-choice questions with some additional open questions for further feedback.

8.3.3 Methodology

The questionnaire contained multiple questions that address the usefulness of the product. This was done by first asking the user if he/she thinks that the product is useful in visualizing Data Lineage. Second, to determine the uniqueness of the product that the team developed, we asked whether the product stands out against alternative solutions.

The first four questions could be answered by one of the following options: *Totally disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree or Totally agree.* The last two open questions were meant to let the users provide any additional ideas or features that they would like to see in the future. This information could then be used by the team to analyze the long term vision and the usefulness of the product.

- The application is useful to visualize Data Lineage
- The application offers benefits over other applications like Microsoft Visio or draw.io
- Visualizing Data Lineage (for example using our application) is actually useful for clients and companies. For example, by reducing Risk, providing Data Governance and Compliance
- Overall: the application provides all the features a Data Lineage Editor should have
- If for the previous question you disagreed, please list which features are missing
- Do you have additional feedback for us?

8.3.4 Results

As can be seen from the results in figure 13, most users were very positive about the usefulness of the product. They all agree upon the vision that visualizing data lineage is actually useful for companies and that our product can do this properly. Most users think that the product offers benefits over alternative applications like Microsoft Visio or draw.io. However, the responses to the last question are more spread. Some users think that our product already provides all the necessary functionality. Others think that there are still some interesting features to be implemented. As an example, one user suggested that a 'redo' button would be nice to have. However, most feedback of additional features was about improving the user interface. Some users also provided positive feedback about the usefulness of our product and were eager to see it integrated into their platform. There was one person from the Data Management department who disagreed with the last question since he would like to have seen that the product was able to detect lineage from data automatically. However, as was stated before, there was an agreement between the team and the client in the early stage of the project that this part would be excluded from the product.

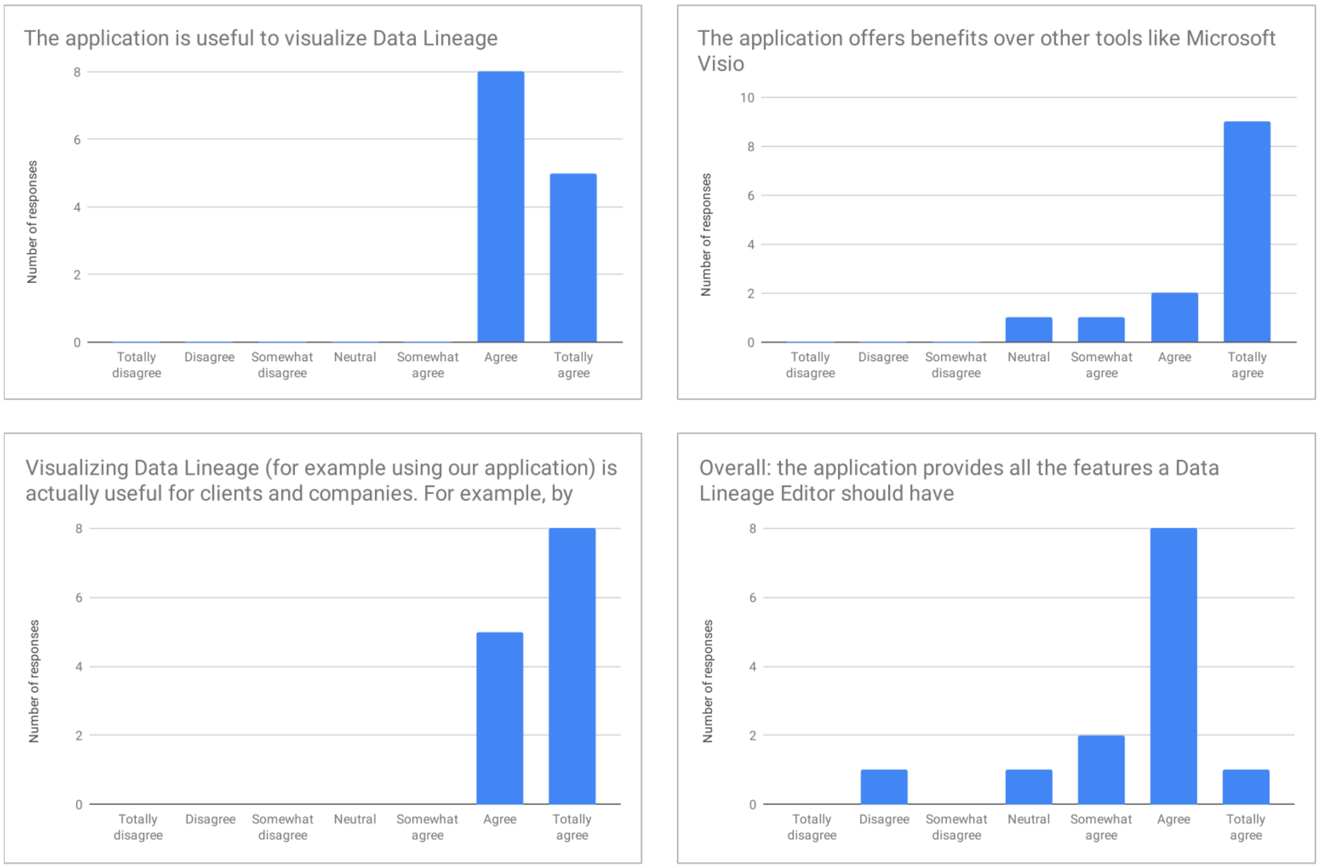


Figure 13: measurement results utility questionnaire

8.3.5 Discussion

The results of the questionnaire overall indicate that the team has made a useful product for the client. Most feedback was very positive. The functionality of the product could be extended with more features, such as adding a ‘redo’ button. However, most suggestions from the users were about improving the usability of the product. Overall, all users were satisfied with the utility of the product. Across these users, some were part of the Data Management department, and the others were part of the platforms department. Throughout the project, the team already noticed some differences in the prioritization of the requirements between both departments. This utility test allowed the team to analyze these differences more clearly. The Data Management department puts more focus on the visualize part of the product (for example the three different view levels were more important to them), and the client’s department puts more focus to the edit functionality of the product. From these results and the results from section 8.2 we could see that all parties were satisfied in the end and most users were optimistic about both the short and long term usefulness of the product.

9 Discussion

The evaluations from section 8 gave insight into how successful the team has been in solving the problem posed by the client. It indicated that the final product contains the desired functionality and that it is intuitive to use. All participants of the evaluation were convinced about the usefulness of the final product. Next to that feedback from both these studies was later on used to improve the user interface of the product. The overall results from the evaluation were positive, and the client showed their satisfaction as well by presenting the product to their customers and other departments. Therefore, the team is convinced that a useful product has been build in the end.

The client is looking forward to the integration of the product into their platform and has provided additional opportunities for the team to work on this.

9.1 Limitations

One limitation of using GoJS, specifically related to auto layout, is that when the input scales, the application becomes slower. The best layout settings become unusable when visualizing more than 1000 nodes on the screen. The faster tree layout option provided, uses a smaller packing [13] option (which is faster, but less clear). This one works okay until 10.000 *nodes*, (takes around 20 seconds), but this is the practical limit where this application is deemed no longer usable by the team.

Another limitation comes from the fact that provided APIs used for requesting data do not support querying by ID. This means that when reading a diagram, all systems tables and attributes have to be read first to resolve the details that have to be shown on the screen. This means that when the data landscape contains millions of (different) tables (not table-entries but actual different types of tables), that the application will use more RAM, and may at some point run out.

9.2 Reflection

The beginning of the project was mainly about understanding the client's problem and what kind of product they had in vision. The team had to communicate with many people from different departments to gain more insight into the use case of the product and therefore what features were required.

During development, there was a need for real-world sample data (figure 1) to clarify to the team what the use cases of the application would be. However, due to circumstances, this sample data was only handed over to the team after a couple of weeks. To solve this the team decided to generate sample data for those weeks so that the development could get started. In hindsight, the team could have put more effort into asking for this sample data to be able to define the problem definition and analysis earlier. Throughout the rest of the project, the team planned things carefully, so that resources would be delivered in time and development was not halted. Because of this, communication was one of the most important aspects of the project.

During the utility study 8.3.5, the team noticed some minor differences between the visions of the different departments of the client. These differences clarified the use case of the product in even more detail. In retrospect, it would have been better to perform an extra study at the beginning of the project to gather

[13]`https://gojs.net/latest/api/symbols/LayeredDigraphLayout.html#static-PackMedian`

the conceptual ideas that people had about the product. Resulting in a better understanding of the use case of the product.

9.3 Ethical Implication

The application built during this project acts as a tool for creating diagrams and thus does not have any direct ethical implications. However, some of the data lineage diagrams created with the application provide proof that organizations are compliant with regulations (like GDPR, Solvency II, or HIPAA). Such laws and regulations heavily revolve around ethics pertaining to balancing liberties and rights. In this sense, the application serves as a purpose of providing documents that are used during ethical rulings. It is important that our application does not obstruct such a process. We argue that by the nature of resolving problems that would lead to inaccurate diagrams, our application serves an ethically just purpose.

9.4 Recommendations for the client

It was a great experience to work with the client as they showed a lot of interest in the product and invested a large amount of time in us. To make sure future project provide an even more seamless experience, we have two recommendations:

- For the client's data analytics platform, we recommend improving the speed of their external APIs. Outside of their network, the response time of the API was not fast enough to store changes in real-time.
- For future projects that work with data; we recommend preparing some data samples at the start of the project. This way, the process of designing the application and its real-world constraints is easier and more efficient. Issues that would not occur in a development scenario, but might arise during a in a real-world scenario would be found much sooner.

10 Conclusion

In this report, we discussed the use of data lineage, both from a technical point of view as well as from a business point of view. However, the problem existed that there was no easy way to map out this existing data lineage.

We've set the goal of our project to build an application that can visualize and edit data lineage diagrams to address the initial problems.

To solve this, we conceptualized a solution around which we designed and implemented our product. By using our product, the client is now able to both view and edit existing data lineage diagrams as well as create new diagrams from scratch. The product also allows the client to find out the source(s) of, for example, attributes by automatically visualizing the lineage trace. After evaluating both the technical details and the usability/utility of our product, we can conclude that it is an appropriate solution for the client's problem.