

MASTER

Improving maintainability of data transformation graphs using visualization

Andriessen, R.

Award date:
2019

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Improving maintainability of data transformation graphs using visualization

Master Thesis

R. Andriessen

Supervisors:

prof.dr.ir. J.J. (Jack) van Wijk

dr. ir. R. (Roeland) Scheepens

dr. ir. D. (Dirk) Fahland

1.0

Eindhoven, December 2018

Abstract

Companies often integrate their data by extracting information from different sources and applying various transformations to receive their desired output (also referred to as ETL). Maintenance on the transformation graphs applying these transformations is performed by a small number of domain experts that often have insufficient tools to do so. We introduce a novel approach that improves the maintainability of transformation graphs by enabling domain experts to recreate their own mental models using a graph hierarchy. By presenting this hierarchy to others through a mental-map-preserving layered graph layout, users can collaborate on a unified understanding of their transformation graph, enabling them to iteratively improve its design and quickly correct errors. Using two real-world use cases and a qualitative analysis, we found that collaboration on a unified mental model is very effective at improving maintainability. Additionally, although finding errors has an initial learning curve for some users, it saves all of them a lot of time.

Preface

This thesis presents my work on *Improving maintainability of data transformation graphs using visualization* at ProcessGold. It represents the last stage of my master's degree in Computer Science and Engineering, provided by the Eindhoven University of Technology.

I would like to thank Jack van Wijk for his extensive support, sharp comments, and detailed feedback on my work, and Roeland Scheepens, for his patient and enthusiastic day to day supervision, feedback, and support. I have learned a lot from both of you and am certain my research would not have been possible without either of you. I would also like to thank Dirk Fahland, for being part of the evaluation committee.

I would like to thank my parents, for all the interest and support they have given me throughout my study.

And finally, I would like to thank all of my colleagues at ProcessGold for their time and feedback on my research and the amazing (sometimes distracting) atmosphere they provided.

Contents

Contents	vii
1 Introduction	1
1.1 Thesis scope & Approach	2
1.2 Thesis structure	2
2 Preliminaries	3
2.1 Data flow	3
2.1.1 Data flow diagrams	3
2.1.2 Data transformation graphs	3
2.1.3 Relational algebra	4
2.1.4 Transformations	5
2.2 Data lineage	6
2.2.1 Dependency graphs	6
2.3 Graph layouts	8
2.3.1 The Sugiyama framework	8
2.4 Related work	9
2.4.1 Approaches of visualizing data transformation graphs	10
2.4.2 Hierarchies	12
2.4.3 Set visualization	12
3 Problem definition	15
3.1 Background: ProcessGold	15
3.2 Stakeholders	15
3.3 Task analysis	16
3.3.1 Presentation	18
3.3.2 Debugging	19
4 Approach	21

CONTENTS

4.1	Mental models	23
4.1.1	Graph hierarchy	23
4.1.2	Hierarchy navigation	23
4.1.3	Attribute aggregation	24
4.2	Graph layout	26
4.2.1	The dot layout	26
4.2.2	Mental maps	26
4.2.3	Layered graph layout	27
4.2.4	Hierarchical layout	29
4.2.5	Edge bundling	31
4.3	Hierarchy view	35
4.3.1	Representing the hierarchy	35
4.3.2	Table nodes	35
4.3.3	Group nodes	37
4.4	Debugging views	41
4.4.1	Column analysis	41
4.4.2	Dependency views	43
4.5	Interaction	46
4.5.1	Navigation	46
4.5.2	Selection	46
4.5.3	Selections in the debugging views	48
5	Evaluation	51
5.1	Case studies	51
5.1.1	Case study 1: Creating a proposal & presenting changes to a customer	51
5.1.2	Case study 2: Debugging	52
5.2	Qualitative evaluation	54
5.3	Conclusions & Discussion	57
6	Conclusions	59
6.1	Future work	60
	Bibliography	61

Chapter 1

Introduction

Many companies nowadays work with large and complex data. To make this data useful, elaborate pre-processing is often required. Such pre-processing is defined via large complex networks of transformations, which are applied to the data to transform them to the required format. A common method by which these transformations are performed is the Extract-Transform-Load (ETL) principle. It involves three steps: extracting information from external data sources, transforming this data based on business logic, and, finally, loading this data into other applications or data storage systems. These networks of transformations are also called *data transformation graphs*. Much research has been done on visualizing these graphs, focused on reproducibility in a scientific context [CFS⁺06, VLFR17] or performance analysis [MH⁺15], with a means of presentation as support.

Visualization tools designed for data transformation graphs are widely available in the industry. Some focus on error checking and visual creation of these graphs [Clo], while others provide performance measurement capabilities or allow users to write custom code [Apa, The17]. However, these solutions do not focus on tracking down errors or understanding the graphs. Instead, they assume data transformation graphs are very small and are therefore easily understood, or requirements are reasonably static. In reality, both of these assumptions fail to hold. Large data transformation graphs are not uncommon, therefore, assuming users have a complete understanding of them is unrealistic. Additionally, requirements on the format of the output and input data constantly change. Hence, maintenance is required, a cumbersome task, prone to error.

The large and complex data transformation graphs that require maintenance are usually built with commercial or custom-made tools, making data gathering hard. Additionally, the users affected are relatively small groups of domain experts. These are both likely reasons for the lack of research on the topic in the visualization field.

Maintenance of data transformation graphs relies on the domain experts that have developed them. Since current approaches are limited in their ability to find errors, domain experts require a deep understanding of the inner workings of the data transformation graphs to resolve errors. However, as with code maintenance, domain experts and requirements constantly change. Therefore, presenting data transformation graphs to others becomes crucial. Visualization could be a solution to improve the maintainability of data transformation graphs.

1.1 Thesis scope & Approach

To find out how and if visualization can help in improving the maintainability of data transformation graphs, we address the following research question in this thesis:

How can we use visualization to improve the maintainability of data transformation graphs?

To answer this question we look at a widely cited, and repeated study by Lientz and Swanson [LS80]. They found maintenance can be categorized into four categories:

- Perfective: Changing or new requirements (51% of the time);
- Adaptive: Changes in the software environment, e.g. changes hardware or the operating system (24% of the time);
- Corrective: Error correction (22% of the time);
- Preventative: Prevent future problems by improving maintainability (3% of the time).

As transformation graphs are often created with commercial tools, adaptive changes are primarily a problem for the software developers of these tools. Due to adaptive changes being much rarer, adaptive maintenance is not as relevant for transformation graph maintenance. Therefore, we keep adaptive changes out of scope of this thesis. For each of the other categories, we define a sub-research questions:

RQ1 How do we enable users to achieve a common understanding of the transformation graph such that domain experts can quickly and correctly respond to new or changing requirements? (Perfective)

RQ2 How do we enable domain experts to find errors in their transformation graphs? (Corrective)

RQ3 How do we enable domain experts with the means to improve their data transformation graph design? (Preventative)

To address these questions, we have developed a visualization that enables users to present and debug data transformation graphs more effectively. This project is done in co-operation with ProcessGold, a software company that develops a platform upon which applications for data visualization are created. Each application defines a data transformation graph that processes data set from external parties before being visualized. These data transformation graphs and the data lineage of the data sets are used as input data for our visualization. Throughout our design process, we received feedback from domain experts, adjusting the visualization to suit their needs as best as possible.

1.2 Thesis structure

The rest of this thesis is structured as follows: First, we discuss data flow, data lineage, and other preliminary knowledge and establish the notation used in the thesis in Chapter 2. Next, we discuss our problem in more depth by analyzing stakeholders at ProcessGold, their problems and the resulting requirements in Chapter 3. Then, we discuss our approach in Section 4. Then, we evaluate our approach using two real-life use cases and a qualitative analysis in Chapter 5. Finally, we present our conclusions and discuss future work in Chapter 6.

Chapter 2

Preliminaries

Our approach has its basis in data flow, data lineage, and graph layouts. We describe these and introduce the notation used throughout the thesis and end by discussing related work in the following chapter.

2.1 Data flow

The term data flow has various meanings based on context. For example, it describes how values are computed in a piece of software when used in the context of data flow analysis. Alternatively, in the context of data flow diagrams, it represents an abstract description of the creation of data. In this thesis, we consider data flow as in the context of data transformations.

2.1.1 Data flow diagrams

Data flow diagrams are a widely adopted method to visualize the flow of information within a system, originally proposed by Stevens *et al.* [SMC74]. They are defined by four components:

1. **External entities** are systems outside of the data flow diagram, being either inputs (e.g., files or databases) or outputs (e.g., external applications or persons).
2. **Processes** represent anything that changes data, performs computations, or applies a form of logic onto the data.
3. **Data stores** represent data stored for later use, such as files or tables in a database.
4. **Flow** represents the route taken by data between external entities, processes, and data stores (often represented by arrows in between them).

For these diagrams, several standard notations have been proposed [GS77, DeM79, CY91]. However, they often remain hand-drawn, abstract representations of the data flow as the size of these graphs would become too large to understand if all details were presented.

2.1.2 Data transformation graphs

In the 70/80s, as more data became available in businesses, the need for data integration grew. Several principles arose to tackle this issue, the most well-known being Extract, Transform, Load (ETL). These

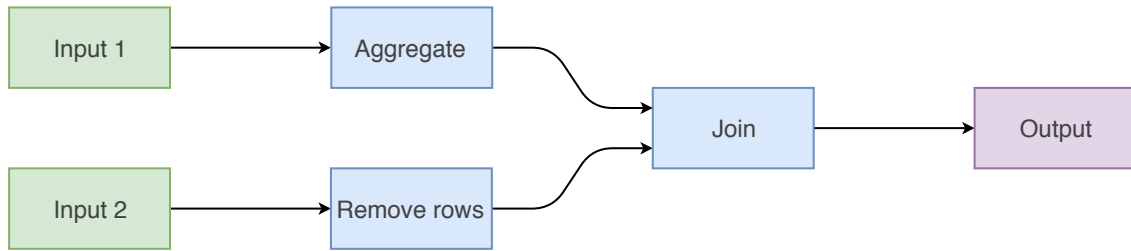


Figure 2.1: A data transformation graph, with two data sources (●), three transformations (○), and one data sink (●).

principles describe the flow of data between external input and output entities, and through the transformations between them. These transformations are applied in sequence or acyclic graphs known as *data transformation graphs* [CW03]. A standard definition of these graphs is not prevalent in research. However, we define them as data flow diagrams without data stores, and processes being renamed to transformations.

More formally, a **data transformation graph** $G = (V, E)$ is a directed acyclic graph, where every node $n \in V$ is either an external entity or a transformation, and an edge $(n, n') \in E$ indicates data flows from n to n' . Additionally, we define a **data source** as an external entity that exclusively produces data (e.g., a database table or a file) and a **data sink** as an external entity that exclusively consumes data (e.g., a visualization or a database table). Finally, a **transformation** consumes data from one or more data sources or transformations, combines, transforms, in some cases temporarily stores, and passes it forward. An example of a data transformation graph is shown in Figure 2.1.

2.1.3 Relational algebra

Relational Database Management Systems (RDBMS) are the most common examples of systems where data transformation graphs are used. RDBMS's have their basis in relational algebra, a formalism created in 1990 by Codd [Cod90], which has its roots in set theory. It is used for modelling and querying data and is heavily used in applications through the widely adopted SQL standard [ISO]. Companies typically use thousands of these queries and base a large portion of their data transformation graph on relational algebra.

Relational algebra is based on five primitive operators:

- **Select** returns a subset of the rows of a table;
- **Project** returns a subset of the columns of a table;
- **Union** returns the combination of the rows and columns of two tables;
- **Difference** returns the difference of rows between two tables;
- **(Cartesian) product** returns the cross product of two tables;

and three additional operators:

- **Intersection:** returns the overlapping rows of two tables;
- **Join:** returns the cross product followed by a select on two tables;
- **Divide:** returns the values of the relations not paired with all values of another relation.

All of these are also shown in Figure 2.2.

Data transformation graphs built upon relational algebra perform one or multiple of these operators on a given data set, often expressed via a query. Relational algebra can express a wide variety of transformations, however, it cannot perform all of them [CW03]. Therefore, we cannot base our visualization completely on this theoretical foundation.

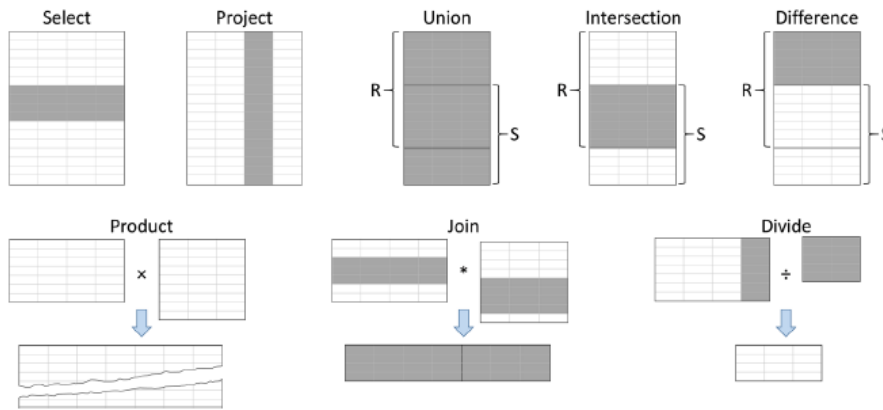


Figure 2.2: An overview of the relational algebra operators as defined by Codd [Cod90]

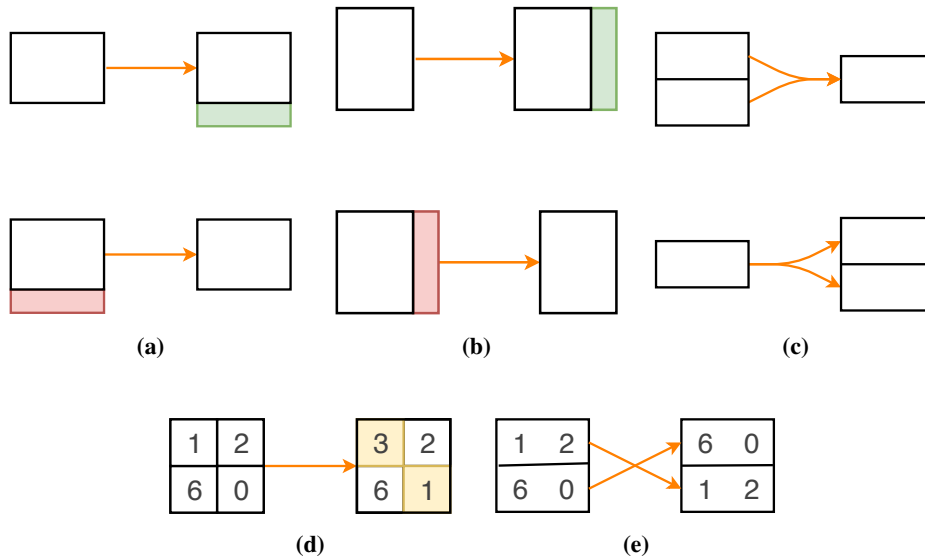


Figure 2.3: The four different types of transformations which can be applied to tables: structural (a, b), merge and duplicate (c), content (d), and order (e).

2.1.4 Transformations

Data transformation graphs are primarily build on relational database systems providing tabular data, a broadly adopted method of storing data. Therefore, we assume external data sources and transformations produce tabular data. On tables, there are four different types of transformations that we can apply (see Figure 2.3) [NSH⁺18]:

1. **Structural changes:** adding and removing rows (Figure 2.3a), or adding and removing columns (Figure 2.3b);
2. **Merge and duplicate changes:** a single row or column can be duplicated. Alternatively, multiple rows or columns can be merged into a single one (Figure 2.3c);
3. **Content changes:** a set of values inside of cells changes (Figure 2.3d);
4. **Order changes:** rows are reordered (Figure 2.3e).

We can represent a content change as two structural changes, by adding a column with the resulting values and removing the old column. Therefore, we leave content changes out of the scope of this thesis. Additionally, we do not consider order changes, as these rarely impact the result of transformations as most transformations do not rely on the order of data to allow for incremental processing. Therefore, the focus of this thesis will lie on assisting users in completing tasks concerning **structural** and **merge and duplicate** changes.

2.2 Data lineage

Data lineage is an encompassing term used for data that describes the data life cycle including its origin and change over time, i.e., it describes how the data actually flows through the data transformation graph. Data lineage is also referred to as why-provenance [De12,HDBL17], first defined by Cui *et al.* [CW03].

The literature discusses two levels of data lineage granularity: coarse-grained (or schema level) [BB99, LBM98, HQGW92, RS98] and fine-grained (or instance level) data lineage [CW03]. Coarse-grained data lineage involves tracking dependencies on a schema level, i.e., tracking column dependencies. Fine-grained data lineage involves tracking individual rows, i.e., row dependencies. We discuss these in more detail in the following section.

2.2.1 Dependency graphs

Lineage information of tables can be modelled via three different dependency graphs:

- **Table dependencies** follow directly from the data transformation graph. Therefore, these dependencies are already available without additional data lineage information;
- **Column dependencies** represent relations between columns. These can be internal in a single table (e.g., creating a new column based on a set of others) or between tables (e.g., in a join).;
- **Row dependencies** provide detailed information on the dependency between single rows.

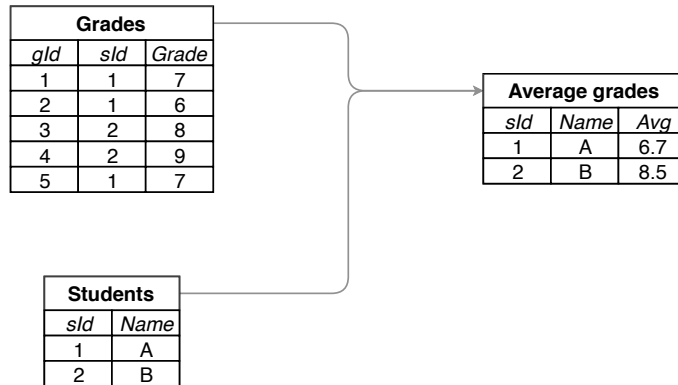
In this thesis, we assume we have all three types of dependencies available. Methods for computing data lineage for relational algebra transformations and generic transformations are given by Cui *et al.* [CW00, CW03].

We consider an example of the different types of dependency graphs. Consider two tables: one with **students** and one which contains their **grades**. We want to compute the average grade for each student and place the result into the **Average grades table**. We apply a transformation that groups the grades by student and computes their average, and copies the name and ID of the student. The resulting dependencies of this are as follows:

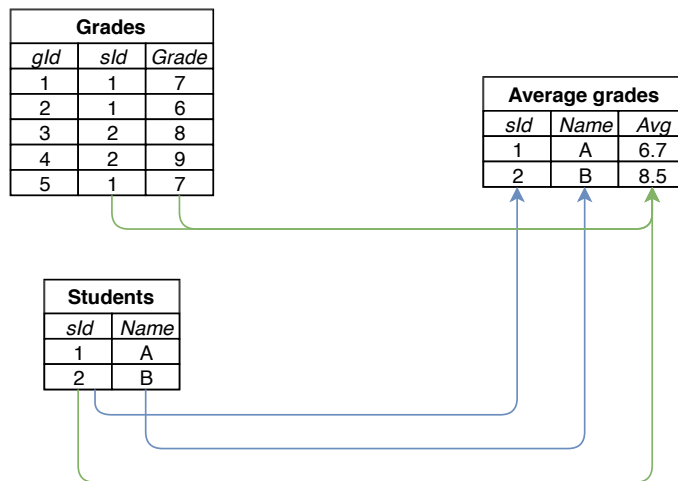
Table dependencies: the *average grades* table depends on the *students* and *grades* table (See Figure 2.4a).

Column dependencies: We first group rows by student id (*sId*) and compute the average of the grades. We then match the *ids* with the student id from the students table. Therefore, the average grade column depends on all three of these columns. Additionally, we copy both the student id and name for each students, hence these have a dependency as well (See Figure 2.4b).

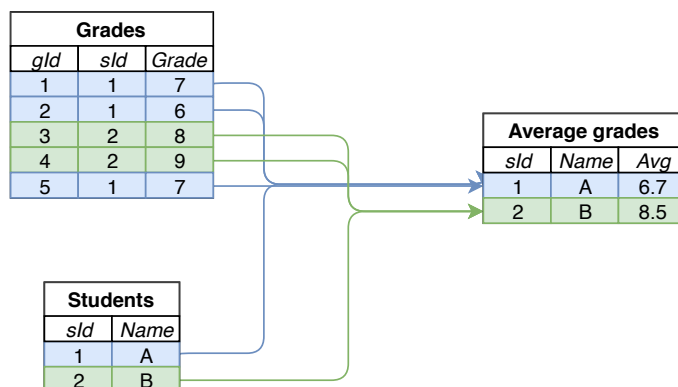
Row dependencies: every row in the average grade table uses one row from the student table and multiple rows of the grades table (See Figure 2.4c).



(a) **Table dependency graph**, or data transformation graph G . Each line indicates a table dependency.



(b) **Column dependency graph** The green arrows (●) indicate the dependencies required to compute the average, the blue lines (●) imply a copy.



(c) **Row dependency graph** Each line indicates a row dependency. Every color refers to a unique student id. Hence, in this example, there are four students.

Figure 2.4: The three different types of dependencies of a transformation using the **Grades** and **Students** table to produce an **Average grades** table.

2.3 Graph layouts

The layout of a graph has a high impact on its readability and understandability, therefore, we carefully look at graph layout techniques in this thesis. A large amount of work has been done on graph layouts, ranging from force-directed layouts [KK89, FR91] to layered graph layouts [BM01, PNK11, SM91]. In this thesis, we use a layered graph layout for their fast computation and to emphasize nodes as steps in a (transformation) process. The most established layered graph layout is the Sugiyama layout [STT81].

A layered graph layout works by splitting the graph into several layers. These layers are also referred to as ranks and determine one dimension of the node position (the x-coordinate in horizontal layouts or the y-coordinate in vertical layouts). For every layer, a linear ordering of the nodes is determined. This order constrains the positions of the other dimension.

2.3.1 The Sugiyama framework

Given a directed graph, the Sugiyama framework describes a set of readability requirements and computes a layout subject to them:

1. Nodes are drawn without overlap. All edges point in a single direction.
2. Short edges are drawn with straight lines.
3. Long edges are drawn as close to straight lines as possible.
4. The number of edge crossings is minimized.
5. Nodes connected to one another are as close as possible.
6. The edges that attach to a node are balanced, meaning they should be evenly spaced around the node.

Although the first two requirements are easily met, requirements 3 through 6 are harder, as they are difficult to compute and often conflicting, resulting in approximated solutions. To compute a layout subject to these requirements, the Sugiyama framework describes the following steps (see Figure 2.5 for a visual representation):

1. **Pre-processing** involves the following steps:
 - (a) **Cycle removal** is the first step in dealing with graphs with cycles, some edges are flipped in order to obtain a directed acyclic graph;
 - (b) **Computing a node rank**. The layers on which the nodes are positioned (ranks) are computed. We denote the rank of a node n by $rank(n)$ and define the set of nodes on rank i as R_i ;
 - (c) **Dummy nodes are placed** to ensure that every edge $(v_i, v_j) \in E$ has that $rank(v_i) + 1 = rank(v_j)$;
2. **Computing node order**. All nodes are placed in a linear order minimizing edge crossings. This so called crossing minimization is extensively researched [EW94, BM01]. We indicate the order of a node n by $\sigma(n)$;
3. **Computing node positions** The positions of each node are computed, adhering to the previously described requirements;
4. **Drawing the graph** Dummy nodes are removed and the remaining nodes and edges are drawn. In some approaches, edges are replaced by smooth splines.

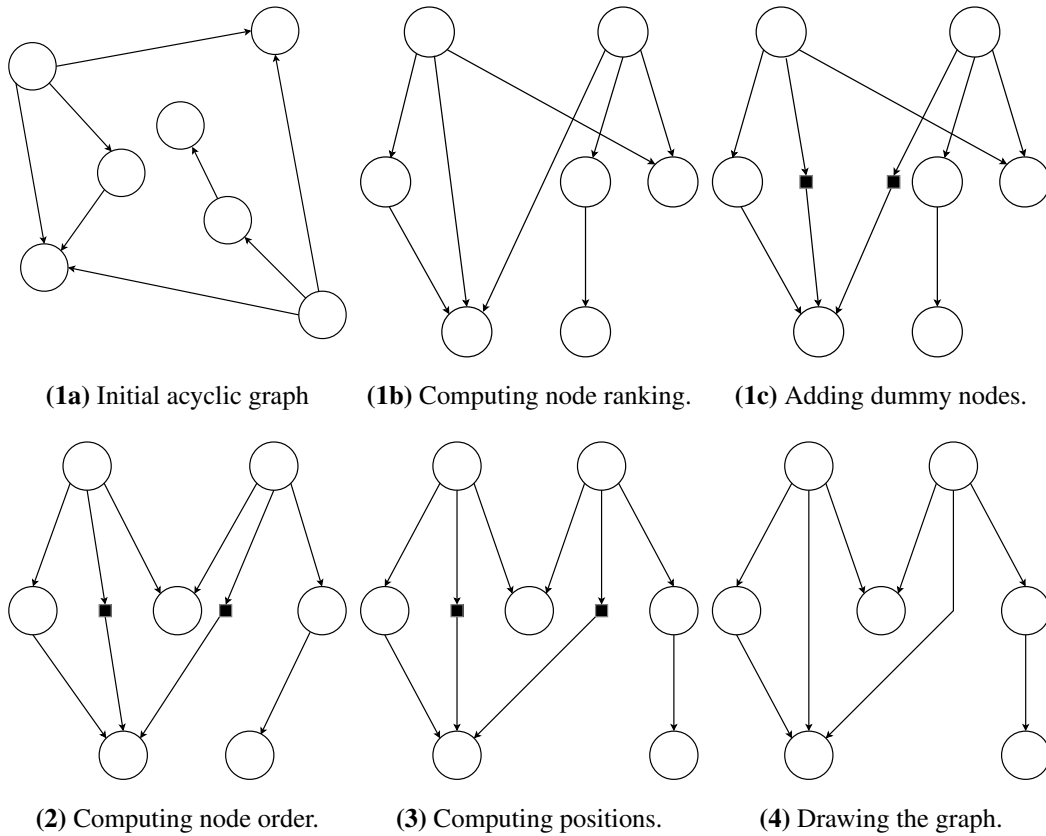


Figure 2.5: A visual representation of a graphs layout being computed according to the Sugiyama framework.

2.4 Related work

We discuss the state of research on data lineage and data flow in this section, as these subjects are closely intertwined. Outside of the visualization field, Herschel *et al.* [HDBL17] provides an extensive survey, discussing uses for lineage information, what forms exist, and techniques are commonly used for a given set of requirements. To establish an overview of research in the visualization field, we provide a subdivision based on the tasks users are enabled to perform in Table 2.1. Before most of these tasks can be performed however, data lineage is required. Yocum *et al.* [De12] describes how data lineage can be computed in distributed systems and how this could be used for debugging. While other research describes how lineage can be computed in relational database systems [GA09, VC07, CW00] or in generic transformations [CW03].

Commercial tools always have some form of presentation, other than that, there is a split: they focus on data quality, reproducibility, or attribution and debugging. Most research focuses on presentation and reproducibility, although there is some on debugging. Most approaches enabling presentation tasks limit themselves to small transformation graphs or use it as a method to enable other tasks. Those that do not, usually let users replace parts of the graph, creating an abstract representation [Clo, SLSG16, AKN16], however, these features are often lacking or require much effort on the users part.

Both research and commercial tools provide only limited debugging functionality, either they exclusively allow for debugging on columns (referred to as schema level or coarse-grained provenance) [Man, ACT06], provide debugging on small sets of rows, usually in a non-visual way [Inf], or only provide users with the raw data in intermediate steps [Clo].

Task description	Research	Industry
Presentation: Information is being transferred between persons or between the visualization and a person.	[CFS ⁺⁰⁶ , RHF05, WSW ⁺¹⁸ , MHHH15, AWMK17, YS17, HPvD11, BYB ⁺¹³ , OAF ⁺⁰⁴ , KOF ⁺¹⁶]	[Apa, Clo, Inf]
Reproducibility: assist users in reproducing similar behavior.	[CFS ⁺⁰⁶ , AKN16, SLSG16, OAF ⁺⁰⁴]	[The17, Apa]
Data quality: enable users to filter or correct invalid data, validation of data and applying business logic.		[Clo, Alo]
Attribution: enable users to find who/what created certain data.	[RHF05]	[Man, Inf]
Performance: find bottlenecks and other performance problems.	[MHHH15]	[Apa]
Debugging: allow for finding errors in data transformations.	[WSW ⁺¹⁸ , HPvD11, ACT06]	[Man, Inf]

Table 2.1: Subdivision of research and commercial industry tools, based on the tasks they perform.

Approach	Research	Industry
Node-link diagrams	[MHHH15, WSW ⁺¹⁸ , AKN16, HPvD11, AWMK17, SLSG16, ACT06, KOF ⁺¹⁶ , OAF ⁺⁰⁴]	[Clo, Apa, Inf, The17]
Sankey	[RHF05]	[Alo]
Matrix	[NSH ⁺¹⁸]	
Radial	[BYB ⁺¹³]	
Others	[YS17]	

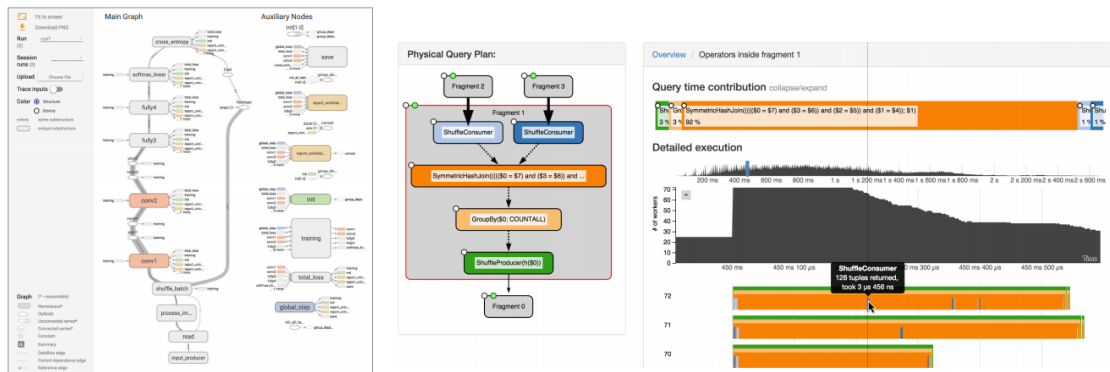
Table 2.2: Subdivision of research and related work the approach used for visualizing data transformation graphs.

2.4.1 Approaches of visualizing data transformation graphs

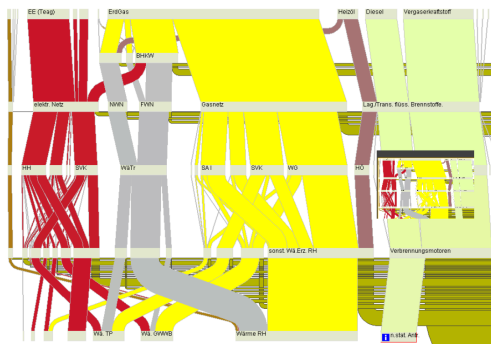
Matrices and node-link diagrams are the two most common representations of graphs. Ghoniem *et al.* [Gho04] compared these two approaches and found that matrices work best for finding neighboring nodes in large graphs and performing path based tasks in high-density graphs. In small, low-density graphs, such as data transformation graphs, however, node-link diagrams were preferred. Therefore, it is not unexpected that the visualization of data transformation graphs is usually done using node-link diagrams (see Table 2.2). Although other approaches have been proposed, these lean on specific use cases such as file systems [BYB⁺¹³], or show only high-level representations of data flow [Alo] without the means to increase the level of detail.

Node-link diagrams

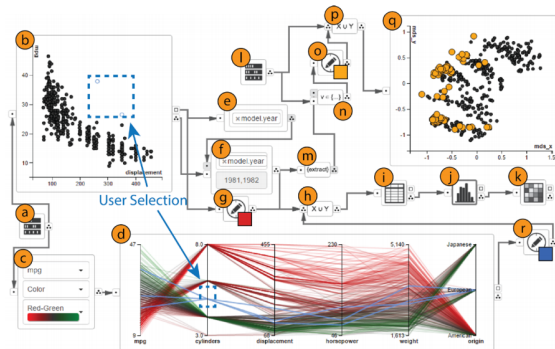
Examples of node-link diagrams being used in data flow are shown in Figure 2.6. Wongsuphasawat *et al.* [WSW⁺¹⁸] (see Figure 2.6a) discuss how to visualize deep learning models in TensorFlow. They work with large, dense graphs, therefore, they extract non-critical operations (loosely connected subgraphs) and place a small glyph adjacent to nodes connected to these subgraphs. They provide a second view in which these extracted subgraphs are displayed. On the remainder of the graph, they build a hierarchy in order to reduce clutter and improve stability of their graph using edge bundling. Moritz *et al.* [MHHH15], discusses Perfopticon, a visualization to analyze the performance of query plans on distributed networks. They distinguish different operations via color, provide detailed information on the distribution of work across the machines, and group parts of the graph to allow users to open and close them on demand.



(a) TensorFlow visualization by Wong-suphasawat *et al.* [WSW⁺18] (b) Perfopticon, a query analysis and visualization tool by Moritz *et al.* [MHHH15]



(c) Riehmman *et al.* [RHF05] build a hierarchy on a Sankey diagram, and allow users to interactively increase the level of detail.



(d) VisFlow, by Yu *et al.* [YS17] uses a node link-diagram where each node can be represented as a visualization.

Figure 2.6: Examples of data flow visualizations in the literature.

Callahan *et al.* [CFS⁺06] also uses a node link-diagram in VisTrails, a visualization designed to help researchers produce visualization pipelines faster and with fewer errors. Unlike systems like VTK [SW], VisTrails provides provenance information to enable researchers to remove the burden of recalling previous input data of their pipeline, allowing them to reproduce previous results easily. They accomplish this by storing data transformation graphs of previous runs in a tree, where each node represents one of these graphs.

Yu *et al.* [YS17] designed VisFlow, they use a node-link diagram, however, they allow users to combine visualizations with a transformation node, allowing users to visualize intermediate steps of their data. They also provide brushing and linking, allowing selections to propagate through the graph (see Figure 2.6d).

Hermans *et al.* [HPvD11] is one of the few that discuss the presentation of fine-grained lineage. They enable users to create a hierarchy on a node-link diagram based on cells in a spreadsheet. They perform an extensive case study, testing their visualizations ability to transfer knowledge between parties, conclude it works well for this purpose and enables users to communicate the reasoning behind their spreadsheet.

Other approaches

Some approaches do not use node-link diagrams. For example, Riehmman *et al.* [RHF05] uses interactive Sankey diagrams to explore complex data flows. However, they also aggregate their diagram into a

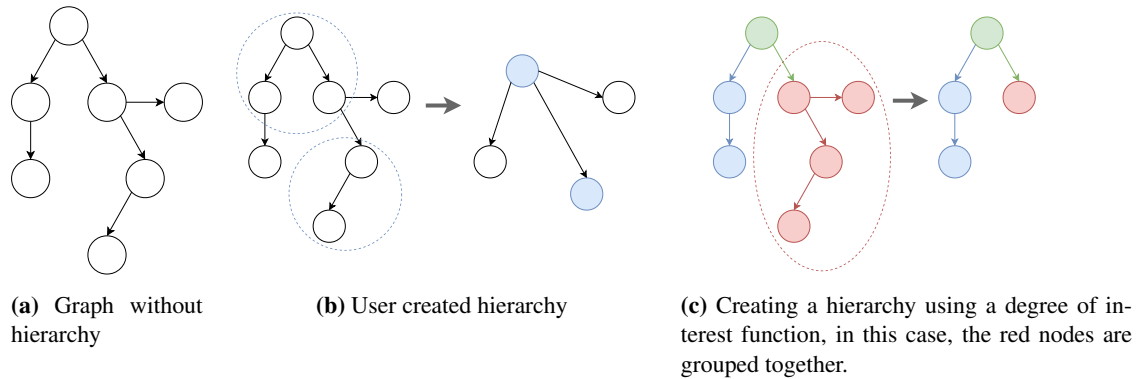


Figure 2.7: Two methods of creating graph hierarchies.

hierarchy, and show a low detail version while allowing users to interactively increase this level of detail. Additionally, users can trace edges through the entire diagram (see Figure 2.6c). Alternatively, Niederer *et al.* [NSH⁺18] describe how they visualize changes in tables over time. They do this by aggregating and showing the difference between two tables in the form of a matrix representing the table structure. Finally, Borkin *et al.* [BYB⁺13] describe how they visualize the data lineage of file-systems. They compare their radial visualization with a previously created node-link diagram designed for the same purpose and conclude that, after users received a period of training, their radial approach worked better.

2.4.2 Hierarchies

When creating visualizations, providing an overview is crucial for many user tasks [Shn03]. To provide an overview in graphs, nodes are often aggregated into a hierarchy by users [AMA08, WSW⁺18] (see Figure 2.7b) or through a (degree of interest) function [SLSG16] (see Figure 2.7c). Elmquist and Fekete state that using aggregation, a summary of the attributes of nodes grouped in these hierarchies can be provided and propose several navigation methods that enable users to request more detail when needed [EF10].

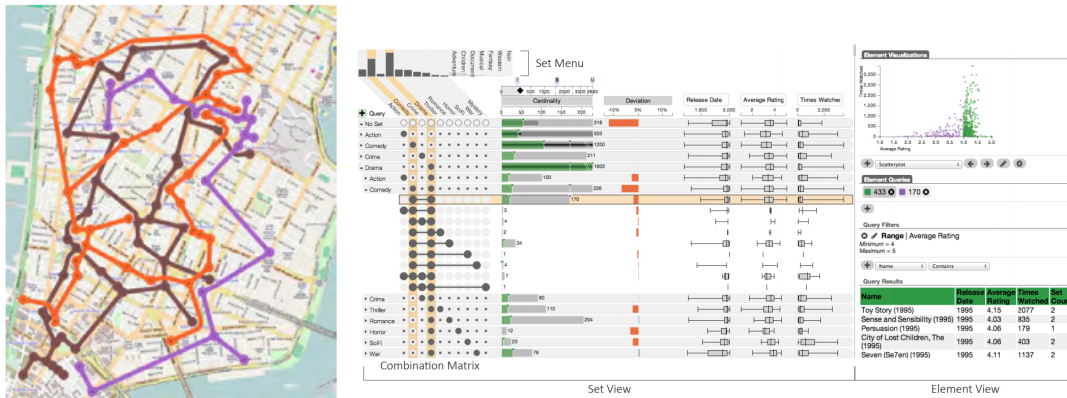
In hierarchies based on node-link diagrams, we can improve the users' mental map [MELS95] by bundling edges, improving stability [BD07, WSW⁺18], and through staged transitions and change highlighting, allowing users to identify, track and understand changes better [BPF14].

2.4.3 Set visualization

Since node-link diagrams are most broadly understood, and hierarchies are commonplace, we discuss techniques for visualizing groups structures in node-link diagrams. A survey done by Vehlow *et al.* [VBW15] summarizes techniques based on the type of group structure. Some of these techniques are designed for the use in spatial graphs, such as Bubble Sets [CPC09] and Kelp diagrams [DvKSW12, MRS⁺13] (See Figure 2.8a). However, these solutions, especially Kelp Diagrams, are expensive to compute. There are also solutions which target node-link diagrams without spatial data, such as BranchingSets [PEF16].

Set visualization is not limited to group structures and can be used for glyphs summarizing attributes of nodes, or summarizing information about the graph topology [IMMS09] There is a broad range of types which can be used, such as Euler, Venn, and Linear diagrams [RD10, Bar69, RSC15], a summary and comparison of which is discussed by Chapman *et al.* [CSR⁺14].

Finally, Lex *et al.* [LGS⁺14] (See Figure 2.8b) created UpSet, a visualization tool to understand relationships between sets. They enable users to explore specific sets and describe a technique of showing attributes



(a) Kelp diagrams by Dinkla *et al.* [DvKSW12]

(b) UpSet by Lex *et al.* [LGS⁺14]

Figure 2.8: Examples of set visualizations in the literature.

related to each possible combinations of the selected sets in a row based visualization.

Chapter 3

Problem definition

To gain a better understanding into the maintenance of data transformation graphs, we undertake this project in co-operation with a company called ProcessGold. We interview several stakeholders to find out where their difficulties lie when maintaining their data transformation graphs. From these interviews, we create a set of tasks and corresponding requirements.

3.1 Background: ProcessGold

ProcessGold is a software company that develops a process mining and visualization platform used to create applications. Each application defines a data transformation graph to processes data according to the ETL principle. First, data is extracted from external files or databases, after which it is transformed using a domain specific language having its basis in relational algebra, before being loaded into visualizations.

When application developers understand the transformations well, they are more confident in the correctness of their visualizations and have to spend less time on maintenance. Therefore, understanding what happens in the transformations of the data transformation graphs is crucial for both the users and developers of the applications.

The applications created on the ProcessGold platform are constantly evolving as visualizations are added or changed, requiring constant maintenance to increase the value they offer to customers.

3.2 Stakeholders

There are two stakeholders: **application users** and **application developers**.

Application users use visualizations to analyze their data. They have access to one or more applications and do not interact with the data transformation graph, but there are some cases where they like to understand it on an abstract level, either to increase confidence in the correctness of the visualizations or as visual support when communicating with application developers.

(Application) developers use the platform to create applications for application users. They maintain the data transformation graph of their applications and are responsible for its correctness.

When we refer to users, we mean both application users and application developers. When we refer to one of the stakeholders, we explicitly exclude the other.

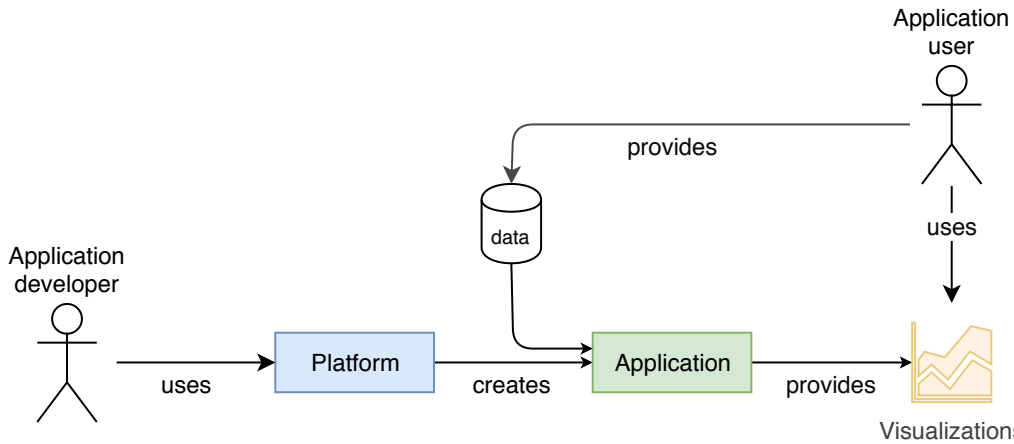


Figure 3.1: An overview of stakeholders and the ProcessGold development process. Developers use the platform (●) to create applications (●) which use data from application users to create visualizations (●). These can then be used by application users to enable them to gain insights from their data.

3.3 Task analysis

To get a better understanding of what maintenance entails for our users, we have analyzed several use cases important for long term maintainability. First, we analyzed how users currently converge their understanding of the transformation graph (RQ1). We found that most communication between developers and application users is done through (sometimes inaccurate) hand drawings created by experienced developers. Then, we analyzed how developers add new features and handle changing requirements (RQ3). We found that once a new requirement is accepted, one or multiple developers determine what to change, apply the change, and verify its correctness. However, determining what to change is usually done without consulting the current transformation graph, resulting in a loss of time when the change is incorrect, as additional discussions have to take place. Finally, we analyzed the current process to resolve bugs (RQ2) and found that developers validate the bug, attempt to find its cause, come up with a method to resolve the bug, apply the method accordingly, and verify the bug is resolved. However, currently, transformations are changed to isolate the specific bug, a complex process, potentially involving long wait times and inaccurate debugging.

Based on the observations we made and previous research [HDBL17], we determine three categories where the tasks of our users fall into. Understanding transformation graphs (**presentation**), finding the cause of bugs (**debugging**) and **modifying** transformation graphs. As we discussed in Section 2.4, data quality (i.e., modifying) transformation graphs is the primary goal of most commercial tools. However, **presentation** and fine-grained **debugging** is often lacking. Therefore, we focus solely on these two categories.

In Figure 3.2, we show how our three sub-research questions relate to one another, and how these work together to improve the maintainability of a data transformation graph. First, we have to ensure application users and developers have a unified understanding of their transformation graph (RQ1), then, users discuss with each on their shared mental model to further improve this understanding, make design decisions and apply changes to the transformation graph (RQ3). Alternatively, they use their mental model to find errors and their cause. Then, they compare them to regular behavior to detect similar errors to resolve (RQ2).

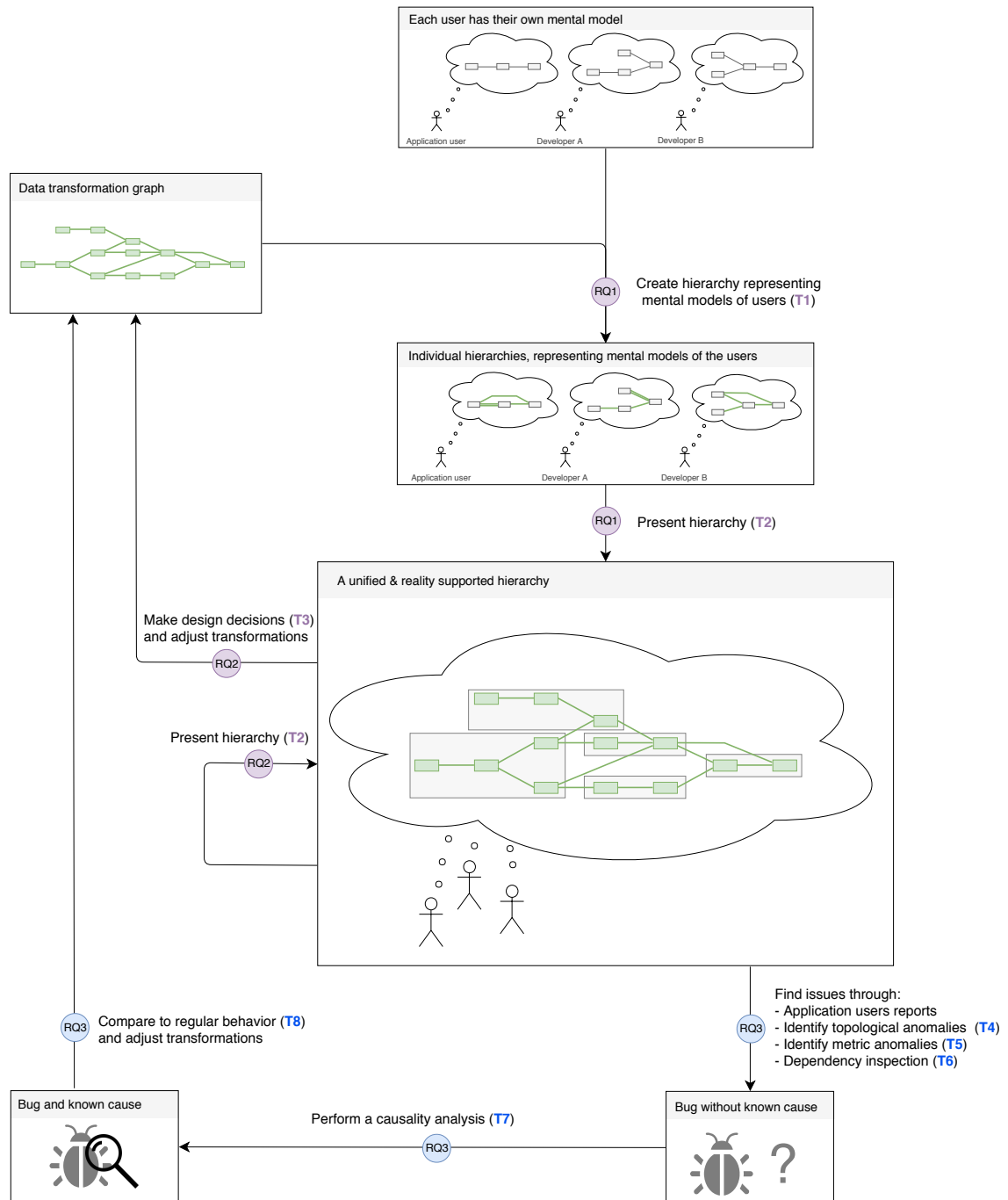


Figure 3.2: An overview of tasks and how they relate to one another. RQ1, RQ3, and RQ2 refer to our sub-research questions discussed in Chapter 1. Purple (●) relate to presentation tasks, while blue (●) relate to debugging tasks.

3.3.1 Presentation

Given a transformation graph, a developer wants to understand the graph on his own or, alternatively, a developer wants to explain the graph to another user. By enabling users to quickly grasp the transformation graph, we anticipate a reduction in the time it takes for a new user to start making changes to the transformation graph. We identify two tasks related to presentation:

T1 Creating presentable data transformation graphs

Users have a so-called *mental model* [WFS93, KMCA06] of their data transformation graph, this model represents an abstract representation of the graph. However, this model is often incomplete or lacks detail. Therefore, we should enable users to re-create their mental model, and automatically integrate this model using the real data transformation graph. This enables them to verify their mental models, which helps to achieve a common understanding of the graph (RQ1).

PR1 A developer should be enabled to create a representation of their mental model.

Developers want to abstract away from a part of the data transformation graph they mentally perceive as a coherent part and be enabled to work with the mental model they have.

T2 Explain structure

Once users have verified their own mental model, users wish to explain the transformation graph in a step by step fashion to other users, starting at an abstract overview and progressively going into more detail. By enabling users to explain the graph to each other, it becomes possible for users to unify their mental models into a single mental model, integrated with the data transformation graph (RQ1).

PR2 A new developer should be enabled to effectively understand the data transformation graph.

When a new developer wants to start working on the graph, he or she requires an understanding of the graph topology to effectively do so. Therefore, an experienced developer wishes to iteratively explain their data transformation graph to the new developer.

PR3 An application user should be enabled to gain insight into the data transformation graph.

Application users want to make sure the data processing is correct. To do so, they want a general understanding of the data transformations applied. Alternatively, they wish to discuss changes to the graph. Therefore, developers should be able to explain the structure to application users.

PR4 Developers should be enabled to create a unified mental model of the data transformation graph.

Developers want to create a single, unified mental model between all developers, such that other tasks can be performed more effectively.

T3 Make design decisions

When users have a unified mental model, they can start making design decisions and improving upon their mental model. As decisions are made, and new changes are applied (RQ3), users mental models can start to diverge when they no longer match their own. This will spark decisions on data modelling, that can be improved due to the presentation capabilities of the visualization.

PR5 Developers should be assisted in making decisions about where and how to extend their data transformation graph

A developer may want to add new features or change existing ones, requiring them to change or extend the data transformation graph. They want to gain assistance in determining where this change or extension would make the most sense.

3.3.2 Debugging

To enable developers to more quickly resolve bugs, they want to perform debugging tasks on their data transformation graphs (RQ2). The primary focus of these debugging tasks is on tracking rows through the data transformation graph.

T4 Identify topology anomalies

With experience, developers develop an understanding of topological anomalies indicative of bad design and wish to detect these. However, these anomalies may vary from user to user, hence, a predefined set of patterns cannot be applied. When an anomaly in the graph topology is found (4 in Figure 3.2), the visualization should enable developers to track down the reason for this anomaly (5 in Figure 3.2), and, once found, enable them to compare it to regular behavior, before they apply changes.

DR1 Developers should be enabled to detect bad design decisions and verify their best practices based on the graph topology

A team of experienced developers has decided that certain patterns should always be used in the data transformation graph. They want to verify these are correctly used. For example, when two different data sources are used, they may deem it a best practice to translate one or both of these data sources into a common format before being used by other transformations.

T5 Identify row changes

Developers want to detect changes in the number of rows, as these may be indicative of incorrect transformations. However, varying business logic is often required to classify them as an error. Therefore, they cannot be automatically detected. For example, it may be the case that a transformation removes some rows because they have invalid timestamps. However, instead of removal, they should have been changed instead. This error can only be detected when we know the change the transformation should have applied.

DR2 Developers should be enabled to detect how many rows are removed in a transformation.

Transformations commonly remove rows. However, in some cases, an incorrect number of rows may be removed. Therefore, developers want to detect this.

DR3 Developers should be enabled to gain an indication of the amount of data duplication.

Transformations may duplicate data. Users wish to detect where this happens, either to verify the correctness of this duplication, or to detect the cause of the duplication.

T6 Inspect dependencies of a transformation

Since developers can design complex transformations, the actual implementation of these transformations sometimes contains errors. Developers wish to quickly see if and how many rows are merged or split to see if the behavior conforms with their expectations. As such, they wish to find issues or assist them in finding the cause of issues.

DR4 Developers should be enabled to detect how a transformation combine their data to form new rows.

In some cases, a developer wants to be certain a transformation correctly combines rows. For example, given two tables, they may expect every row of the first table to be merged with at least one other row of the second table. They wish to verify such behaviour.

DR5 Developers should be enabled to detect disjoint tables.

In some cases, a developer may expect the input tables to not have any relation with each other whatsoever. They want to verify this assertion.

DR6 Developers should be enabled to detect where data of a table is used.

In some cases, a developer expects all data from a table to be used. If not, this could result in errors. Therefore, developers wish to gain insight into where and what data is used given a certain table (i.e., they wish to detect row splits).

T7 Causality analysis

Once a developer has found an anomaly, represented by a set of rows causing unexplained behavior, they wish to identify the cause of this anomaly. Given this set of rows, developers want to either follow these to their origin or decrease the set until a cause for the anomaly is found. They can decrease this set either based on values of one of its columns, or based on their origin or destination.

DR7 Developers should be enabled to find out where rows originate from.

Sometimes, there are too many rows available, resulting in invalid data being produced for data sinks. As a result, application users may report a bug. Alternatively, developers can detect this themselves. Once verified, developers wish to find where these additional rows originate from.

DR8 Developers should be enabled to find out where rows are removed.

Sometimes rows may be missing in a transformation, therefore, developers should be enabled to detect where rows are removed.

DR9 Developers should be enabled to analyze column values.

When a developers wants to understand why certain behavior is occurring, they often require a detailed understanding of the transformation. This understanding can be improved by letting them analyze column values.

DR10 Developers must be enabled to select rows from a table.

To reduce their anomalous set of rows, developers should be enabled to perform selections. Such selections should be easy to perform.

T8 Comparing to regular behavior

Once the specific case is found, developers wish to compare this behavior to the rest of the data, to establish if the behavior is indeed diverging.

DR11 Developers must be enabled to compare a subset of data to the entire data set.

Developers want to compare a subset of the data to the rest, as this allows them to find diverging behavior.

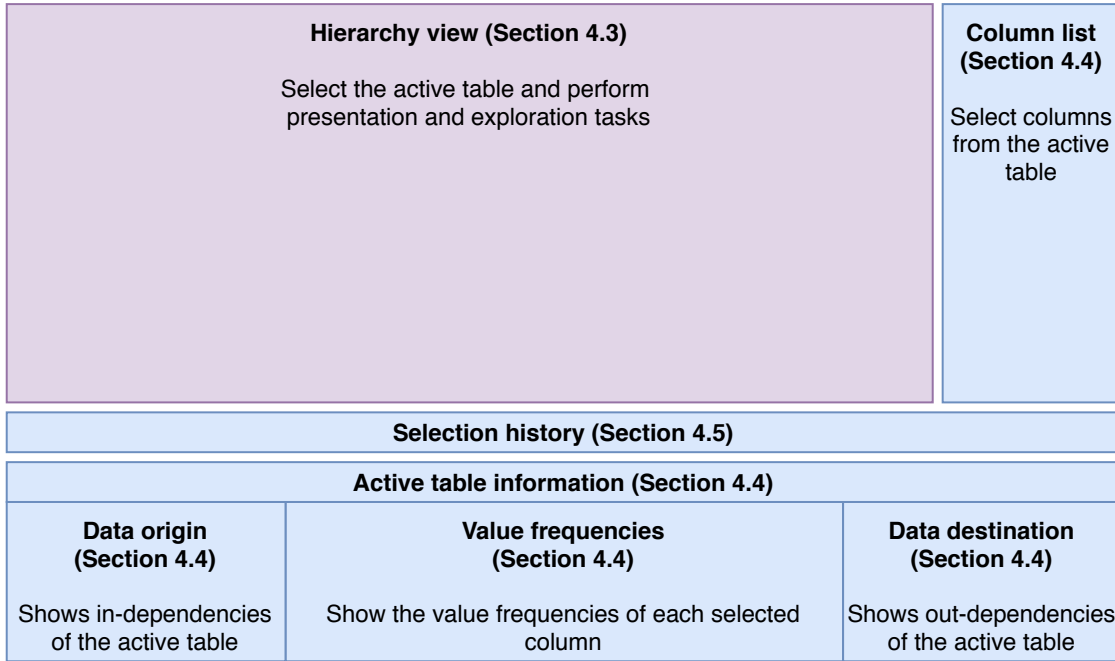
Chapter 4

Approach

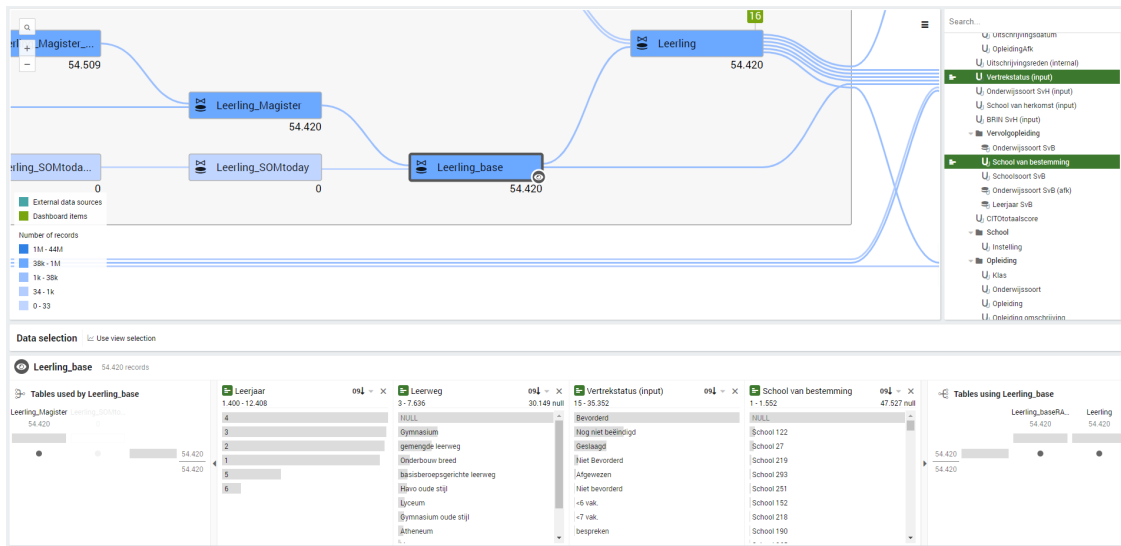
In this chapter, we describe our approach to satisfy the requirements as discussed in Chapter 3. We propose a framework consisting of several components, shown in a schematic overview in Figure 4.1a. We discuss each component in the following sections. First, we discuss mental models and how these play a role in presenting data transformation graphs (PR1, PR2, PR3, PR4) in Section 4.1. In short, we allow developers to construct a hierarchy which abstracts away from the detailed information of the data transformation graph while preserving dependencies between tables. The resulting hierarchy is visualized using the layered graph layout we discuss in Section 4.2.

The graph layout is used to construct the hierarchy view, which we discuss in Section 4.3 (an example is shown in Figure 4.1b). The hierarchy view provides a means of supporting design decisions and enables users to find anomalies in both graph topology and row information (PR5, DR1, DR2, DR3), and serves as the connecting element between the other components by linking these views.

The other views enable users to perform highly technical debugging tasks (DR4, DR5, DR6, DR7, DR8, DR9, DR10, DR11). We discuss these debugging views in Section 4.4. Finally, we discuss interaction and selections in Section 4.5. This enables developers to perform causality analysis and allows them to compare their selections to regular behavior, though the use of the selection history and linking with the hierarchy view (DR7, DR8, DR9, DR10, DR11).



(a) A schematic overview of our approach, with component names in bold and their descriptions underneath.



(b) Our visualization being used on a data transformation graph.

Figure 4.1: An overview of our visualization. The hierarchy view enables users to perform presentation tasks and, in combination with the other views, for debugging tasks.

4.1 Mental models

We found that when users are asked to explain their data transformation graph, basic transformations play a minor role. Instead, they have an abstraction in their mind, called a **mental model** [WFS93, KMCA06]. This abstraction maps all elements of the underlying data transformation graph onto a higher order representation of the underlying operations. We have to enable users to use this mental model while performing other tasks in the visualization (PR1). However, this mental model differs for each developer and for each data transformation graph, hence, it would be impossible to automatically derive this information. Therefore, instead of automatically deriving their model, we let users insert their own mental model.

By enabling users to recreate their own abstraction, it can be tailored for specific purposes, for example, one abstraction may help to explain the graph topology to another employee, while another can help application users (PR2, PR3). Additionally, by allowing only a single abstraction to be used between multiple developers, they are forced to create a unified abstraction, which, through presentation, results in a unified mental model (PR4). Therefore, we enable users to drag and drop transformation nodes into groups, enabling them to represent their mental models in a graph hierarchy.

4.1.1 Graph hierarchy

A graph hierarchy, or hierarchy, has previously been used to store complex types of relations spanning multiple levels of abstraction [AHK06, AMA08]. Such a hierarchy is formed by defining a hierarchy tree consisting of groups on top of an existing graph. A hierarchy tree and groups are defined as follows:

Definition 4.1.1 (Hierarchy tree). A hierarchy tree $T = (V_T, E_T)$ for a graph $G = (V, E)$ is a tree rooted at $root(T)$ where $V \subseteq V_T$. Given a node $n \in V_T$, $nodes(n)$ defines the set of nodes for which there is a path from n to a node $n' \in V$ given the set of edges E_T . The children of n are defined as $children(n)$, while the parent of n is defined as $parent(n)$.

Definition 4.1.2 (Group). We refer to a node n in a hierarchy tree T as a group if it is a non-leaf node.

An example of a hierarchy is shown in Figure 4.2. In the figure, the leaf nodes (shown in blue) form a directed acyclic graph G , while the colored rectangles indicate the groups of the hierarchy.

4.1.2 Hierarchy navigation

We can reduce complexity by using a hierarchy, however, when explaining the transformation graph to another developer (PR2), detailed information is often required. Therefore, users should be enabled to navigate through the groups of the hierarchy. There are two main methods of navigation in groups: drill down and roll up. When the user wants more detailed information they can perform a drill down to show the children of the groups they drill down on. Alternatively, when they want to hide the details of a given set of groups, they can perform a roll up, which hides the children of the groups the roll-up is performed on.

We define a boolean $open(u)$ for every group u , which indicates whether it is showing its children. Hence, when we perform a drill down on u , $open(u)$ becomes true and when we perform a roll up $open(u)$ becomes false.

Elmqvist and Fekete [EF10] classify five different methods of performing navigation through a hierarchy (shown visually in Figure 4.3):

- **Above traversal** shows all nodes above the currently selected depth;
- **Below traversal** shows all nodes below the currently selected depth;

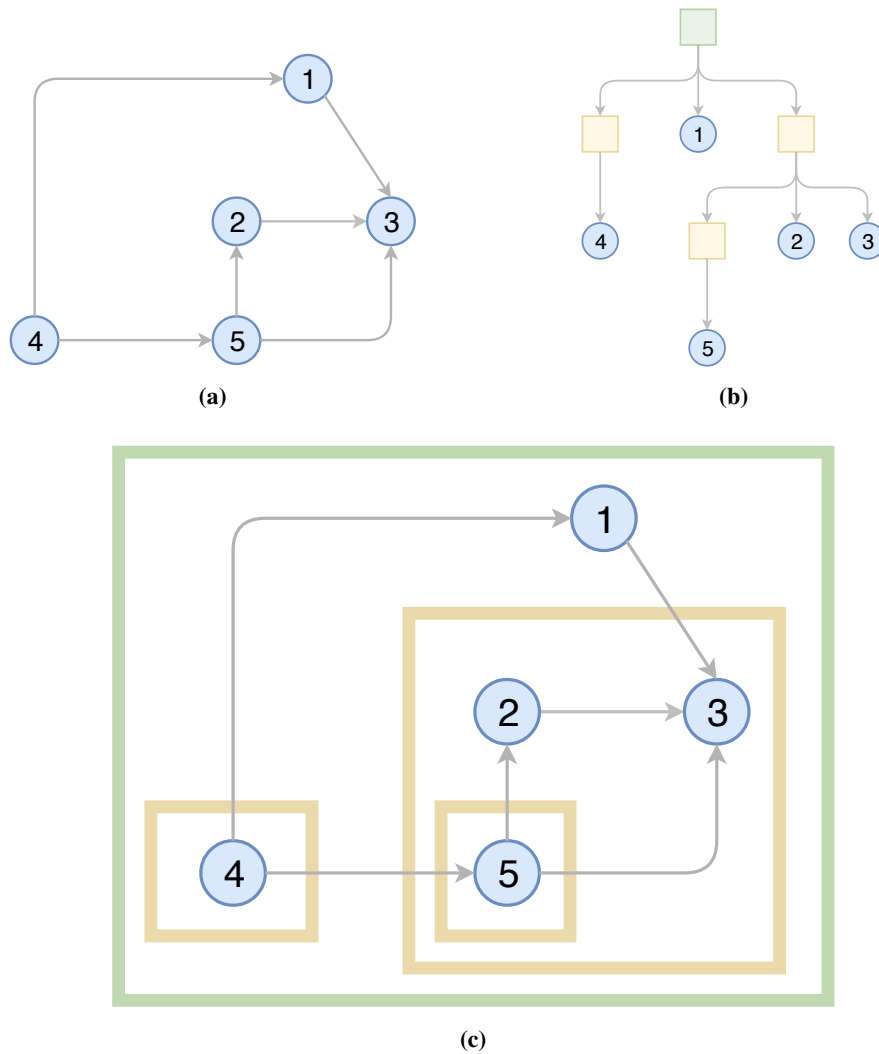


Figure 4.2: An example of (c) a hierarchy created by combining (a) the graph $G = (V, E)$ and (b) hierarchy tree T . The nodes V are shown in blue (○). The rectangles indicate the groups of the hierarchy tree T . The root is shown in green (◻), while the other groups are shown in yellow (◻).

- **Level traversal** shows all nodes on the current level of depth.
- **Range traversal** shows all nodes in between two levels of depth;
- **Unbalanced traversal** shows only certain branches of the tree.

To facilitate incremental presentation (PR2) our visualization makes use of unbalanced traversal since we wish to enable users to select very specifically which groups of the tree they wish to present/explore. This is done by enabling them to perform a drill-down or roll-up on every individual group node, with the root always being opened. We refer to drill downs and roll ups as *opening* and *closing* of a group.

4.1.3 Attribute aggregation

We use a hierarchical aggregation method which allows us to aggregate information from the leaves to any group in the hierarchy [EF10]. We aggregate this information using a bottom-up approach: starting at the leaves, we compute the attribute values for every group based on their children until we reach the root.

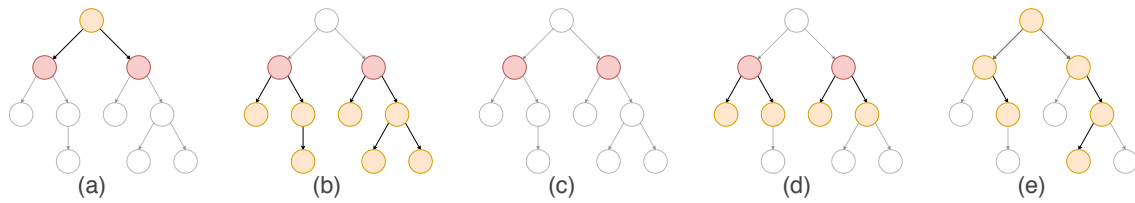


Figure 4.3: The different hierarchical traversal types, where red (●) indicates the current depth while both orange (●) and red are visible nodes: (a) above traversal, (b) below traversal, (c) level traversal, (d) range traversal, and (e) unbalanced traversal.

Doing so allows us to visualize all attribute information on every level of the hierarchy using attributes based on our leaves. This allows us to visualize important information such as the number of data sources which are present in a group. In Section 4.3 we describe which attributes we use and how we visualize them.

In our visualization, we make use of two aggregation methods:

- **Sum** We simply sum all the values of our children. An example of this aggregation method is the total number of data sinks and sources;
- **Extrema**, or the extremes of an aggregated attribute. For example, the maximum and the minimum number of rows.

4.2 Graph layout

Since our visualization is based on a node-link diagram, an effective graph layout is crucial to understanding the data transformation graph. In this section we describe the reasoning behind creating our own graph layout and the approach we took.

4.2.1 The dot layout

Originally, we looked into using the most widely used implementations of the Sugiyama framework [STT81] for our problem, the dot layout [Gra16], but noticed several shortcomings of the algorithm for our application.

- **Nodes do not align vertically:** When nodes are aligned, users perceive these as belonging together. However, the algorithm places nodes such that edge length is minimized, not taking alignment of nodes into account;
- **Limited bundling:** Edge bundling is an effective method of reducing clutter and showing an overall trend [PNK11]. Understanding this overall trend enables users to understand the transformation graph better (PR2, PR3), however, defining constraints on edge start and end positions is limited in dot, therefore, the implementation of some forms of edge bundling is hard or impossible;
- **Stability:** As we navigate through the hierarchy, nodes are re-sized. However, the dot layout does not guarantee stability when we change node size.

Due to these shortcomings, we have decided to create our own layered graph layout algorithm, which we describe in the following sections.

4.2.2 Mental maps

Node sizes can change when users open or close groups of the hierarchy, as children of an open group are drawn inside of them. However, changes in the size of nodes may cause regular graph layouts to break the so called 'mental map' of users. The mental map was first defined by Misue *et al.* [MELS95] and describes how users believe the graph topology is built up, outside of the visible part of the screen.

Misue *et al.* define three models that describe the mental map. The orthogonal ordering model tries to ensure left, right, up, and down have meaning, while the proximity and region model, state the proximity and region of a node should not change, respectively. In our visualization, we deemed the ordering and region to be the most important to preserve, while the proximity model is of less importance, and according to Misue *et al.* also quite intractable.

In our layout algorithm we maintain the mental map of the users as best as possible by using phased animation and graph stability, as the loss of the mental map makes understanding the transformation graph harder (PR2, PR3).

Phased animation in graphs has previously shown to be an effective method of preserving the mental map [BPF14]. It consists of the following three phases: nodes are removed, moved and re-sized; and finally, added. When a phase is not required (e.g., there are no nodes which were removed), it is skipped, immediately continuing to the next phase. We tested several duration for each phase, ranging 100ms to 1s and asked users whether they could pinpoint the change. We found that a duration of 500 ms per phase gives users sufficient time to detect changes between two states of the graph. Phased animation is used when adjusting the hierarchy or when opening and closing its group.

Several of our requirements benefit from preservation of relative directions between nodes, as this makes tracing paths easier for users (PR2, PR3, DR1, DR7, DR8). Therefore, we ensure that when opening and

closing groups, all relative directions between nodes are preserved, i.e., when a node is at the top left before opening a group, it remains at the top left after opening.

4.2.3 Layered graph layout

Our layout algorithm uses the same framework as set by Sugiyama (see Section 2.3.1). However, we made several changes to enable us to satisfy our visualization requirements. We describe every step and the changes made in the following section.

Cycle removal

As discussed in Section 2.3.1, we have to remove cycles to compute a layered graph layout. Therefore, we identify the smallest possible set of edges of which, when reversed, cause the graph to become acyclic. This problem is also known as the (NP-complete) feedback arc set problem.

There are several approaches to solve this problem, ranging from fast greedy heuristics to slower algorithms computing optimal solutions. We chose to use a fast greedy approach [ELS93] which runs in linear complexity based on the number of edges, since this algorithm provides the same asymptotic bound on performance compared to more complex algorithms [BS90] when computed for sparse graphs, such as data transformation graphs.

The algorithm uses a greedy approach to compute a vertex sequence s . A vertex sequence is a horizontal ordering of each vertex in the graph. Imagine this ordering as a horizontal line on which every node is positioned. We wish to find the ordering of these nodes, such that the edges going from right to left is minimized. Once s is computed, we determine the feedback arc set of the vertex sequence by taking all edges pointing to left.

Computation of the vertex set s is done by processing nodes one by one. First, we remove sinks and concatenate them with a vertex sequence s_2 until none are left. Then, we do the same with sources, but concatenate the vertex sequence s_1 with each node. Finally, we pick a u to remove where $deg_{out}(u) - deg_{in}(u)$ is maximized and concatenate u to s_1 , until all nodes are removed. We then compute the vertex sequence $s = s_1 s_2$. For a more formal definition, see Algorithm 1.

Once the vertex set is computed, we reverse all edges pointing left and perform the next steps.

Algorithm 1 Eades greedy FAS algorithm

```

1: procedure GREEDYFAS( $G : graph$ )
2:    $s_1 \leftarrow \emptyset, s_2 \leftarrow \emptyset$ 
3:   while  $G \neq \emptyset$  do
4:     while  $G$  contains a sink  $u$  do
5:        $s_2 \leftarrow us_2$ 
6:        $G \leftarrow G - u$ 
7:     while  $G$  contains a source  $u$  do
8:        $s_1 \leftarrow s_1u$ 
9:        $G \leftarrow G - u$ 
10:    choose a vertex  $u \in G$  for which  $deg_{out}(u) - deg_{in}(u)$  is maximized
11:     $s_1 \leftarrow s_1u$ 
12:     $G \leftarrow G - u$ 
13:  return  $s_1 s_2$ 

```

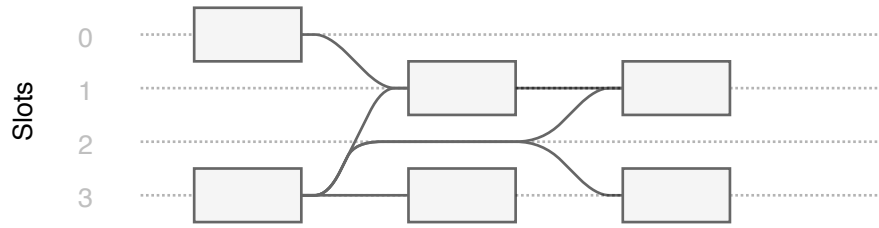


Figure 4.4: A horizontal graph layout with the slots at which nodes are positioned are indicated by horizontal lines.

Node ranking

The Sugiyama approach uses a network simplex approach to assign node ranking. The goal of the ranking computation is to create a balanced number of nodes on every rank. In practice we found that the number of nodes in data transformation graphs tends to decrease the more edges one follows from a data source (nodes with an in-degree of zero). Therefore, instead of using a network simplex approach, we compute node ranking based on the maximum distance from a data source. Therefore given a graph $G = (V, E)$, we compute the rank for every node $n \in V$ as follows:

$$rank(n) = \begin{cases} 0 & deg_{in}(n) = 0 \\ \max_{(a,n) \in E} (rank(a) + 1) & otherwise \end{cases} \quad (4.1)$$

Using this approach, nodes tend to move towards the right side of the layout, which, assuming the graph is indeed decreasing in the number of nodes, creates a balanced layout. In case the number of nodes does not decrease further away from data sources, this can be prevented by using a network simplex approach such as Sugiyama does, however, this is more complex and expensive to compute.

Node order

Our layout uses a technique similar to the median-based crossing minimization proposed by Eades *et al.* [EW94]. We perform a down-up sweep across all ranks: Starting at the second rank, we order every node based on the median position of nodes on the previous rank. This is then repeated, but in reverse. This sweep is repeated several times, as this allows the layout to improve after every so called iteration. The iterations stop when the order is stable (i.e., no changes in order occur) or when the number of iterations becomes too large.

The difference between the technique proposed by Eades and ours is what we use to compute the median. Eades uses the median y-position of the neighbours of each node, while we use the median slot. Slots are an additional constraint on the positions of our nodes. These constraints may lead to longer edges which are harder to trace, however, they create a more structured visual representation of the transformation graph, and preserves the mental map of users by preserving the relative directions between nodes when a group is opened or closed. We apply these constraints through the concept of slots (See Definition 4.2.1 and the visual example in Figure 4.4).

Definition 4.2.1 (Slot). A slot indicates a possible position for a (virtual) node in a graph layout. There are $\max_{\{0 < i < |ranks|\}} |R_i|$ slots on every rank. The slot of node n is denoted as $slot(n)$.

More formally, given a node u having k neighbours, n_1, \dots, n_k , where the nodes are sorted based on their order value (i.e. $\sigma(n_1) < \sigma(n_2) < \dots < \sigma(n_k)$). The median of node u , $med(u) = slot(\lfloor k/2 \rfloor)$.

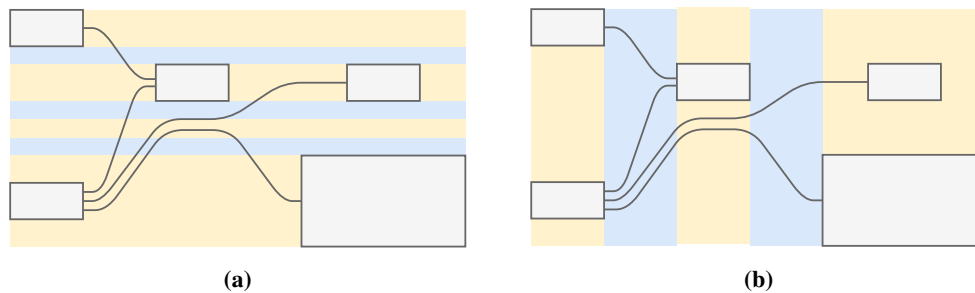


Figure 4.5: Computation of rank (a) and slot (b) sizes. Yellow (◻) indicates the ranks or slots, while blue (◻) indicates the margin in between. Note how the size of a slot depends on node or edge bundle size.

Node positioning

After computing ranks and slots, node positions are assigned. To do so, we first compute the size of every rank and slot (See Figure 4.5 for an example). Using these values, we create a grid and place our nodes and edges at the center of their respective cell in this grid. Doing so results in a structured layout, that maintains the mental map of the user by preserving relative positions. However, it causes longer edges in between nodes when groups are opened.

Spline drawing

Before splines are drawn, we restore all edges which were previously reversed when removing cycles. For each spline, we make sure every edge (u, v) starts at the right border of u and ends at the left border of v . Due to this property, users can infer the edge direction without the use of directional arrows.

This is done by drawing a simple Bezier curve when the edge is increasing in rank (i.e. an edge which does not cause a cycle), or a more complex one, creating a loop going from the right side of the source node to the left side of a target node. A comparison of forward and back splines is shown in Figure 4.6.

4.2.4 Hierarchical layout

When computing the layout for a hierarchy, we have to ensure it correctly represents the state of the hierarchy to our users (PR1, PR2, PR3, PR4), therefore, a hierarchical layout must satisfy the following requirements:

- HL1 **All nodes must be contained in their parent:** If they are not, users will be unable to detect which nodes are children of a group.
- HL2 **Only nodes of which all ancestors are either an open group or the root should be visible:** As previously stated in Section 4.1.2, we apply unbalanced traversal to facilitate incremental presentation (PR2), therefore, only these nodes should be visible.
- HL3 **Edges may only cross a group boundary if either their source or target is part of this group:** When edges cross a group boundary, users will associate this edge as having a relation with this group. Therefore, if we allow crossing a group boundary without the edge having a relation with any nodes in the group, users will likely be confused.
- HL4 **Performing a roll up or drill down on a group should not change the relative positions of the node, as this would negatively affect mental map of the user:** As previously indicated, preserving the mental map is important to satisfy our visualization requirements.

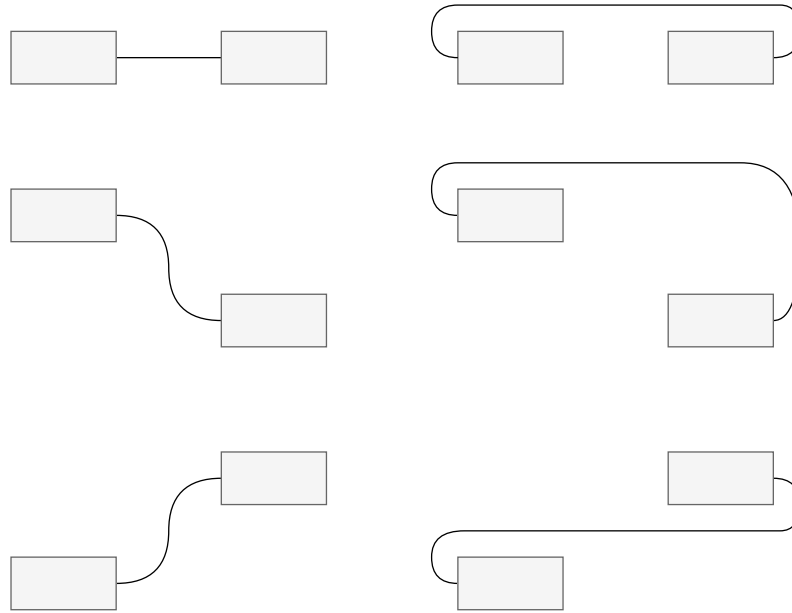


Figure 4.6: How splines are drawn given nodes on the same rank, one on a lower rank and one on a higher rank. On the left, we show the forward edges between those nodes, while on the right, the backward edges are shown. Note how all nodes have their outgoing edges on the right and incoming edges on the left.

Algorithm 2 Hierarchy layout

```

1: procedure HIERARCHYLAYOUT( $n$ )
2:   if  $open(n)$  and  $isgroup(n)$  then
3:      $nodes \leftarrow \emptyset$ 
4:     for  $c \in children(n)$  do
5:        $cn \leftarrow HierarchyLayout(c)$ 
6:        $nodes \leftarrow nodes \cup cn$ 
7:      $size(n) \leftarrow ComputeLayout(nodes)$ 
8:   return  $n$ 

```

To satisfy HL1, HL2, and HL3, we treat every open group as a subgraph and compute a layout for each group individually. Every open group is then replaced by a node with the size of the bounding box of its subgraph when computing its parents layout. Pseudo code is given in Algorithm 2 where $isgroup(n)$ is true if n is a group, $size(n)$ defines the size of a node n , and $ComputeLayout$ is a function which uses a set of nodes (and their edges) to compute a layout for those nodes, returning the bounding box of the layout.

To satisfy HL4, we compute the layout of the subgraph of each group individually, meaning we cannot use information about the layout of a subgraph outside of its group. To ensure every edge going in or out of each group is drawn correctly, we add two virtual nodes in the center of the lowest and highest rank. Every edge that goes in or out of the group must go through one of these virtual nodes (See Figure 4.7). Applying this technique enables us to ensure stability when opening and closing groups, although it may cause an increase in edge length and crossings, in practice we found this increase to be minimal.

When we open a group u , we can limit the layout computation to only its children if we move all nodes surrounding the group we open. Similarly, when closing, we only have to move the surrounding nodes and edges, requiring no layout computations. Although we could compute these changes quickly, it requires additional logic to be implemented for moving surrounding nodes and edges. Therefore, we instead recompute the layout of all ancestors of u when opening or closing a group. Since our layout is independent of

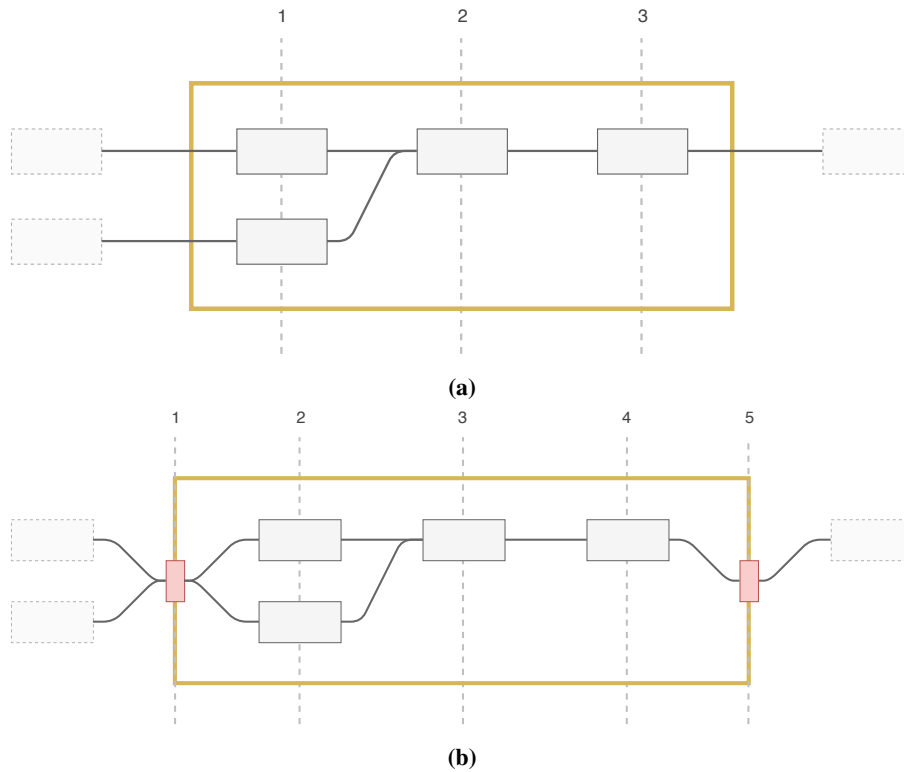


Figure 4.7: Comparison of a group being drawn without (a) and with (b) virtual nodes. The edges that cross the group boundary (●) cause the layout outside of the group to become dependent on the position of nodes inside the group. Adding virtual nodes (●) on the border of the computed layout may result in more edge bends and longer edges, however, the layout outside the group becomes independent of the node positions inside.

node size, recomputing the ancestors layout will effectively perform such a move.

4.2.5 Edge bundling

Graphs can often get cluttered when a large number of edges is drawn. Edge bundling is an effective method of reducing this clutter and showing an overall trend [PNK11]. However, some edge bundling techniques suffer from a loss of traceability of individual edges. Since many of our tasks are path based, the loss of traceability is an issue we want to prevent. Therefore, we add a small margin around the edges, ensuring each edge can be tracked individually. Finally, we apply edge straightening, to reduce the number of edge bends.

Edge bundling is applied after the node order is computed. For every rank we merge all adjacent virtual nodes into a single bundle, i.e., given two nodes u and v we merge them if $abs(\sigma(u) - \sigma(v)) = 1$. Then, we add a margin to the edges by moving the start and endpoints away from each other with a predefined margin. To accomplish this, we use a technique introduced by Pupyrev *et al.* [PNK11]. We first sort the edges, such that crossings in the bundle are minimized, then, we move them apart. The result is shown in Figure 4.9b.

Then, given a bundle B , we then sort every pair of edges $(e_0, e_1) \in B$ as follows: the start nodes u of e_0 and v of e_1 , we walk across the ranks in decreasing order until we reach a fork: i.e. where we reach nodes u' and v' for which $u' \neq v'$ (see Figure 4.8a). We then sort the edges based on y-position of u' and v' . If such nodes u' and v' are not found, we start again from u and v but instead walk across the ranks in increasing order (see Figure 4.8b). If both fail to find such nodes u' and v' , it means both edges are

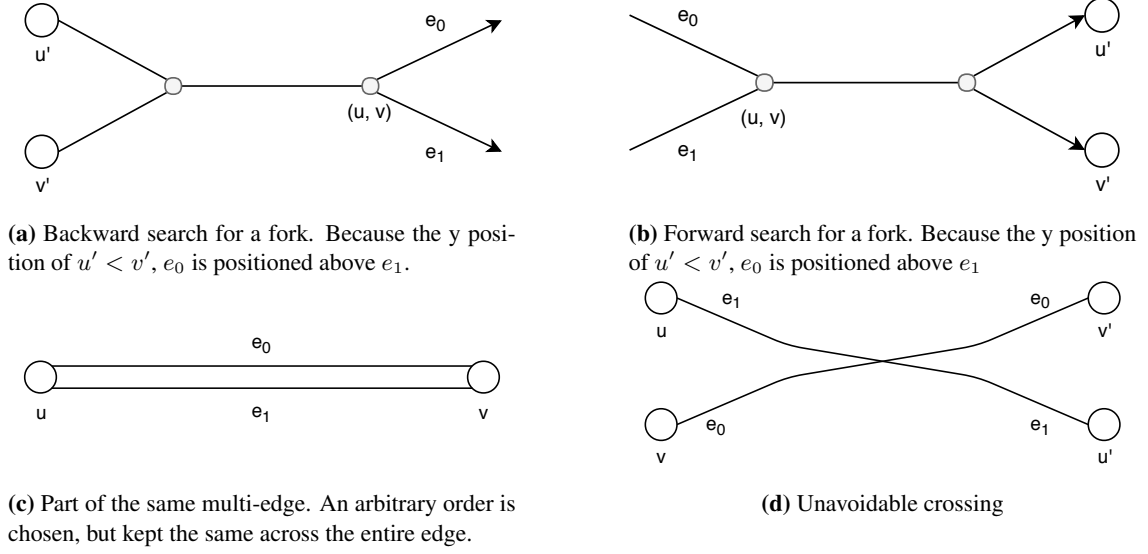


Figure 4.8: The different cases that occur when sorting an edge bundle.

Algorithm 3 Edge straightening

```

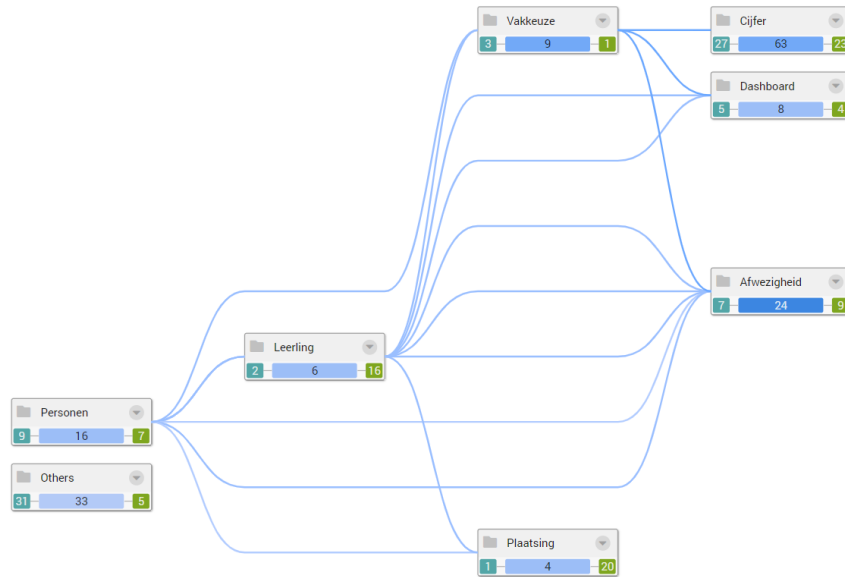
1: procedure STRAIGHTENEDGES( $R : ranks$ )
2:    $iterations \leftarrow 0$ 
3:   while  $iterations < maxIterations$  do
4:     for  $i = |R| - 1$  To 1 do
5:       for all  $n \in R_i$  do
6:          $y \leftarrow$  sorted array of y end positions of edges  $(n, m)$  where  $m \in R_{i+1}$  and  $slot(m) = slot(n)$ 
7:          $B_n.y \leftarrow (y_{\lfloor (|y|+1)/2 \rfloor} + y_{\lceil (|y|+1)/2 \rceil})/2$ 
8:       for  $i = 2$  To  $|R|$  do
9:         for all  $n \in R_i$  do
10:         $y \leftarrow$  sorted array of y start positions of edges  $(m, n)$  where  $m \in R_{i-1}$  and  $slot(m) = slot(n)$ 
11:         $B_n.y \leftarrow (y_{\lfloor (|y|+1)/2 \rfloor} + y_{\lceil (|y|+1)/2 \rceil})/2$ 
12:        $iterations ++$ 
    
```

part of the same multi-edge, i.e., the edges have the same source and target (see Figure 4.8c). In this case, we choose an arbitrary order, and keep this order fixed for the entire edge. If $(y_u - y_v)(y_{u'} - y_{v'}) < 0$ a crossing is unavoidable. Such as case is shown in Figure 4.8d.

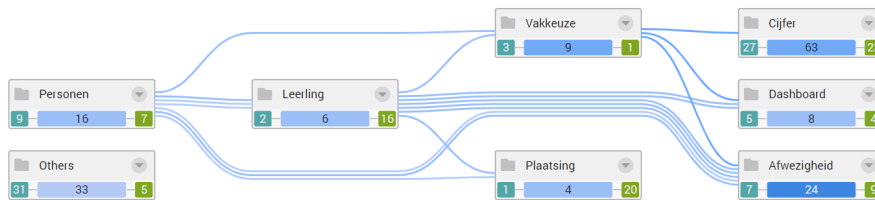
It may become possible that bundles are too large to fit within the height of a node. If this is the case, we normalize the positions of the edges based on the node height, resulting in some loss of traceability.

By using edge bundling, we get a graph with individually traceable edges, however, edges may still have tiny distracting bends (See Figure 4.9b). To reduce this effect, we apply edge straightening, i.e., moving the start and endpoints with a given offset. This offset is based on the median y positions of the neighbouring bundle on the same slot, and is limited to the node height.

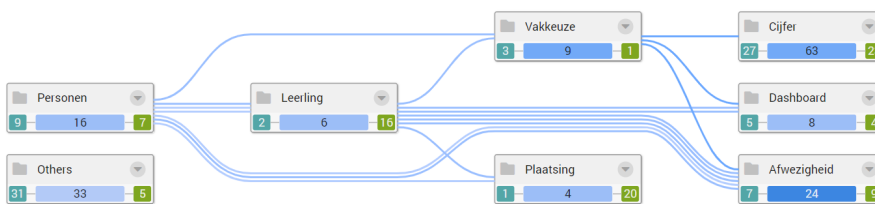
More formally, given a graph with max rank $|R|$, and starting at rank 1, we perform a down-up sweep as shown in algorithm 3 (recall that the set of nodes at rank i is defined as R_i). Similar to edge crossing minimization, this is performed for several iterations, or until the number of iterations grows too large.



(a) Before bundling, the graph is not compact, although edges are traceable.



(b) When applying bundling, edges become individually traceable, but due to the amount of curves, the visualization seems more complex than it actually is.



(c) After applying straightening, the layout becomes clearer.

Figure 4.9: A comparison of the same graph after every step.

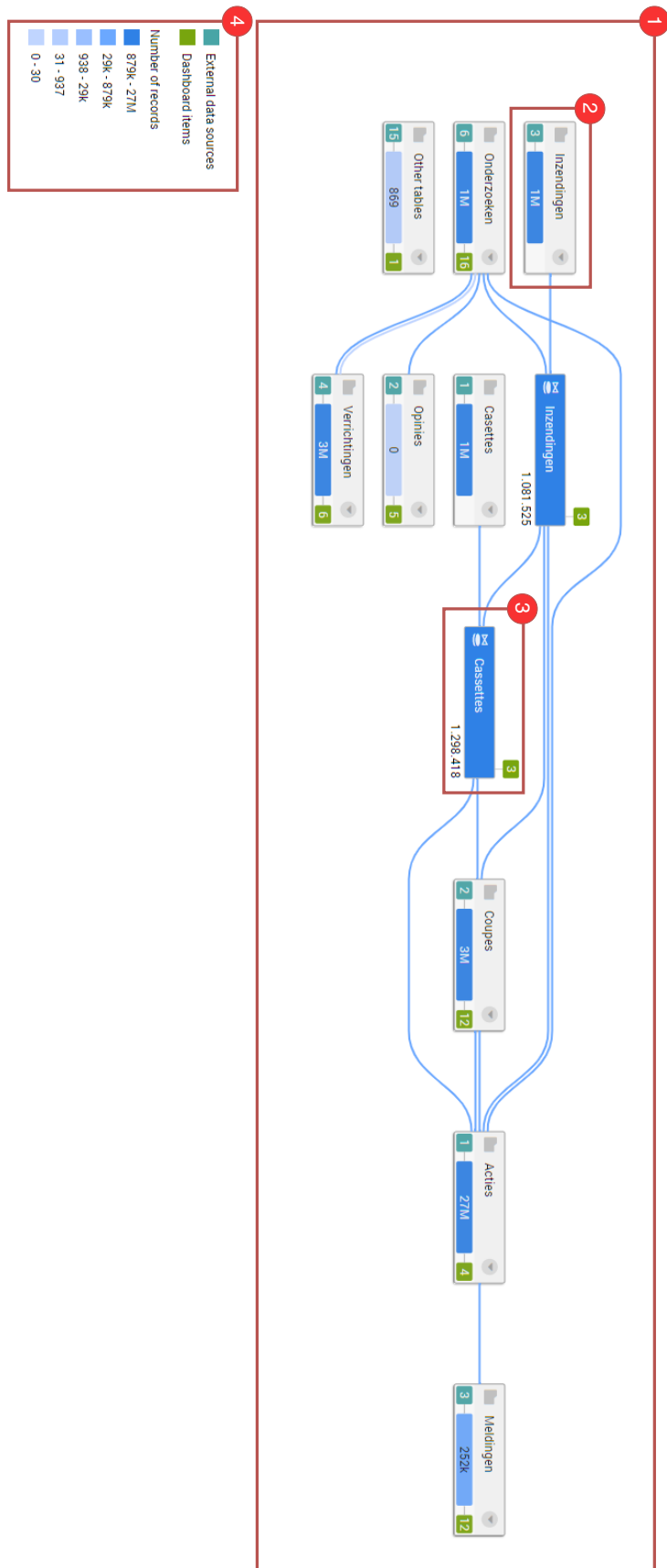


Figure 4.10: The hierarchy view, showing the current state of the hierarchy (1). The hierarchy consists of two components, table nodes (2) and group nodes (3). As stated in the legend (4), the blue color represents the number of rows of each table, while the teal (●) and olive (●) represent the data sources and sinks respectively.

4.3 Hierarchy view

The hierarchy view is shown in Figure 4.10. It shows the current state of the hierarchy (1), using table nodes (2) and group nodes (3). Group nodes can be opened and closed to explore the hierarchy in more detail, while table nodes represent transformations and the table they produce. The view enables users to recreate their mental by dragging and dropping (PR1), presents a clean and clear overview of the data transformation graph (PR2, PR3, PR4), enables developers to quickly detect anomalies and make design choices (PR5, DR1, DR2, DR3), and provides context for causality analysis (DR7, DR8, DR9, DR10).

In the following sections, we discuss how we represent hierarchy and the design of the table and group nodes.

4.3.1 Representing the hierarchy

We enable users to construct their own hierarchy tree to form a hierarchy which fits their mental model (Section 4.1). The current state of this hierarchy is shown in the hierarchy view, using the graph layout discussed in Section 4.2.3.

We use a horizontal graph layout rather than a vertical layout. There are several reasons for this choice. First, our requirements state we should enable developers to easily track row propagation throughout the graph (DR4, DR5, DR6, DR7, DR8), therefore, since rows are stacked vertically in a table, a horizontal layout reinforces the idea that edges represent row propagation. Additionally, edges in the hierarchy do not represent inheritance relationships, therefore, previous research shows a horizontal layout is more effective [WS06]. Finally, a horizontal layout causes forward edges to align with the reading direction of users (in the cultures where we evaluated our visualization), improving understandability.

4.3.2 Table nodes

A nodes in the data transformation graph represents either an external entity or a transformation. Data sources can be quite diverse, for example, they can represent the result of a query performed on an external database, while they can just as well be a simple spreadsheet. All of these are immutable, i.e., developers using our visualization have no way to change this external data. Similarly, data sinks represent external applications or visualizations where data arrives after being transformed by developers. Once here, developers can no longer perform any transformations. Therefore, we extract all data sources and sinks from our data transformation graph, and instead of including them directly in the layout, the information about data sources and sinks is used as an attribute for each neighbouring transformation. Hence, all table nodes in our visualization represent transformations.

The table nodes should satisfy the requirements we described in Section 3.3. Given these, we derive the following requirements for our table nodes:

TN1 Outlier detection: The design of a table node should enable users to detect outliers quickly, as these often indicate data duplication or missing rows (DR2, DR3, DR8).

TN2 Comparable: Any difference in the number of rows between two nodes should be visible. Although outlier detection is effective for large changes, small changes should also be detectable (DR2, DR8).

TN3 Identifiable: To enable users to effectively present the hierarchy and detect structural anomalies, the nodes should be identifiable as tables for users. (PR2, PR3, PR4, PR5, DR1).

TN4 Usage: We should enable users to quickly see how a table node relates external entities and other table nodes as this enables users to detect bad design decisions (DR1) and provides context to why rows are (not) available in a table (DR7, DR8)

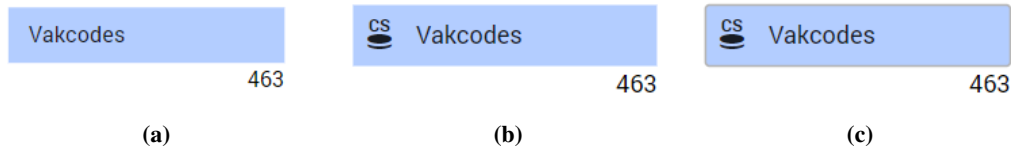


Figure 4.11: Table node design progression. (a) The first version uses color maps and row counts to make it easy to detect outliers and be comparable with other nodes. (b) We add an icon to make them more identifiable. (c) Finally, a gray border is added to increase separation with the white background.

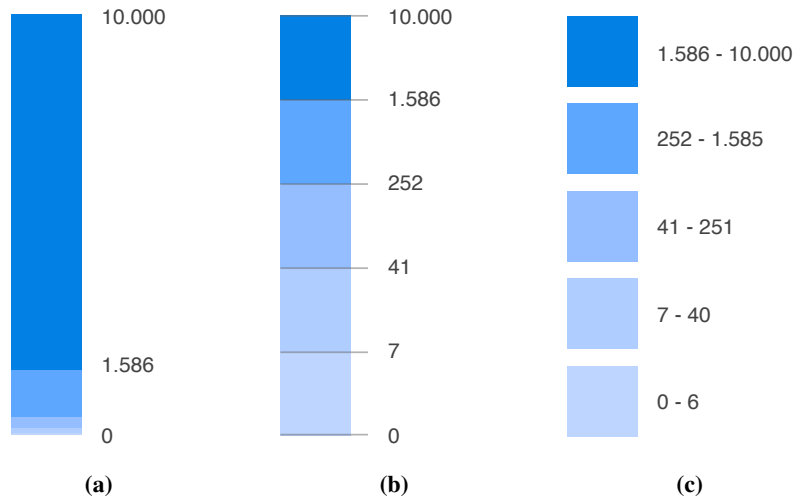


Figure 4.12: Three types of color maps representing the same color map. (a) Proportional stacked bar (b) stacked bar (c) separated legend.

Each transformation can consume data from multiple tables to produce new data. However, a transformation can produce at most a single table as result. Since we want our visualization to be as presentable as possible (PR2, PR3, PR4, PR5, DR1), every node of a transformation is used to visualize information about the table it produces, enabling us to create a clean and simple visualization. The progression of our table node design is shown in Figure 4.11. We describe our design choices below.

We enable users to quickly detect outliers (TN1) by projecting the number of rows onto the table nodes using a color map. Although size is also an effective channel for representing the number of rows [Rot17, Mun14], when using size, identifiability (TN3) of the table nodes suffers, as nodes no longer have the same, and group node size also varies when being opened or closed.

In data transformation graphs, the number of rows has a large range (between zero and a tens of millions of rows). Unlike linear scales, log scales are an effective means of representing large ranges of numbers, as they are based on orders of magnitudes instead. Therefore, we use a color map based on a log scale. By conducting several interviews, we found that users were mostly interested in differences of at least an order of magnitude, therefore, we use a \log_{10} scale.

We evaluated several legends for displaying the color map we use (Figure 4.12). When using a stacked bar the size of some bars becomes too small to detect its color (Figure 4.12a). The stacked bar has ambiguous binning (e.g. it is unclear what color the value 7 will receive in Figure 4.12b). Therefore, we use a separated legend (Figure 4.12c), as it is unambiguous on how the color map works.

Using color scales satisfies detection of outliers (TN1), however, with this method, two nodes are not comparable when the difference in row count is small (TN2). Since these small deviations are often anomalies in data flow (e.g., a single row is removed in a transformation), users wish to be able to detect these. Therefore, we provide text-based row counts to allow users to detect small deviations (See Figure 4.11a). When

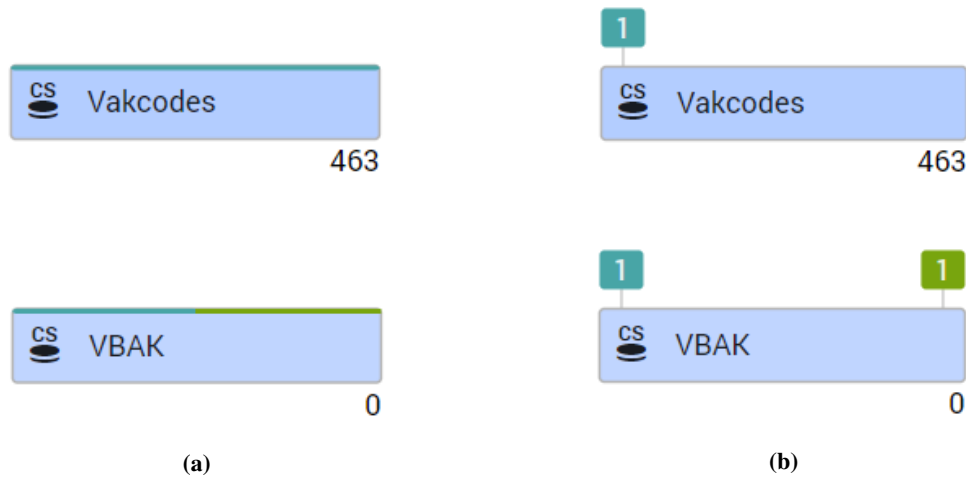


Figure 4.13: Designs considered for sinks and sources being shown next to nodes. The teal (●) indicates information about data sources, while the olive (●) indicates information shown about data sinks. (a) the number of data sources and sinks are shown as a line inside the table node. (b) Tiny nodes are used and a number is added.

text is placed on the bottom right of the table node, we found users associated the number with the outgoing data, as the outgoing edges are also on the right.

Our first design (design 1, Figure 4.11) allows users to detect outliers and compare nodes (TN1, TN2) but fails to meet TN3. Therefore, we add an icon depicting a table to the design (design 2, Figure 2). Our users associate this icon with a table, however, a more broadly recognized icon for representing a table should also be effective. Although improved, users remarked on issues with node contrast, especially when a lighter blue color was used for the nodes with our white background. Therefore, borders were added in the third design shown in Figure 4.11. We chose a subtle gray color, which is not distracting but does provide separation between the node and the background.

In addition to the edges indicating what tables are connected to this table, each table node also has an indication of the data sources and sinks it connects to (TN4). Since we do not explicitly include data sinks and sources, transformations can be connected to both data sources and sinks at the same time. We considered the designs shown in Figure 4.13. Eventually, we settled on the design in Figure 4.13b. There are several reasons for this: first, the design in Figure 4.13a is subtle and hard for users to see. Secondly, it should be an indication that the node is connected to a data sink or source, however, it can be misinterpreted as the table node itself being such a sink or source. In the second design, these two issues are resolved. Although it causes an increase in visual load for the user, it removes any misinterpretations, and allows the number of data sources and sinks to be displayed. Adding these, we receive the final design of our table nodes, satisfying all of our requirements (shown in Figure 4.14).

4.3.3 Group nodes

When a group is closed, the table nodes it contains are hidden and cannot satisfy our visualization requirements. Therefore, the closed group node should satisfy the same requirements. Based on this observation and the requirements specified in Section 3.3, we derive the following requirements for group nodes:

GN1 Identifiable: A group node should be quickly identifiable as one, such that creating and presenting mental models becomes easy (PR1, PR2, PR3, PR4, PR5, DR1)

GN2 Summary: A closed group should provide a summary of the underlying table nodes such that presentation and finding bad design choices becomes easier (PR1, PR2, PR3, PR4, PR5, DR1).

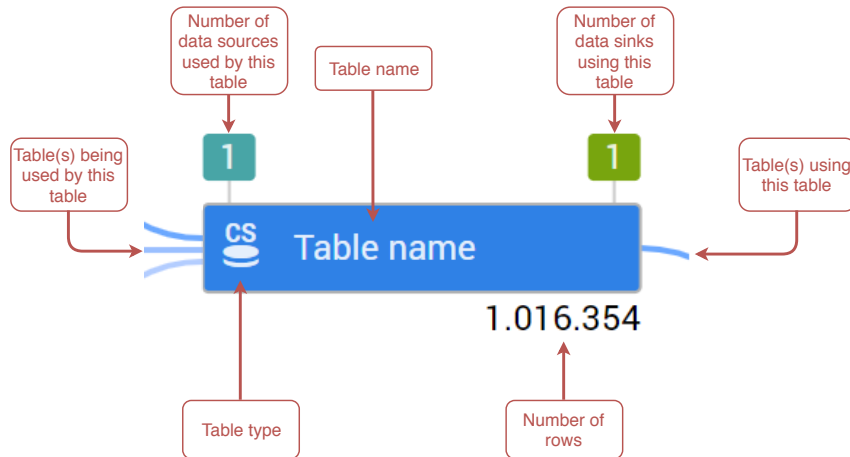


Figure 4.14: The final design of a table node, showing the number of data sources and sinks, a label indicating the table name, an icon to indicate the type of a table, and the number of rows.

GN3 Usage: A closed group node should provide an indication of what data is used by tables inside of the group, and what data is used from the group by other tables, such that presentation becomes easier (PR2, PR3, PR4, PR5), bad design decisions can be detected (DR1), and context is provided to where rows come from and are removed (DR7, DR8).

GN4 Exploration value: A closed group node should provide an indication of exploration value when trying to detect anomalies, such that the user can determine which group to explore (DR2, DR3, DR8).

GN5 Containment: An open group should indicate which nodes are its children, otherwise users cannot accurately present their mental model (PR2, PR3, PR4, PR5), detect bad design decisions (DR1), and context is provided to where rows come from and are removed (DR7, DR8).

The group nodes have a group icon (see Figure 4.15a) and are slightly larger than table nodes to make them easily identifiable (GN1).

We allow users to name each group such that it represents the mental concept the user has of the group (GN2). Additionally, we compute the number of data sinks/sources and the maximum number of rows using sum and extrema aggregation, respectively (see Section 4.1.3). Using this aggregated information, we create a descendant summary.

The descendant summary is placed inside of the group and uses the same colors and shapes as the table node, such that users associate the summary with the underlying graph. The number of data sources and sinks are shown using the same color and shape, while the maximum number of rows has the same shape and color as a table node (GN2 and GN3 for external entities).

The maximum number is shown to provide exploration value (GN4) and show users the color map is correct. When we do not show the maximum number of rows, we found users conclude the visualization shows incorrect information because the darkest color of the color map was not visible without opening groups.

When groups are closed, all edges inside the group remain hidden, but edges from tables within the group to tables outside of the group are still shown, such that GN3 is satisfied for table data.

Initially, we used a border to indicate containment of nodes in a group (GN5). However, when nesting groups, this is insufficient, as containment becomes unclear to users. Therefore, groups have a semi-transparent background. As a result, when groups are nested, the background of the groups becomes darker, as shown in Figure 4.16 satisfying GN5 for nested groups.

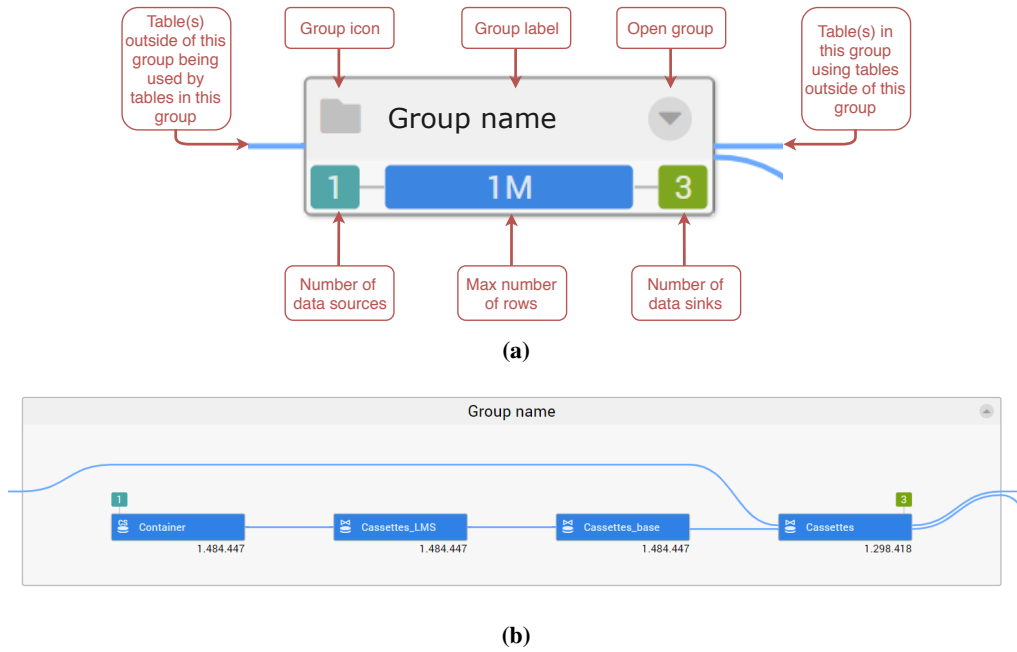


Figure 4.15: A group in its closed and opened state. (a) Showing a group icon, a user-defined label, a button to open the group, the number of data source and data sink nodes connected to tables in the group, and the maximum number of rows of the tables within the group. The incoming edges indicate tables not in this group being used by a transformation inside the group, while the outgoing edges indicate tables not in this group using a transformation inside this group. (b) When opening the group, the summary shown before is replaced by the layout of its child nodes and the groups size is adjusted accordingly.

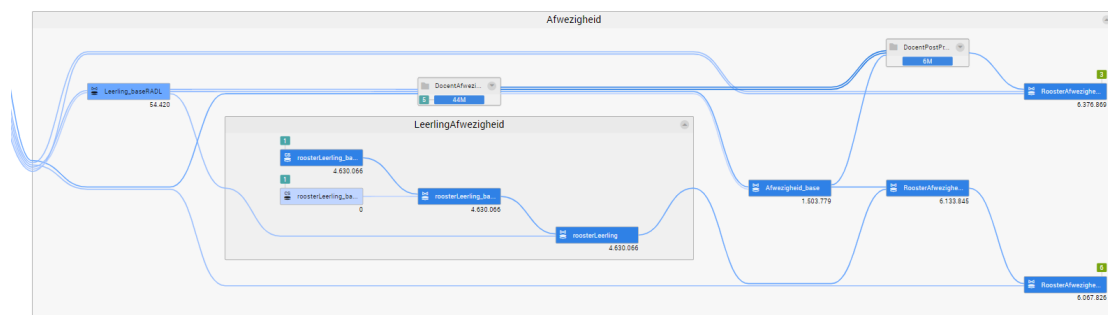


Figure 4.16: An example of a nested group node. The background of the nested group becomes darker.

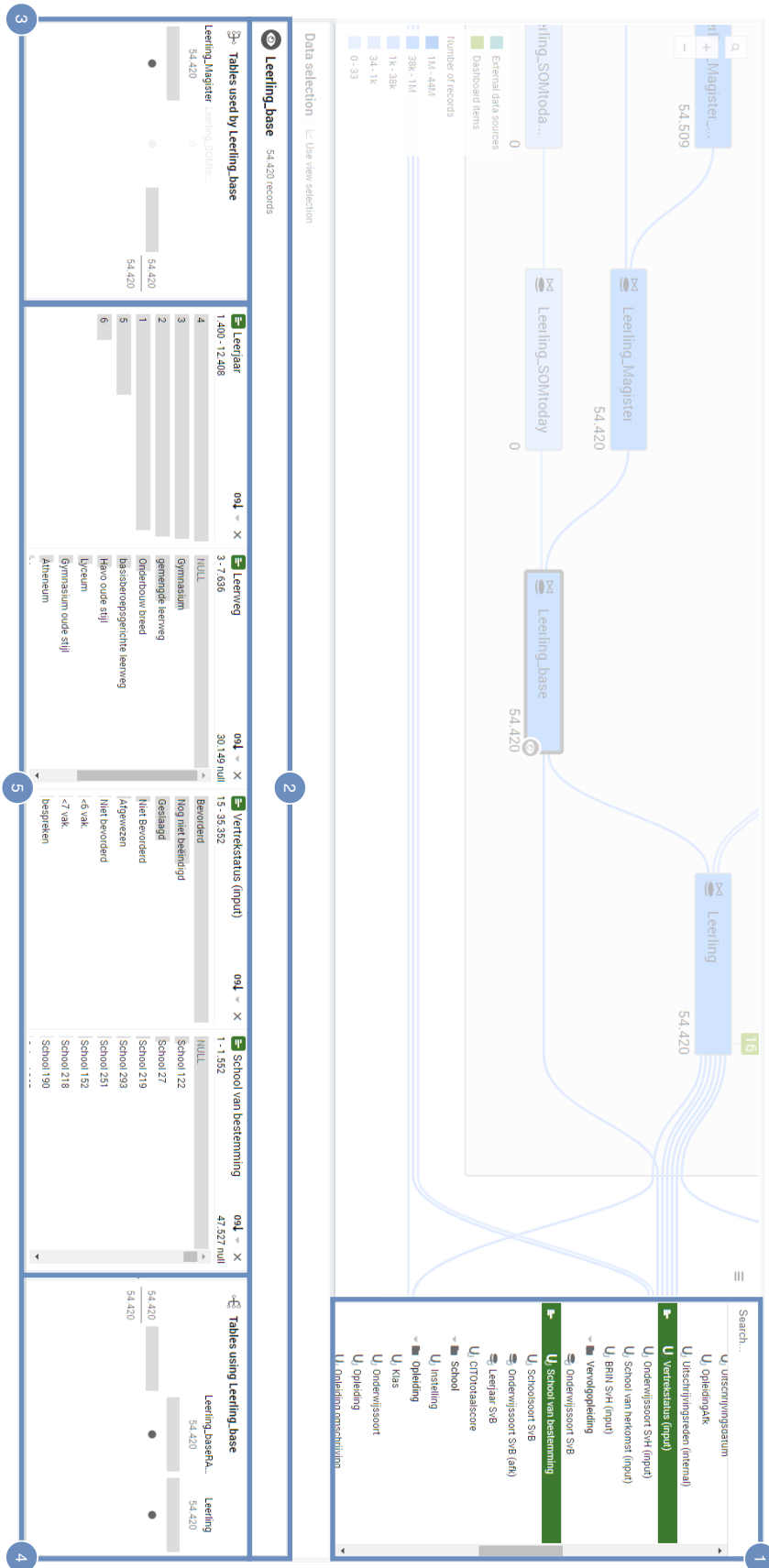


Figure 4.17: The debugging views: the column list (1), active table information (2), in/out-dependencies views (3 and 4 respectively) and value frequency views (5).

4.4 Debugging views

The debugging views enable developers to perform a dependency analysis by letting them inspect the row dependencies of a table (DR2, DR3, DR4, DR5, DR6). They also enable developers to perform a causality analysis by letting them trace rows through multiple transformations or analyze column values (DR7, DR8, DR9). Finally, these views also provide an easy means of selecting rows (DR10).

The debugging views are shown when a user clicks on a table node to investigate it further. This table node will change in appearance in the hierarchy view to allow developers to quickly scan the context in which the table is used. This is done by drawing a thick gray (●) border around the node and showing an eye icon in the bottom right of the node (see Figure 4.18). We automatically zoom the view to center the active node. We refer to the table shown in the debugging views as the *active* table (node).

There are several debugging views (See Figure 4.17). The column list and value frequency views (1 and 5 respectively) work together to enable developers to perform a column analysis (DR9). The active table information (2) shows the total number of rows and links the active table in the hierarchy view to the debugging views. Finally, the in- and out-dependency views (3 and 4 respectively) show the row dependencies of the active table, enabling developers to perform a dependency and causality analysis (DR2, DR3, DR4, DR5, DR6, DR7, DR8).

4.4.1 Column analysis

We enable developers to perform a column analysis using the column list, which shows the columns of the active table, and the value frequency views (DR9).

Amar *et al.* describes several low level tasks users wish to perform when using information visualization tools for understanding data [AES].

CA1 **Retrieve value:** given a set of rows, get their values for a given column;

CA2 **Filter:** given several conditions on the data, find data satisfying those selections;

CA3 **Compute derived value** of data;

CA4 **Find extremum:** finding the maximum or minimum of a selected data set;

CA5 **Sort:** sorting data is usually not a task in itself, but often used to accomplish finding the values at the extreme [AES];

CA6 **Determine range** of values within a column;

CA7 **Characterize distribution:** given a set of rows and a quantitative attribute of interest, characterize its distribution;

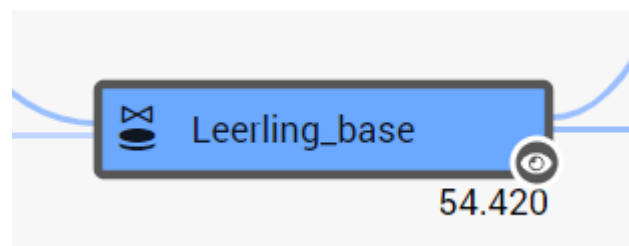


Figure 4.18: When a user clicks on a table, the node gets a dark gray (●) border and an eye icon, indicating it is shown in the debugging views. The eye icon inside of a circle is repeated in the active table information.

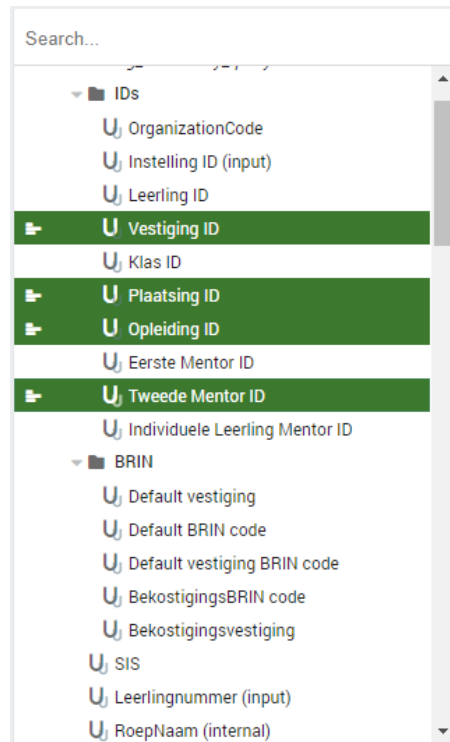


Figure 4.19: The column list view, showing the columns of the active table. The green (●) background of the row is used in conjunction with the white icon to both enable developers to quickly spot the selected columns, and makes the green color recognizable for developers as a column selection.

CA8 **Find anomalies** identify any anomalies within a given set given a relationship or expectation;

CA9 **Cluster**: given a set of data, find data with similar values;

CA10 **Correlate**: given a set of selected rows and two columns, determine relationships between them.

To ensure developers can effectively analyze their data, we enable them to perform all of these tasks except for CA3, CA6 and CA9, because transformation tools already provide the ability to perform the computation of derived values (CA3), and we found developers did not require clustering or value ranges while debugging (CA6, CA9).

The column list shows the columns of the active table (Figure 4.19), with the selected columns highlighted by a green background and a white icon in front to link these to the value frequency views. For every column, we show a value frequency view. In Figure 4.20, an example of one of these is given. It displays the following information: (1) An icon similar to the one shown in the column list, linking the two views, (2) a label showing the column name, (3) sort options (CA4, CA5), (4) a close button, (5) a label showing the range of the value frequencies (CA7), (6) the number of missing (or *NULL*) values (CA1) and, (7) the value frequencies shown as a bar chart (CA1, CA4, CA7, CA8).

Developers wish to know which values are causing certain behavior. Therefore, we aggregate the rows of each column based on their value, showing only the frequency each value occurs. This provides a clear distribution of the data (CA7) and enables more expressive selections (CA2, CA10). An example of a value frequency view is shown in Figure 4.20 (7): there are over one million rows, nearly 600 thousand with value 'T', about 400 thousand with value 'C', around 2500 with value 'B' and 'S' and 7 null values.

We use bar charts such that developers can compare the frequencies of values within a single column to one another (CA8). Each bar has a minimum of 1 pixel on the left side if a value occurs, such that developers can still retrieve values (CA1).

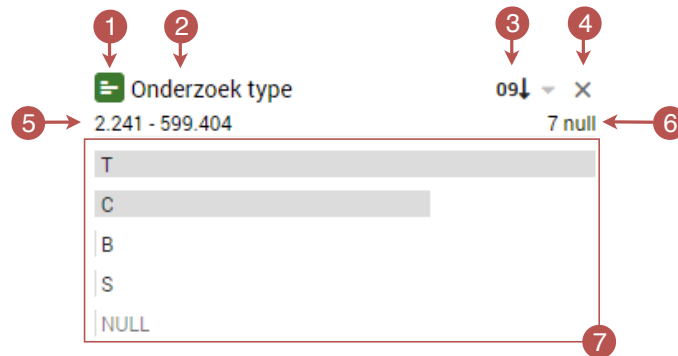


Figure 4.20: An example of a value frequency view, showing an icon that mirrors the selection in the column list (1), the name of the column (2), sort options (3), allowing for sorting on name or frequency, a close button (4), information about the range of value frequencies (5), the number of missing (or *NULL*) values (6) and, a bar chart showing the frequencies of each value (7).

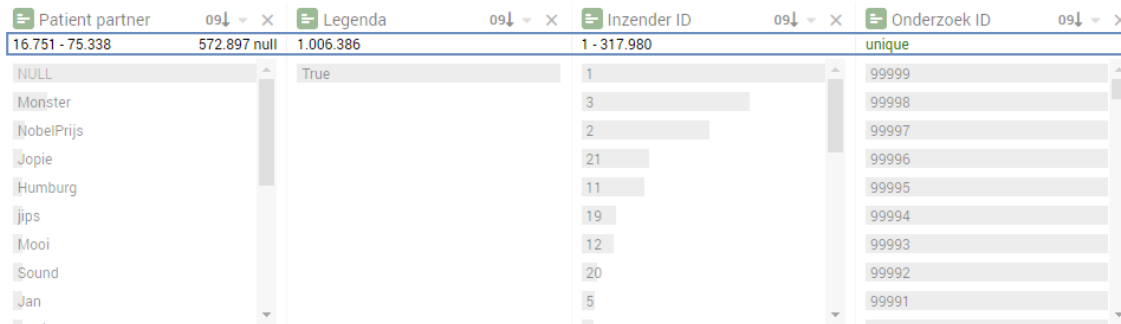


Figure 4.21: Four different value frequency views, each with different distribution information. From left to right: The *Patient_partner* column has 572.897 thousand missing values, and a range from 16.751 up to and including 75.338. The *Legenda* column has only one value, with a frequency of about a million. The *Inzender_ID* column has no missing values and the range goes from 1 up to and including 317.980. The *Onderzoek_ID* column has unique values for every row.

When there are many different values, the value frequencies are cumbersome for developers due to the large number of bars shown. The range of the value frequency however, is a question often asked by developers, therefore, sorting the bar chart to determine this range is undesirable. For example, developers want to know if each value is unique (i.e., all values have a frequency of one), or whether the range of the values frequencies lies between two numbers (CA7). This value frequency range should not include missing values, therefore, we separate these values from the frequency range and show them separately as the number of missing values are of interest to developers in many other cases (CA1).

We provide this information in a summary of the value frequency range (indicated by number 4 and 5 in Figure 4.20). A label (4) shows the frequency range. If the frequency range is $[x, y]$, the label shows "unique" if $x = y = 1$, "x" if $x = y$ and " $x - y$ " otherwise. Finally, the number of missing (or *NULL*) values is shown (5). Examples of each are shown in Figure 4.21.

4.4.2 Dependency views

The dependency views enable developers to see what data is used to create the active table (DR4, DR5, DR7), and detect where the data from the active table is being used (DR6, DR8). The dependency views essentially provide a means to detect 'groups' of rows that are processed in the same way. For example, given a data transformation graph that processes rows about high schools, developers may wish to inspect

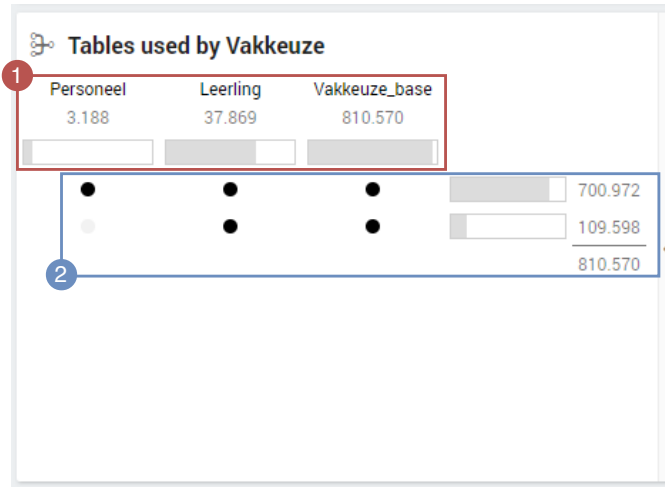


Figure 4.22: The in-dependency view of a table named *Vakkeuze* showing the tables used to create it (1). Underneath every label a number representing the number of rows used is shown, and a bar showing the percentage of of the total the number above represents. The view also shows two dependency sets (2). Each indicates a groups of rows, the black dots indicate which tables were used to produce these rows. In this case, there are two groups: the rows that use information from the *Personeel* table (700.972 of them) and those that do not (109.598 of them).

the subjects which do not have a teacher. These subjects can be found by selecting the rows in the subjects table that do not use information from the teacher table.

We provide two dependency views. One shows information about the data used to create the active table. Another shows information about the tables using the active table (the in- and out-dependency views respectively). In most cases, only one of the two is relevant, e.g., when developers wish to trace back to the origin of a set of rows (DR7), only the in-dependency view is used, while if developers wish to find where rows are removed, only the out-dependency view is used (DR8). Therefore, we allow the two dependency views to be opened and closed by the developers, providing additional space to the value frequency views.

In Figure 4.22, an example of the in-dependency view is shown. For every table being used, there is a summary (1), containing the name, the number of rows used, and a bar indicating the percentage of rows used (e.g., in the example shown in the figure, 37.869 rows from *Leerling* are used, approximately 70 percent of its total number of rows) as this helps to satisfy DR7 and DR8. Additionally, the figure shows an example of the in-dependency sets of the active table (2). The in- and out-dependency sets define the tables that the rows use or are used by respectively. For example, if the rows in table C are produced by using table A and B, there are four different dependency sets a row can have: $\{A, B\}$, $\{A\}$ or $\{B\}$ (A visual example is shown in Figure 4.23). An out-dependency set can also have \emptyset (i.e., the row is unused).

More formally, we define in- and out-dependency sets of a row as follows:

Definition 4.4.1 (In-dependency set). The in-dependency set of a given row r is the set of tables D where for every $D_i \in D$ there is a $(r', r) \in E_r$ for which $r' \in rows(D_i)$.

Definition 4.4.2 (Out-dependency set). The out-dependency set of a given row r is the set of tables D where for every $D_i \in D$ there is a $(r, r') \in E_r$ for which $r' \in rows(D_i)$.

In the dependency views, we show the number of rows that have a specific dependency set. The black dots indicate the table in that specific column is in the dependency set, the numbers on the right side indicate the number of rows that have this dependency set, and the bars show the percentage of the total number of rows. These numbers sum up to the total number of rows of the table, therefore, the sum is repeated underneath. As an example, see Figure 4.22. There are two input dependency sets used by the rows: Out of the 810.570 rows, there are 700.972 rows that use data from the input tables (i.e. the dependency set is $\{Personeel, Leerling, Vakkeuze_base\}$), and 109.598 rows which do not use information from *Personeel*,

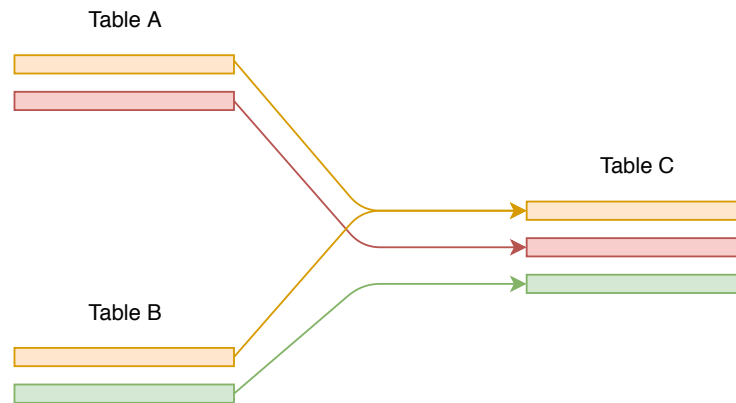


Figure 4.23: An example showing the in-dependency sets for three rows of the table C. Each color indicates a dependency. The orange (○), red (●) and green (●) colored rows have the input dependency set $\{A, B\}$, $\{A\}$, and $\{B\}$ respectively.

resulting in the input dependency set $\{Leerling, Vakkeuze_base\}$. The other dependency sets are not used, therefore, we do not show them.

Design choices

We have made several choices in the design of our dependency view, such that we are able to (better) satisfy our requirements.

Unused data: When a table is unused, the entire column becomes more transparent. When a certain permutation of a dependency set does not exist, it is not shown, as this improves DR4, DR5 and DR6.

Sorting: We sort the dependency sets by frequency, enabling developers to quickly detect how most of the data is processed.

Bars: We use bars for representing the proportion of data used, since previous research shows when visualizing more than a small number of proportions, (stacked) bars are more effective [Sii14]. In our interviews we observed that developers rarely need exact percentages, making a tool tip with the exact percentage sufficient. However, developers do want to know if bars are empty or full (DR7, DR8, DR11), therefore, we use disproportional bars. These bars are given several pixels of margin if they are not strictly equal to 0 or 100 percent. A comparison between bars is shown in Figure 4.24. Alternatively, we could have used icons to convey the same message. However, icons increase the clutter the visualization and require developers to learn them. When selections are added, these bars become stacked bars, with a blue color (●) indicating the selected proportion of the data.

View orientation: The output dependency view (4 in Figure 4.17) is similar, except for the row counts and bar, which are aligned on the left instead of the right. We change the alignment for two reasons, first, the values of the active table can be inspected in the middle of the two dependency views, therefore, placing the aggregation of the rows closer to this view causes developers to associate them with the rows. Secondly, it places the tables using the active table on the right side. Since every table that uses the active table results in an edge on the right side of the active table, this improves the link between the hierarchy view and the dependency view.

The in-dependency view fulfills DR3, DR4 and DR5. The out-dependency view fulfills DR6. The in- and out-dependency views partially fulfill DR7 and DR8, respectively, as data selections are required to completely fulfill these.

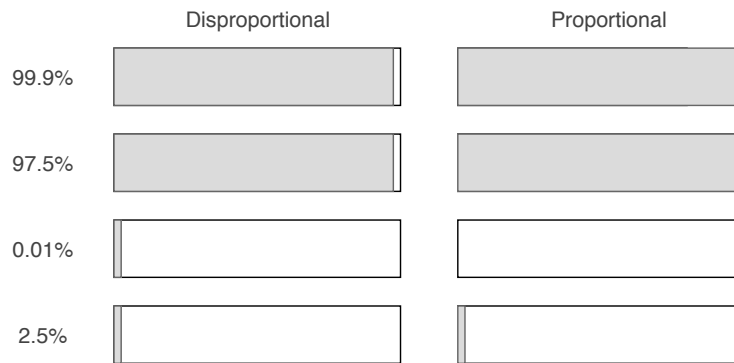


Figure 4.24: Comparison between the two types of bars used. The disproportional bar still shows a few pixels when the percentage is almost a hundred or zero percent, while the proportional does not.

4.5 Interaction

Interaction plays a major role in enabling our users to perform their tasks. In our visualization, we have two methods through which users can interact with the visualization: **navigation** and **selection**. We discuss these in the following sections.

4.5.1 Navigation

We enable users to navigate through the hierarchy using panning, zooming, and opening and closing groups. Additionally, the dependency views (see Section 4.4.2) enable developers to navigate to the neighbors of the active table. To smooth interaction as much as possible, we automatically zoom on some interactions.

Panning and zooming in the hierarchy view is done by holding the left mouse button and dragging the mouse, while zooming can be done using the mouse wheel.

Neighbor navigation is possible in the dependency views (see Section 4.4.2). When the label of the neighboring table in the view is clicked, we zoom and pan to this neighbor and set it to the new active table. This allows developers to follow complex paths easier, as they do not have to find neighbors in the hierarchy view. When the active table is not visible, the group(s) it is in will automatically open.

Opening and closing groups is possible by clicking on the open and close button of a group respectively (see Figure 4.15a). When a group is opened, we zoom to the group that was just opened. When a group is closed, we zoom to the parent of closed group. We apply this automated zooming as we observed that during presentation tasks, users usually perform this action after opening and closing.

4.5.2 Selection

Developers wish select part of the data (DR10) such that they can perform a causality analysis (DR7, DR8, DR9). Additionally, after they found the cause of an anomaly, they wish to compare this to the regular behaviour of the data (DR11). Based on that, we define several requirements on selections:

SR1 Linkage: To improve the ability of the hierarchy views to provide context to detailed causality analysis (DR7, DR8, DR9) the active selection should be reflected in both the debugging views and the hierarchy view.

SR2 Undo and replay: It is rare that every action produces a desirable outcome. Since many steps are required during exploration (DR7, DR8, DR9), we should enable users to perform an undo and redo and provide them with a history of previous actions [Shn96].



Figure 4.25: An example of the selection history showing a selection. (a) Shows the selection in its entirety. (b) The user went back three steps, indicated by the increased transparency.

Selection history

To assist developers in understanding the actions they performed and their resulting selections, we show them a history of every action taken (see Figure 4.25 for an example), and enable them to go back and forth between the selections by clicking on them (SR2). Additionally, some of the selection operations are shown in the selection history (the invert operation, the data sink selection, and an option to clear the current selection).

We found developers expect selections to be applied on all views below the selection history. Therefore, we position the selection history above all of the debugging views showing selections (all debugging views except for the column list). All selection actions are shown as a separate step in the selection history. An example of each is shown in Table 4.1.

Selection types

We define a selection on a table as a subset of its rows. More formally, given a data transformation graph $G = (V, E)$, a selection R on a node $n \in V$ is a subset of rows of the corresponding table, i.e., $R \subseteq rows(n)$.

Selecting individual rows is cumbersome and hard to reason about. Therefore, we provide developers with several interactions that select groups of rows. Each is shown in Table 4.1 (DR10). We consistently use the same icons for these throughout our visualization (SR1).

Although it is possible for developers to perform the same selection on the same edge or node multiple times, we do not display this in a different manner in our hierarchy view. In the cases where such a complex selection is required, the selection history enables the developer to perform this task (see 4.5.2).

In- and out-dependency selection: Once a developer has detected an anomaly in the in- or out-dependency view, the natural next step is to select the anomalous case. Therefore, we enable developers to select all rows with the same dependency set. We represent this selection in the hierarchy view using the same dots as those the dependency views (shown in Table 4.1).

Used by and using selection: Developers want to inspect where data comes from and where it is removed (DR7, DR8), therefore, we enable developers to perform the *used by* and *using* selection. The *used by* and *using* selection select the rows in a new active table that were used to create or were using the selected rows in the currently active table, respectively. This selection can be performed by clicking on a neighboring node in the hierarchy view or in one of the dependency views. This selection is also represented in the hierarchy view, as shown in Table 4.1. An example of the used by selection is shown in Figure 4.26.

We only define *used by* and *using* selections for neighboring nodes, as defining these selections on two arbitrary nodes in the graph can lead to multiple paths existing and would require a method of visualizing and choosing between them.

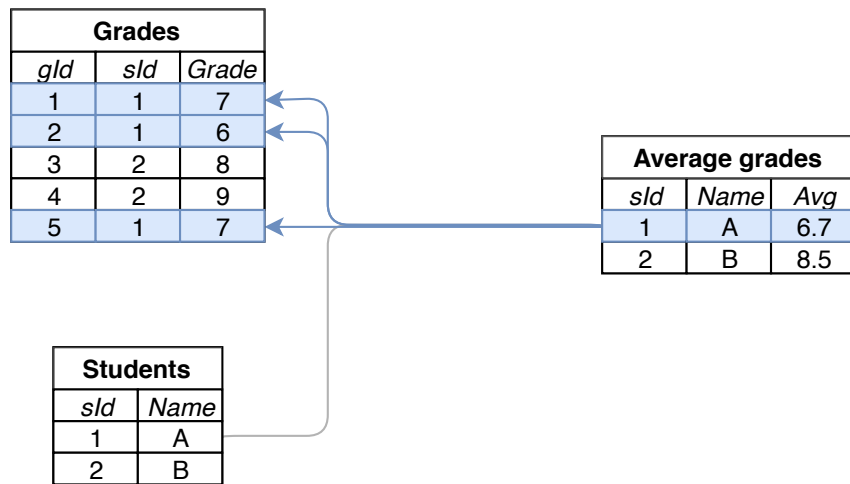


Figure 4.26: Given the row dependency graph shown in Figure 2.4c, where the table **grades** and **student** are used to create **average grades** and a selection of student 1 on the **average grades table**. A used by selection on **grades** selects all rows in **grades** used to compute this average.

Value selection: Developers wish to select a combination of one or several values that they deem interesting (DR9), therefore, we allow developers perform a value selection by clicking on a row in a value frequency view. We then remove all rows not having the selected value from the active selection. When one or multiple value selections are performed on a node, this is indicated by the node by showing the value selection icon underneath the node (see Table 4.1).

Data sink selection: Developers sometimes receive reports of bugs from external parties (i.e., those using the data produced for the data sinks). These bug reports are often accompanied by a set of rows that are in error, which we enable developers to select using the *data sink selection*. Similar to the value selection, when a data sink selection is performed, we show the respective icon underneath the node (see Table 4.1).

Invert operation: Developers sometimes wish to perform excluding selections, i.e. given a selection, they want to select the inverse of that selection. Therefore, we provide an inverse operation, which inverts the current selection.

In contrast to the other selections, we do not show invert selections on the hierarchy view. The reason for this is twofold: First, unlike the others, the inverse operation does not reduce the selection making it is a different type of operation. Additionally, inverse operations have a different result depending on the order in which the selection is applied. Hence, showing them in the hierarchy view without information about the order can cause developers to draw incorrect conclusions. Instead, the selection history (see 4.5.2) provides a far clearer, unambiguous solution for visualizing the invert operation (shown in Table 4.1).

4.5.3 Selections in the debugging views

Whenever a selection is performed, the debugging views will change accordingly (shown in Figure 4.27). Instead of the the gray bar indicating the proportion of data used, a stacked bar is shown in the dependency views, showing the proportion of data selected using a blue color (●). Similarly, stacked bar charts are shown in the value frequency views. However, a lighter blue (●) is used instead, improving contrast with the black text showing the values. In addition, all numbers in both the dependency view and the distribution information in the frequency views will show information about the selected rows instead of all rows. To indicate this, all labels change their color to blue (●). Finally, the total number of selected rows is shown in the table information.

Selection name	Description	Icon	Hierarchy view	Selection history
<i>In-dependency</i>	Select rows in the active table that come from a selected set of tables.			
<i>Out-dependency</i>	Select rows in the active table that are used by a selected set of tables.			
<i>Used by</i>	Select rows in a new active table that were used to produce the selected rows in the current active table.			
<i>Using</i>	Select rows in a new active table that were using the selected rows in the current active table.			
<i>Value</i>	Select a value in a selected column.			
<i>Data sink</i>	Select rows from an external data sink (e.g., views in ProcessGold)			
<i>Invert</i>	Invert the current selection.		n/a	

Table 4.1: A summary of the selections we enable developers to perform.



(a) The debugging views before a selection is made.



(b) The same views with a selection active. The dependency views show the percentage of rows that are selected. Every value frequency view which part of the values are selected. All selections are shown in blue (●). For the value bar charts, a lighter blue (●) is used to improve contrast with the black text.

Figure 4.27: The effect of selections on the debugging views.

Chapter 5

Evaluation

In this chapter we discuss how we evaluated the effectiveness of our visualization. We discuss two different case studies where our visualization is used to assist users in their maintainability tasks. Then, we discuss the qualitative evaluation we performed. Finally, we summarize the results of these two evaluations and discuss them.

5.1 Case studies

We use two case studies to evaluate effectiveness of our approach. These two real-world problems at ProcessGold, one focused on presentation and the other on debugging, are solved by making use of our visualization.

5.1.1 Case study 1: Creating a proposal & presenting changes to a customer

A customer (i.e., application user) wants to move part of the data transformation graph, currently maintained by ProcessGold, to their own data centers (their so called data mart and data lake). In response, a developer at ProcessGold wants to propose what part of the data transformation graph can be easily transferred and discuss where responsibilities between the two parties lie.

The developer uses our visualization to achieve this goal. We expect the developer to first explore the initial hierarchy (PR2) to gain a better understanding of the structure of the data. After the developer understands the structure sufficiently, we expect the developer to adjust the hierarchy tree to create two separate groups (PR1). One that indicates the transformations easily moved to the data centers and one indicating the transformations better kept in the ProcessGold platform. Finally, we expect the developer to present and adjust the proposal to the application users until they both agree on a solution (PR3, PR4).

Results

The developer, as expected, starts with exploring the initial hierarchy (shown in Figure 5.1a). Groups are individually opened, explored, and closed again. Once the developer understands the majority of the data transformation graph, a new hierarchy is created (shown in Figure 5.1b). This hierarchy splits the data into four groups: two groups kept by ProcessGold (the *connector* and *output* group) and two indicating the transformations that can be moved to the customers' data centers (the *data lake* and *data mart* group). The developer starts reordering the hierarchy by dragging and dropping the transformations into their respective groups until satisfied.

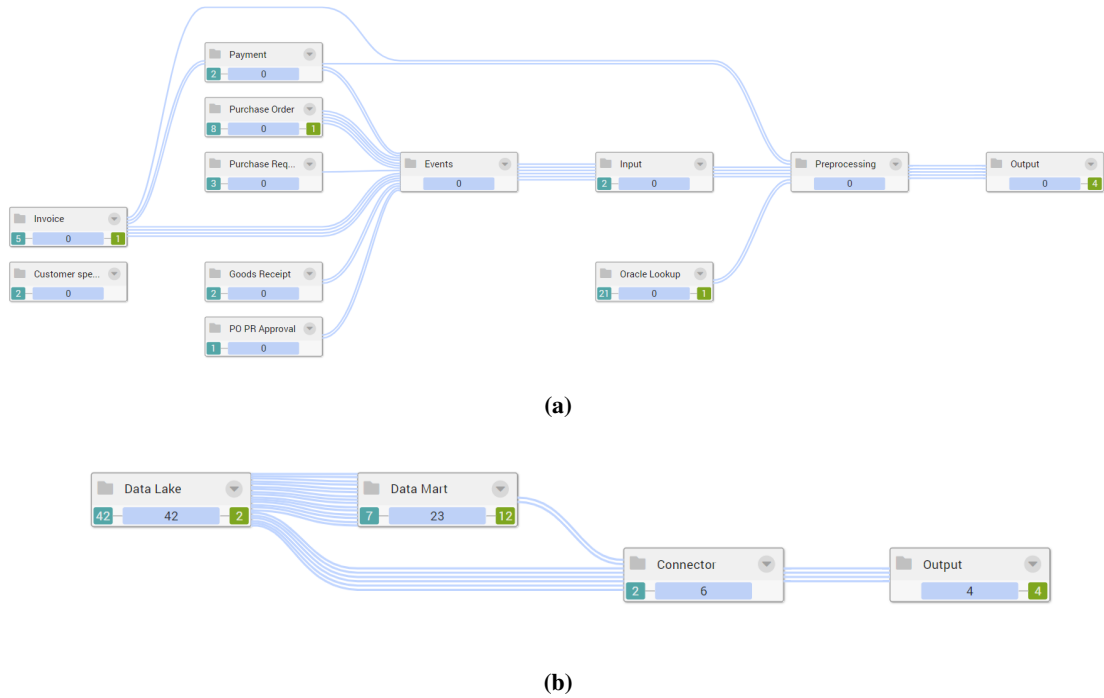


Figure 5.1: (a) The hierarchy before the developer made any changes. (b) The hierarchy created to facilitate presentation to application users.

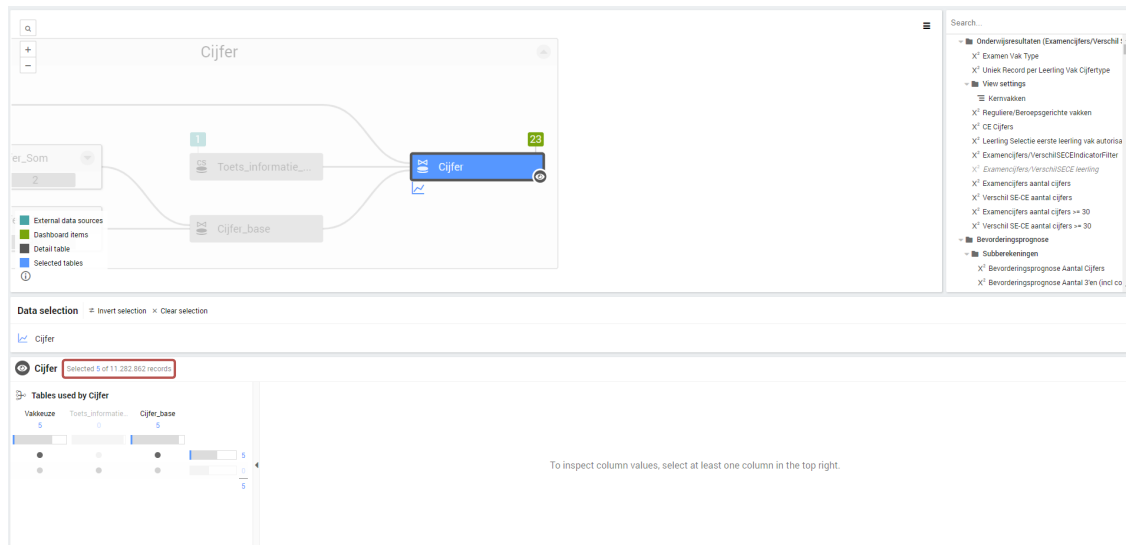
The developer then goes to the customer, and using the hierarchy, the developer discusses each group in detail, discussing individual leaf nodes and why they were placed inside of the group. The dependencies between the two parties were then clear. All data used by the *connector* group are the responsibility of the customer.

Using our visualization the developer is able to quickly create a proposal and adapt it while discussing changes with the customer. Because the visualization ensures the validity of the dependencies no matter what mental model is created, it allows both parties to focus on collaborating to create a unified mental model. Therefore, we can state the visualization satisfies [PR1](#), [PR2](#), [PR3](#), and [PR4](#).

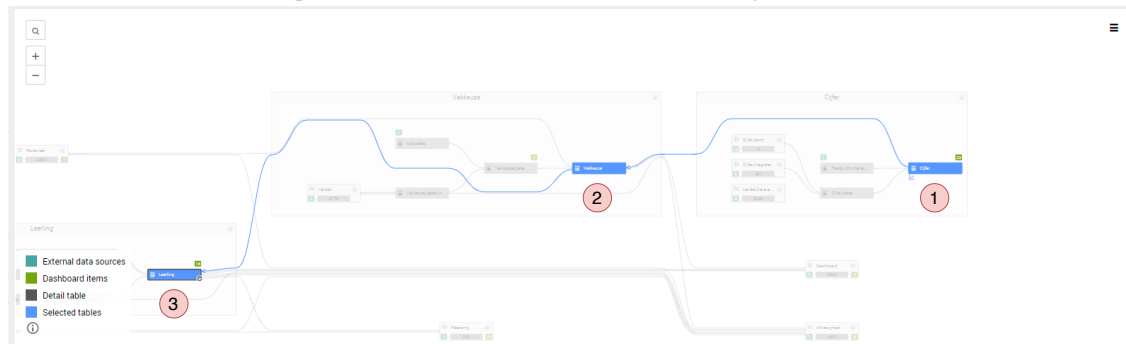
5.1.2 Case study 2: Debugging

An application designed to provide insight into high school data has a major change applied to its data transformation graph. This change involves reworking a large part of the transformations used to compute student grades. As a result, many issues arise in the application. Our visualization is used to find the cause of these issues: Before the change, all data was removed when applying certain filters. However, after the change, there was information left, even though the same filters were applied.

A hierarchy was previously created that represents the mental model of the developer. Using this hierarchy, we expect the developer to perform a causality analysis on the affected rows using several selections, and compare the affected data to the rest of the data to establish whether it diverges from normal behavior ([DR7](#), [DR10](#), [DR9](#), [DR11](#)). After this, we expect them to apply a change that resolves this issue, or reason that the new behavior is correct.



(a) The developer selects five rows from the visualization using the *data sink* selection.



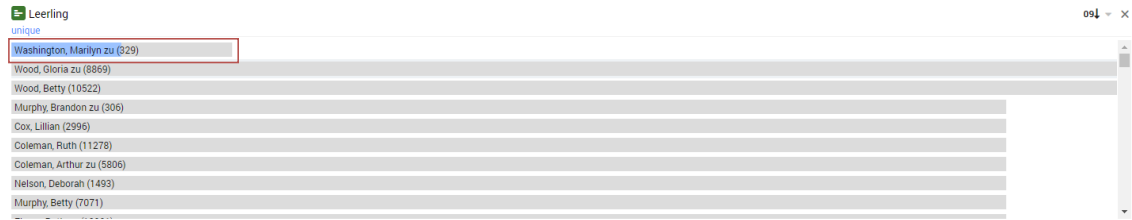
(b) The developer follows these rows back to a single row in the student table using the *used* selection. Starting at (1) the user applies two *used* selections to arrive at (2) and (3), respectively.

Figure 5.2: The user first tries to find what students produced the affected grades.

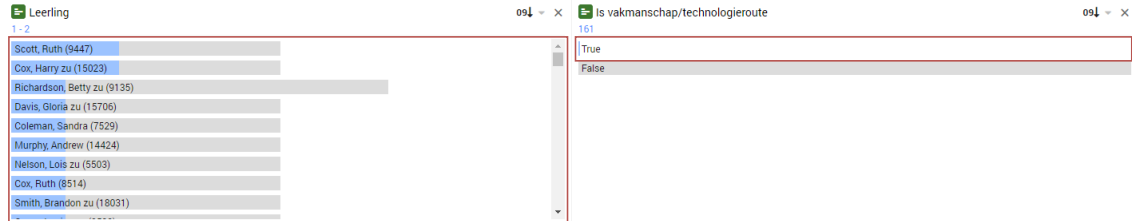
Results

The developer uses the *data sink* selection to select the five rows that they expected to be unavailable (Figure 5.2a, DR10). Then, they use the *used* selection to trace the rows back to the student table (Figure 5.2b, DR4, DR7). Here, the developer sees a single student causes the behavior. The developer looks at the value frequencies (DR9), and concludes that there are two rows in the student table for this one student, one of which states the student was part of the technology track, while the other states the student was not (Figure 5.3a). The developer then reasons why this could occur, and verifies their reasoning. Then, the developer wants to see whether this is a unique case, or if this occurs more often, such that a correct priority can be assigned to resolving the error. Therefore, the developer selects all rows of students that were part of the technology route and, using the value frequency view, concludes there are several students that exhibit this behavior (Figure 5.3b, DR11).

The developer attempted to find the cause of the issue for several hours without the visualization. With the visualization, they were able to find the cause within ten minutes. They even commented they might not have found the bug without the visualization. Therefore, we can conclude that the visualization satisfies DR4, DR7, DR9, DR10, DR11.



(a) The developer used a value frequency view to find out that this student had multiple rows in the table, and that only one of these rows stated the student was part of the technology track.



(b) By selecting all the students part of the technology track (the right value frequency view), the developer finds that several students (the left value frequency view) have both a row showing they are part of the technology track and another that shows otherwise.

Figure 5.3: The user explores student data.

5.2 Qualitative evaluation






We have performed a qualitative evaluation, interviewing eight developers, ranging from new developers, having little experience with maintaining data transformation graphs, up to seasoned developers with over ten years of experience. We provided them with the same presentation and debug task, providing us with a baseline to compare the interviews to another. When they encountered anything they found interesting outside of these two tasks, they were asked to explore this as well. We observed the developers while performing these tasks, and asked them to report any observation they made and actions performed.

We concluded with an interview asking them whether the visualization meets the requirements we posed in our problem description (See Section 3.3). Additionally, we asked them how effectively they can perform presentation and debugging tasks using our visualization. In general, the developers found presentation tasks to be very easy to perform. But were more split on debugging tasks. We provide a summary of their answers for presentation and debugging tasks in Table 5.1 and Table 5.2, respectively.

We found users are very enthusiastic about the ability to use their mental models to present their transformation graph, and indicate it is an effective tool for presenting their transformation graphs to other developers or application users. Collaboration on a unified mental model was received positively as it would help in making better design choices and might replace part of their documentation. We also found our tool has two limitations in terms of presentation. For their first presentations, some users want to hide certain parts of the transformation graphs to reduce the complexity. Additionally, while making design choices, users wanted to visualize proposed design choices, without having to execute them, e.g. by drawing on the graph.

Users found our visualization effective at detecting errors in their design, row changes and data duplication. All users found the in-dependency view very clear as it enabled them to clearly see how rows were combined to form a new table. However, the out-dependency view, although similar to the in-dependency view, was more difficult to understand for our users.

We found users can effectively find where rows come from or are removed, however, finding out why rows arrive somewhere or are removed requires a detailed analysis of the transformations, which our visualization does not provide sufficient capabilities for. Most users therefore also indicated they would have liked more extensive selection methods on columns, as they can both help in understanding why changes occur

Requirement	Avg	Remarks
PR1 Create mental model		All developers found creating mental models intuitive.
PR2 Present to developer		Some prefer to use a whiteboard/paper when presenting the transformation graph for the first time to reduce initial visual load. All of them stated they would use the visualization to support further presentation, as it shows the correct dependencies.
PR3 Present to application user		The developers agreed that the visualization is an effective means to explain the transformation graph to technical application users.
PR4 Unify mental model		Developers indicated the visualization would be able to unify their mental model. Some suggested it might even be able to replace part of their documentation.
PR5 Make design choices		Developers commented the visualization would improve their ability to make design choices. However, some indicated the visualization would still have to be supported by a whiteboard or other tools, as they would like draw their proposed design changes without having to execute them.

















 Fully agree
  Mostly agree
  Neutral
  Slightly disagree
  Fully disagree

Table 5.1: A summary of the presentation requirements evaluated during our qualitative evaluation. We show an average of how much developers agree that the visualization satisfies each requirement, and provide their most important remarks.

in a transformation and makes some debugging cases easier.

Most users found selecting data easy and intuitive, requiring little to no explanation. However, some had difficulty interpreting the selection history. Users were positive about the ability to redo and undo selections. Finally, comparing selections to the rest of the data was very intuitive for users.

Requirement	Avg	Remarks
DR1 Detect bad design		Developers found the need for domain knowledge of the transformation graph is reduced, allowing them to detect bad design choices quickly.
DR2 Detect row changes		Developers found the detection of row changes easy.
DR3 Detect duplication		All developers could find data duplication rapidly, however, some developers commented that small amounts of data duplication could be found, but would take more time.
DR4 Detect merge		All developers could easily detect merge changes, using the in-dependency view.
DR5 Detect disjoint		All developers were able to detect disjoint tables rapidly.
DR6 Detect split		Some developers required a short explanation before they understood the out-dependency view. Notably, split changes were harder to understand than merge changes.
DR7 Where do rows come from		Tracking rows back to where they come from was intuitive for all developers. Developers commented that although finding where bugs occur was intuitive, finding out why it occurs was harder, as it requires them to understand the transformation in detail.
DR8 Where are rows removed		Tracking rows to where they are removed was intuitive for all developers. Finding out why rows are removed is more difficult.
DR9 Column analysis		Column analysis was acceptable to all developers. However, they indicated a more extensive column analysis, providing additional selection methods, may be beneficial for some debugging cases.
DR10 Select data		Most developers found performing selections easy and intuitive, however, some had difficulty interpreting the selection history.
DR11 Compare data		Comparing a selection to the rest of the data was easy.






 Fully agree
  Mostly agree
  Neutral
  Slightly disagree
  Fully disagree

Table 5.2: A summary of the debugging requirements evaluated during our qualitative evaluation. We show an average of how much developers agree that the visualization satisfies each requirement, and provide their most important remarks.

5.3 Conclusions & Discussion

To evaluate our approach, we applied it on several real world examples at ProcessGold. The data transformation graphs we used are in between ten and a hundred nodes, with approximately the same number of edges. For these, we used the data lineage of data sets having an average of 3 million data rows per table, with a maximum of 45 million rows. Computation of the information required to apply our technique takes less than a few hundred milliseconds for tables with 3 million data rows, up to approximately eight seconds for 45 million data rows. To maintain responsiveness of the visualization with larger data sets, increased computation power would be required.

During our use cases, we were able to verify our approach satisfies most of our requirements (PR1, PR2, PR3, PR4, DR4, DR7, DR10, DR11). By making use of a qualitative evaluation, we found our approach also satisfies the requirements not covered by the use cases (PR5, DR1, DR2, DR3, DR5, DR6, DR8).

Although our debugging use case shows that our column analysis is effective, our qualitative evaluation shows it is lacking (DR9). A likely reason is the simplicity of the column analysis in the use case. Additionally, although the developer of our debugging use case had no trouble understanding selections (DR10), our qualitative evaluation shows that others do. Finally, the qualitative evaluation shows that developers have difficulties understanding why changes occur in transformations, especially when they lack domain knowledge about the transformation graph.

In conclusion, we find our approach helps developers to improve the maintainability of their data transformation graphs. We found that creating mental models that can be used by both developers and application users is very valuable, as it reduces the need for extensive domain knowledge to perform maintenance tasks and improves collaboration between users. The debugging functionality our visualization provides is able to save a lot of time when used in real life scenarios. However, due to the complexity of selections and limiting column analysis, some developers may find it harder to adopt our approach.

Chapter 6

Conclusions

In this thesis, we presented an overview of existing work in the field of data flow, data lineage, and how maintainability of data transformation graphs is approached in both visualization research and the industry. From this overview and interviews with domain experts we identified that current approaches are lacking in presentation capabilities, since they require a lot of maintenance to maintain the mental model of users, and do not take mental map preservation into account. Additionally, debugging capabilities of these approaches are also lacking, due to tools being primarily focused on attribution or debugging with coarse-grained lineage information. Improving these can improve the maintainability of data transformation graphs.

Our novel approach improves maintainability by enabling users to recreate their own mental models [WFS93, KMCA06] using a graph hierarchy. This hierarchy is presented to other users through a custom-made, mental-map-preserving, layered graph layout adopting the Sugiyama framework [STT81]. Through such presentation, users can create a unified understanding of their transformation graph. Using this unified understanding, they can make effective design decisions that improve long term maintainability. Additionally, developers can find errors by either exploring the graph hierarchy or through the visualization of the lineage information of a transformed data set. Then, using several types of selections and a column analysis, developers are able to perform a causality analysis to find out where errors are caused.

In this work, we evaluate our approach using a qualitative analysis and two real-world use cases. These show that our approach is very effective for presentation tasks and has the potential to save a lot of time for debugging tasks. Our qualitative analysis with eight domain experts show similar results. Users are very satisfied with the approach, and some already adopted the visualization in their day to day tasks.

There are however, several limitations to our approach. First off, we found that presentation capabilities can be improved even further. Users would like additional control over what is shown in the visualization such as hiding parts of the graph, or constraining node positions. Additionally, they would like the ability to draw on the graph, such that they can propose new design choices. Debugging capabilities can also be improved. We currently provide a limited set of selections on column values, which are not ideal for some debugging cases. Additionally, our visualization does not provide a clear reason why certain behaviour is occurring in a transformation. Finally, our approach is limited to the software at ProcessGold, a more general implementation would enable a more diverse set of users to be evaluated.

In this work, we find that by letting users create their mental model in a hierarchy, we can unify their mental models through presentation (RQ1). This unified hierarchy can then be used to support users in making design choices (RQ3), and when used in combination with data lineage, it also provides an effective means to find errors (RQ2). Using this approach, we can effectively improve the maintainability of data transformation graphs.

6.1 Future work

Tools such as ours are never complete. Additional features that add more value can always be found. We have identified the following pieces of future work:

- **Combining column and row dependencies:** exploring columns dependencies in addition to row dependencies could lead to a better understanding of what is happening to the data, improving maintainability even further. For example, we could visualize column dependencies within a table by showing tables as a group that contains column nodes and their dependencies, enabling users to see why transformations behave in a certain way;
- **Extensive column analysis:** our value frequencies provide some means to perform a column analysis. However, as we also concluded in our evaluation, an improved analysis will likely improve maintainability even further. For example, one could show small previews of value frequencies or other aggregated information about the current selection next to each attribute. The challenge here lies not only in visualization but also in ensuring interactivity remains fast when using large data sets.
- **Implementation outside of ProcessGold:** This approach could be used in the implementation of other, more generic tools. For example, one can apply this approach on data transformation graphs built upon RDBMS systems;
- **Increased presentation capabilities:** our visualization is limited to showing the current state of the transformation graph. When presenting, users would like to hide parts of the transformation graph, or draw onto the visualization;
- **User defined layouts:** our visualization is currently limited to placement of nodes based on our graph layout. However, we found users sometimes want to place certain nodes next to, or directly underneath one another. We can enable this by letting users add constraints to the graph layout, e.g., by enforcing the slot of a node on a neighboring rank to be equal to their own. The difficulty here lies in visualizing and maintaining these constraints to the user, even if the transformation graph changes;
- **Used by / using selection propagation on arbitrary nodes:** users sometimes want to find the propagation of rows between two nodes in their transformation graph, without the intermediate steps. However, enabling such selections can lead to ambiguity, as there may be multiple paths available between the two nodes. If users can quickly select one of these paths it may allow them to perform traceability queries faster. For example, one could show the available paths in the hierarchy view and let users select one of them. The difficulty here lies in making sure the solution is scalable, as it may involve a large number of partially overlapping paths.

Bibliography

- [ACT06] Bogdan Alexe, Laura Chiticariu, and Wang Chiew Tan. SPIDER: a Schema mapPIng DE-buggeR. In *VLDB*, page 1179–1182, 2006. 9, 10
- [AES] Robert Amar, James Eagan, and John Stasko. Low-level components of analytic activity in information visualization. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.*, pages 111–117. IEEE. 41
- [AHK06] James Abello, Frank Van Ham, and Neeraj Krishnan. ASK-GraphView: A Large Scale Graph Visualization System. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):669–676, 9 2006. 23
- [AKN16] Hakim C. Achterberg, Marcel Koek, and Wiro J. Niessen. Fastr: A Workflow Engine for Advanced Data Flows in Medical Image Analysis. *Frontiers in ICT*, 3, 2016. 9, 10
- [Alo] Aloomo. Aloomo — Enterprise Data Pipeline Platform. 10
- [AMA08] Daniel Archambault, Tamara Munzner, and David Auber. GrouseFlocks: Steerable Exploration of Graph Hierarchy Space. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):900–913, 7 2008. 12, 23
- [Apa] Apache Airflow (incubating) Documentation — Airflow Documentation. 1, 10
- [AWMK17] Kareem S Aggour, Jenny Weisenberg Williams, Justin Mchugh, and Vijay S Kumar. Colt: Concept Lineage Tool for Data Flow Metadata Capture and Analysis. *Proc. VLDB Endow.*, 10(12):1790–1801, 8 2017. 10
- [Bar69] Margaret E Baron. A Note on the Historical Development of Logic Diagrams: Leibniz, Euler and Venn. *The Mathematical Gazette*, 53(384):113, 1969. 12
- [BB99] PA Bernstein and Thomas Bergstraesser. Meta-data support for data transformations using Microsoft Repository. *IEEE Data Engineering Bulletin*, 22(1):10–15, 1999. 6
- [BD07] Michael Balzer and Oliver Deussen. Level-of-detail visualization of clustered graph layouts. In *Asia-Pacific Symposium on Visualisation 2007, APVIS 2007, Proceedings, 2007*. 12
- [BM01] Oliver Bastert and Christian Matuszewski. Layered Drawings of Digraphs. In *Drawing graphs*, pages 87–120. Springer, Berlin, Heidelberg, 2001. 8
- [BPF14] Benjamin Bach, Emmanuel Pietriga, and Jean Daniel Fekete. GraphDiaries: Animated transitions and temporal navigation for dynamic networks. *IEEE Transactions on Visualization and Computer Graphics*, 20(5):740–754, 5 2014. 12, 26
- [BS90] B Berger and P Shor. Approximation Algorithms for the Maximum Acyclic Subgraph Problem. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 236–243, 1990. 27

- [BYB⁺13] Michelle A. Borkin, Chelsea S. Yeh, Madelaine Boyd, Peter Macko, Krzysztof Z. Gajos, Margo Seltzer, and Hanspeter Pfister. Evaluation of Filesystem Provenance Visualization Tools. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2476–2485, 12 2013. 10, 12
- [CFS⁺06] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. VisTrails. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, SIGMOD '06, page 745, New York, NY, USA, 2006. ACM. 1, 10, 11
- [Clo] CloverETL. CloverETL Rapid Data Integration. 1, 9, 10
- [Cod90] E F Codd. *The Relational Model for Database Management : Version 2*. Addison-Wesley; 1 edition (April 1, 1990), 1990. 4, 5
- [CPC09] Christopher Collins, Gerald Penn, and Sheelagh Carpendale. Bubble Sets: Revealing Set Relations with Isocontours over Existing Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 15:1009–1016, 11 2009. 12
- [CSR⁺14] Peter Chapman, Gem Stapleton, Peter Rodgers, Luana Micallef, and Andrew Blake. Visualizing sets: An empirical comparison of diagram types. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8578 LNAI, pages 146–160, Melbourne, Australia, 2014. Springer. 12
- [CW00] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 367–378. IEEE Comput. Soc, 2000. 6, 9
- [CW03] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *International Journal on Very Large Data Bases*, 12(1):41–58, 2003. 4, 6, 9
- [CY91] Peter Coad and Edward Yourdon. *Object-oriented analysis*. 1991. 3
- [De12] Soumyarupa De. *Newt: An Architecture for Lineage-based Replay and Debugging in DISC Systems*. PhD thesis, University of California, 2012. 6, 9
- [DeM79] Tom DeMarco. Structure Analysis and System Specification. In *Pioneers and Their Contributions to Software Engineering*. 1979. 3
- [DvKSW12] Kasper Dinkla, Marc J. van Kreveld, Bettina Speckmann, and Michel A. Westenberg. Kelp Diagrams: Point Set Membership Visualization. *Computer Graphics Forum*, 31(3pt1):875–884, 6 2012. 12, 13
- [EF10] Niklas Elmqvist and Jean Daniel Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):439–454, 5 2010. 12, 23, 24
- [ELS93] Peter Eades, Xuemin Lin, and W F Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47:319–323, 1993. 27
- [EW94] Peter Eades and Nicholas C Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 4 1994. 8, 28
- [FR91] Thomas M.J. Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991. 8
- [GA09] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *Proceedings - International Conference on Data Engineering*, pages 174–185. IEEE, 3 2009. 9

-
- [Gho04] M Ghoniem. A comparison of the readability of graphs using node-link and matrix-based representations. In . . . , 2004. *Infovis 2004*. . . . , pages 17–24, 2004. 10
- [Gra16] Graphviz. Graph Visualization Software, 2016. 26
- [GS77] Chris Gane and Trish Sarson. *Structured Systems Analysis: Tools and Techniques*. Improved System Technologies, 1977. 3
- [HDBL17] Melanie Herschel, Ralf Diestelkämper, and Housseem Ben Lahmar. A survey on provenance: What for? What form? What from? *VLDB Journal*, 26(6):881–906, 12 2017. 6, 9, 16
- [HPvD11] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 451, 5 2011. 10, 11
- [HQGW92] Nabil I. Hachem, Ke Qiu, Michael A Gennert, and Matthew O. Ward. Managing derived data in the Gaea scientific DBMS. Technical report, 1992. 6
- [IMMS09] Takayuki Itoh, Chris Muelder, Kwan Liu Ma, and Jun Sese. A hybrid space-filling and force-directed layout method for visualizing multiple-category graphs. In *IEEE Pacific Visualization Symposium, PacificVis 2009 - Proceedings*, pages 121–128. IEEE, 4 2009. 12
- [Inf] Informatica. Informatica - Data Transparency With End-to-End Data Lineage on Hadoop. 9, 10
- [ISO] ISO/IEC 9075-1:2016 - Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework). 4
- [KK89] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989. 8
- [KMCA06] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006. 18, 23, 59
- [KOF⁺16] Troy Kohwalter, Thiago Oliveira, Juliana Freire, Esteban Clua, and Leonardo Murta. Prov viewer: A graph-based visualization tool for interactive exploration of provenance data. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9672, pages 71–82, 2016. 10
- [LBM98] T Lee, S Bressan, and S Madnick. Source Attribution for Querying Against Semi-structured Documents. 1998. 6
- [LGS⁺14] Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. UpSet: Visualization of Intersecting Sets. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1983–1992, 12 2014. 12, 13
- [LS80] Bennett P. Lientz and E. Burton Swanson. Software Maintenance Management. *Guid and Control Software*, 1980. 2
- [Man] Manta. MANTA — Automate your data lineage. 9, 10
- [MELS95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages & Computing*, 6(2):183–210, 6 1995. 12, 26
- [MH⁺15] Dominik Moritz, Daniel Halperin, Bill Howe, and Jeffrey Heer. Perfopticon: Visual Query Analysis for Distributed Databases. *Computer Graphics Forum*, 34(3):71–80, 6 2015. 1, 10, 11

- [MRS⁺13] Wouter Meulemans, Nathalie Henry Riche, Bettina Speckmann, Basak Alper, and Tim Dwyer. KelpFusion: A Hybrid Set Visualization Technique. *IEEE Transactions on Visualization and Computer Graphics*, 19(11):1846–1858, 11 2013. 12
- [Mun14] Tamara Munzner. *Visualization Analysis and Design*. A K Peters/CRC Press, 12 2014. 36
- [NSH⁺18] Christina Niederer, Holger Stitz, Reem Hourieh, Florian Grassinger, Wolfgang Aigner, and Marc Streit. TACO: Visualizing Changes in Tables Over Time. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):677–686, 1 2018. 5, 10, 12
- [OAF⁺04] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 11 2004. 10
- [PEF16] Francesco Paduano, Ronak Etemadpour, and Angus G. Forbes. BranchingSets. In *Proceedings of the 9th International Symposium on Visual Information Communication and Interaction - VINCI '16*, VINCI '16, pages 9–16, New York, NY, USA, 2016. ACM. 12
- [PNK11] Sergey Pupyrev, Lev Nachmanson, and Michael Kaufmann. Improving layered graph layouts with edge bundling. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6502 LNCS, pages 329–340. Springer, Berlin, Heidelberg, 2011. 8, 26, 31
- [RD10] Nathalie Henry Riche and Tim Dwyer. Untangling Euler diagrams. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1090–1099, 11 2010. 12
- [RHF05] Patrick Riehmann, Manfred Hanfler, and Bernd Froehlich. Interactive sankey diagrams. In *Proceedings - IEEE Symposium on Information Visualization, INFO VIS*, pages 233–240. IEEE, 2005. 10, 11
- [Rot17] Robert E. Roth. Visual Variables. In *International Encyclopedia of Geography: People, the Earth, Environment and Technology*, pages 1–11. John Wiley & Sons, Ltd, Oxford, UK, 3 2017. 36
- [RS98] A. Rosenthal and E. Sciore. Propagating Integrity Information among Interrelated Databases. In *Integrity and Internal Control in Information Systems*, pages 5–18, Boston, MA, 1998. Springer US. 6
- [RSC15] Peter Rodgers, Gem Stapleton, and Peter Chapman. Visualizing Sets with Linear Diagrams. *ACM Transactions on Computer-Human Interaction*, 22(6):1–39, 9 2015. 12
- [Shn96] Ben Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. *Proceedings 1996 IEEE Symposium on Visual Languages*, 1996. 46
- [Shn03] Ben Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *The Craft of Information Visualization*. IEEE, 2003. 12
- [Sii14] Harri Siirtola. Bars, Pies, Doughnuts & Tables – Visualization of Proportions. pages 240–245, 2014. 45
- [SLSG16] H. Stitz, S. Luger, M. Streit, and N. Gehlenborg. AVOCADO: Visualization of Workflow-Derived Data Provenance for Reproducible Biomedical Research. *Computer Graphics Forum*, 35(3):481–490, 6 2016. 9, 10, 12
- [SM91] Kozo Sugiyama and Kazuo Misue. Visualization of structural information: automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):876–892, 1991. 8
- [SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974. 3

- [STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiro Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981. 8, 26, 59
- [SW] Lorenzen Bill Schroeder Will, Martin Ken. VTK - The Visualization Toolkit. 11
- [The17] The Apache Software Foundation. Apache Taverna: incubating, 2017. 1, 10
- [VBW15] Corinna Vehlow, Fabian Beck, and Daniel Weiskopf. The state of the art in visualizing group structures in graphs. In *Eurovis: Eurographics/IEEE Symposium on Visualization*, 2015. 12
- [VC07] Stijn Vansummeren and James Cheney. Recording Provenance for SQL Queries and Updates. *IEEE Data Engineering Bulletin*, 30(4):29–37, 2007. 9
- [VLFR17] Tatiana Von Landesberger, Dieter W Fellner, and Roy A Ruddle. Visualization System Requirements for Data Processing Pipeline Design and Optimization. *IEEE Transactions on Visualization and Computer Graphics*, 23(8):2028–2041, 2017. 1
- [WFS93] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 39(5):793–812, 11 1993. 18, 23, 59
- [WS06] Kenny Wong and Dabo Sun. On evaluating the layout of UML diagrams for program comprehension. In *Software Quality Journal*, volume 14, pages 233–259, 2006. 35
- [WSW⁺18] Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mane, Doug Fritz, Dilip Krishnan, Fernanda B. Viegas, and Martin Wattenberg. Visualizing Data-flow Graphs of Deep Learning Models in TensorFlow. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):1–12, 1 2018. 10, 11, 12
- [YS17] Bowen Yu and Claudio T. Silva. VisFlow - Web-based Visualization Framework for Tabular Data with a Subset Flow Model. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):251–260, 1 2017. 10, 11

