FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Visualization of data data flows and business rules using graph database
Student:	Bc. Daniel Hampl
Supervisor:	Ing. Michal Valenta, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Create a proof of concept for processing, enrichment and visualization of data lineage metadata provided by the Manta system.
Manta tool provides raw metadata describing the data lineage on the maximum level of granularity. These dependencies expressed as edges of graph structure need to be aggregated to a higher level and also need to be further processed to remove duplicates and derive indirect dependencies. The goal is to find out whether graph databases can provide an advantage in fulfilling this task or its parts compared to older solutions built on relational database technologies.
Tasks:
- The student will select a graph database to use.
- Student will devise a method to process the data provided by the Manta system into a technical data lineage.
- The student will analyze the past attempts made with the use of a SQL database and propose parts of the process to improve with the use of a graph database.
The main result of the thesis will be a proof of concept of this approach.

Electronically approved by Ing. Michal Valenta, Ph.D. on 5 November 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Visualization of data flows and business rules using graph database

Bc. Daniel Hampl

Department of Software Engineering

Supervisor: Ing. Michal Valenta, Ph.D.

April 22, 2022

Acknowledgements

I would like to thank my family and friends for supporting me through my studies. I would also like to thank my supervisor Michal Valenta for taking on this thesis. I would also like to thank the companies which provided me with the access and opportunity to research this topic, Profinit and Česká Spořitelna. Most of all, I would like to thank Petr Hájek, who has arranged this cooperation and who has provided me with valuable information sources and helped me shape this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on April 22, 2022 . . . . . .

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

Hampl, Daniel. *Visualization of data flows and business rules using graph database*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

V této práci zhodnotíme relační a grafové databáze pro použití při ukládání metadat a vizualizaci datové linie. Datovou linii zpracujeme pro použití v různých scénářích a vylepšíme ji pro efektivní využití ve vizualizaci. Navrhneme proces vizualizace různých druhů datových linií, jak do statických objektů, tak i v dynamickém prostředí, kde uživatel může procházet linií na v reálném čase.

Klíčová slova Datová linie, Neo4j, vizualizace, graf

Abstract

In this thesis, we will evaluate relational and graph databases for use in metadata storage and data lineage visualisation. We will process the data lineage for use in various scenarios and enhance it for efficient use in visualisation. We will devise a process to visualise various kinds of data lineage into static objects, as well as in a dynamic environment, where the user can go through the lineage on-demand in a realtime.

Keywords Data lineage, Neo4j, visualisation, graph

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we will be focusing mainly on the analysis and processing of metadata provided by the manta system. In order to get a broad enough dataset and work out any problems on real data, which will include most of the problematic parts we could encounter in the future. We have cooperated with one of the leading banks in the Czech republic, Česká spořitelna. The bank has provided access to its Data warehouse (DWH) metadata and Česká spořitelna metadata sandbox, which gave us a sufficient data sample for our analysis.

The data sources are then analysed with the use of the Manta tool, which is a data-gathering tool that scans various systems and their interactions and dataflows throughout and across these systems and their domains.[2]

Once scanned, we can export the data into CSV files containing a graph structure of nodes and edges as well as additional attributes gathered during the scan. This exported dataset, however, contains only the most basic connections on the lowest level, such as columns and variables in a script. The structure also contains information about parent nodes, which gives us a topological graph with the vertical axis being information about parent objects such as tables and their columns. On the horizontal axis, we then get the edges signifying the data flow.

This basic structure can be used to visualise the data lineage, but it is hard to orient in, and it also requires considerable expertise to understand the data. On top of that, once we decide to query data lineage for some of the larger and more complicated datasets, we can also meet with a performance problem

where the query would take too much time to complete.

To make the visualisations comprehensible for users without extensive technical knowledge, we need to process the data into some form that is easier to read and understand. We will need to aggregate the edges in the graph to a higher level, such as tables, columns and queries. This will allow us to visualise the data lineage in a more intuitive way. We can also further aggregate the edges to higher levels, as some use-cases only demand the bigger picture information where we are not interested in exact tables contained in each database. This type of aggregation will be further described as vertical aggregation.

The basic data lineage also may also contain many detailed steps between two column nodes, mapping the exact path of information from the source database column to the target database column. In most cases, we will not need the exact path of data flow through methods and functions. Instead, we would be content with the information that the data of one column are flowing into another column without the previous details. Thus, we will need to aggregate the data on the horizontal axis as well. This will be further described as horizontal aggregation.

Once we aggregate the graph to the required dimensions, we would like to examine the options of visualising the aggregated information and the basic data lineage as exported from Manta. We would like to explore the options for interactive visualisation, where the user can traverse the data lineage as they need, as well as the options for having static documentation for specific use-cases with parts of the data lineage that concern these use-cases.

As part of this proof of concept, we would like to determine whether it is possible to process the data and visualise the desired data lineage within a reasonable time frame using a graph database. We would also like to analyse the past attempts made based on a relational database, which was deemed unfit for the structured graph data.

We would also like to focus on the aspect of efficiency and performance, as the past failed mainly on the basis of performance, where it would take too long to process the dataset with desired results.

Chapter 2

Data lineage

As described in an article by R. Ikeda and J. Widom [3], Data lineage, also known as provenance, describes the lifecycle of data, where it came from, and various transformations it went through, and where it is going. Data lineage can be used for various purposes. We can use it for verifying data validity by tracking their sources. We can also use the lineage for confidence computation in probabilistic databases. It can also be used for understanding system behaviour and the evolution of data.

As much as the data lineage is valuable, it is also expensive to store and query. There are many approaches to solving these problems, and each has its advantages and disadvantages. For example, the approximate lineage, which lowers the storage requirements, can be used to determine the validity of data, but it is not possible to determine the exact lineage.

Data lineage can have two different levels of granularity, schema and instance. The schema level is a coarse-grained level of granularity, where the data lineage is described by the schema of the data. The instance level is a fine-grained level of granularity, where the data lineage is described in a more detailed way with concrete variables and database columns.

We can also separate data lineage into two types, where and how. The where lineage gives us the information, where the data came from. Thus, it contains only tables and direct data flow between them without any information about the transformations and other processes which helped the data reach the destination. On the other hand, when we look at the how lineage, we can see all the information about various transformations and other processes which

interacted with the data.

Each one of these types and granularities has its own uses. In this thesis, we will be starting with the fine-grained how granularity. It is in a form of a graph. The nodes of the graph act as the carrier of information. It represents the place where the information is stored at a given time, such as variables and columns. However, the nodes of the graph contain not only the most granular information but also the higher-level objects such as tables, databases, scripts and files. As for the edges in the graph, they represent the data flow between the nodes, such as the assignment of values from one column to another.

2.1 Metadata structure

As we receive the data, it contains a base graph structure with nodes and edges. We obtain this dataset as multiple CSV tables exported from Manta, namely Nodes, Node attributes, Edges, Resources and Layers.

The first table, Nodes, contains the actual nodes in our graph structure. Each node in this table has a name that does not necessarily need to be unique. Every node also has a type, such as a Column, Table or Report. Each node also has a unique id, and all but the root nodes also have a parent id.

Then we have the node attributes table. It contains a node id that identifies the node the attribute belongs to. There is also a name of the attribute, which is unique for each node. Lastly, there is also the value of the attribute, which can contain anything from an SQL query to human-written commentary.

Next, we have the Edges. This table contains connections between nodes. It has two node ids, the origin and the target, forming a relationship between the two nodes. Each edge also has one of the three types. The first is "DIRECT", signifying a direct flow between nodes. Then we have "FILTER" and "MAPS_TO", representing data being filtered according to the source node and data being mapped to another node, respectively.

Each of the above items also contains a resource id which signifies the source of the item, such as a database, system or DWH. Information about the resource is included in the resource table with its name, description type and layer id.

Last we have the layer table, which marks the layer of the scanned resource.

In our case, it is either a physical or logical layer. It contains the layer name and type as well as the id of import.

Thanks to the Manta tool, the graph contained in this export already has the basic data lineage mapped. The problem is that the actual data flow connections are mapped at the most detailed layer of columns and query variables. This level is hard to keep track of and difficult to visualize, as the data flow path would be too long, and it would spread across too many levels for a human to be capable of processing it.

Chapter 3

Database type overview

As time goes on and technological advancements progress, we no longer have only one database type as we were used. Nowadays, in addition to the usual Structured Query Language (SQL) or otherwise known as relational databases, present in most information systems developed up until now, we also have the Not only SQL (NoSQL) database category. These NoSQL databases have their own query language and, in some cases, also support the SQL. However, this is not the only difference. The NoSQL databases also differ in the way they store, query and overall work with data. Some databases approach data as objects, some as documents and some as a graph. There is a plethora of different approaches and various databases, but as for this chapter, we would like to limit our selection to the most common SQL, document and graph databases.

3.1 SQL databases

The traditional relational SQL database is based on the relational model. The database is based on tables and relations between these tables. These tables consist of rows and columns. Relational databases employ the use of keys. The most important key is the primary key which all rows must have and is unique in the entire table, as it identifies each record in the table. There are also the four basic operations known as Create, Read, Update and Delete (CRUD), which use the SQL. One of the most significant parts of a relational database is its Atomicity, Consistency, Isolation and Durability (ACID) properties. The SQL databases are heavily optimised for storing and retrieving data quickly and efficiently, employing indexing or query optimisation. It also allows us to

store data within predefined structures and constraints. [4]

3.1.1 Structure

The main element of each SQL database is a table. It has a name and a strictly defined structure for each record in it, where the record equates to a row in the table, and the structure is defined by columns. Every column has a strictly defined name, data type and size. As mentioned above, each record has its primary key, which uniquely identifies it and can be used to access the record directly. Some records also can have a foreign key. The foreign key matches the primary key, and together, they form a relationship between the two records.

Another one of the database objects is View. View offers data in the same structure as a table, but it is only a middle man, and it does not directly store data. Instead, it has a predefined script to query other views or tables and transform their data into a predefined structure. Some databases also support a materialised view, which stores data once materialised and can be used as a cache for more complex queries and processes.

Another essential part of a SQL database is an index. It helps with optimising database processes and queries or defining a unique value. Indexes are also commonly known as keys, such as the primary or foreign keys. There are multiple ways to store indexes, each having its own advantages and disadvantages as well as uses.

Constraints introduce a limit on values within each table column. For example, while saving a battery charge percentage, we could limit its value to be between zero and one hundred. Constraints can also be used for creating a multi-column primary key.

Then we have the trigger, which defines a query that should be run once an event happens, such as data being inserted into a table. Each trigger also has the option to be activated before or after an event. This way, we can validate the data before an insert or update our materialised View right after its data have been modified.

Last but not least, we have Transaction. It is a unit of work that has the ACID properties and signifies a transfer from and to a valid state.

3.1.2 The ACID properties [1]

The ACID properties, as described further, were coined in 1983 by Andreas Reuter and Theo Härder in their article ”Principles of transaction-oriented database recovery”.

When we speak about the ACID properties, we mean Atomicity, Consistency, Isolation and Durability. These properties are the baseline for every SQL database and all of its transactions, which are the operations modifying the database, namely the CRUD operations.

The atomicity has the same meaning while talking about databases as when speaking about atomicity anywhere else in informatics or any other field in the world. When we say a transaction is atomic, it means it is indivisible, it either completes as a whole or it will make no changes to our database. Thus, there can never be an undefined state.

Next of the ACID properties is Consistency. Consistency means that a transaction cannot disrupt a database structure and its integrity. Thus, if a transaction would lead to an inconsistent database, it will instead fail and leave the database in the original consistent state.

Isolation signifies the requirement for each transaction to be completely isolated and not influenced by any other transaction. If we were to read a record while it is being modified, we would still get the original value until the update operation is successfully completed.

The last property is Durability. Once a durable transaction is successfully completed, all the changes will be saved to the database. Furthermore, all these changes are permanent until modified by other transactions later, and these changes will not be affected by, for example, a system crash.

3.1.3 Database normalization

In the SQL database, we have something called normal forms. The normal forms are numbered by their level of normalisation. The main target of database normalisation is to reduce redundancies in the database, thus reducing the possibility of conflict occurring and lowering the size of a database. Each of these forms also requires the requirements of the preceding form to be satisfied.

The zeroth normalised form is achieved by all tables, as the only requirement is to contain at least one column.

The first normalised form requires all columns to contain atomic values. This means we cannot have a list of phone numbers in one column. Instead, we must create a table containing one phone number in a column and a foreign key to our original record.

The second normal form requires that all columns are dependent on the entire primary key. This means that when we have a primary key consisting of two or more columns, we can not identify the value in any of the columns by only part of our primary key.

The third normal form requires tables not to contain any transitive dependency. For example, let us have a table for a car and a table for the brand of the car. If we were to store the brand id in the table containing cars as well as the name of the brand, we would have created a transitive dependency. It would be more efficient to move the brand name into the table of brands. It would also be much more efficient to change the name of the brand if it was renamed.

There are further levels of database normalisation. If the reader would like to find out more information about this subject, we would like to recommend the "Database in Depth: Relational Theory for Practitioners", on which we based this section. [5]

3.1.4 Graph structure

When we approach the graph structure in general, we have to somehow store it in the relational database in the form of relations and tables. As we can have a variable number of node types, it would be hard to separate each node type into a separate table and normalise the data. We would have to dynamically create tables on-demand as the data changes.

Even if we worked with a static unchanging dataset, we would have to store the relations. This is another significant problem, as, within the graf structure, we can have an infinite number of relations since two nodes can have multiple relations between each other, and these relations can have various types. Thus, it makes it close to impossible to store the relationships between data nodes as database relations between tables.

Thus, it is much easier to just place the nodes into a single table containing all nodes, add a second table with the node attributes, and finally add a table of edges, leading to the normalised structure as it is described in the Section 2.1.

As mentioned in the Section 3.1.1, indexes in a database can increase the efficiency of record lookups. For example, when we have an index in the form of a binary tree, we can shorten the lookup for each record from a linear $O(n)$ to a logarithmic complexity $O(log(n))$. This indexation allows us to administer sizable datasets with relative efficiency. [6]

On the other hand, when we have data in a graph structure as described in the Section 2.1, it can get quite arduous to traverse the graph data, as the nodes are not directly linked together. Furthermore, when the dataset rises in volume, we are forced to do a high number of joins, which in turn considerably slows down our transaction, even with a proper indexation. The problem originates from the way joins are processed in the SQL database, as the query lookup process is performed for each join separately. Thus, it leads to a time and resource-consuming process, as described within the following article comparing SQL and NoSQL databases.[7]

Therefore, it makes any attempts to normalise data into the table structure as mentioned above a naive approach, which is not efficient enough for larger datasets. [6]

If we would like to efficiently process data within an SQL database, we would first have to extend the database itself to optimise the approach to the graph structure on a lower level. This approach was attempted in a thesis written by J. Hovad in 2011 on the PostgreSQL database.[6]

Another problem that arises with the use of SQL database when we traverse a graph structure is cycles. Our data can either have paths leading into the element itself or cycles containing multiple nodes. Some of the SQL database traversal options have safeguards implemented for the first issue. However, the second type of cycle is usually not implicitly resolved and needs to be taken care of as part of our algorithm. It is also more challenging to find libraries containing graph-related algorithms for SQL databases than when we try to search for them in graph databases, where they are usually part of the database itself. This, in turn, leads to arduous development when working

with graphically structured data on a relational database where we have to emulate the structure of a graph.

3.2 NoSQL databases

> Though it is desirable to have consistency, availability, and partition tolerance in a distributed system, unfortunately no system can achieve all three at the same time.[8]

Figure 3.1: The CAP Theorem

NoSQL databases became more commonly used with the spread of big data with the main purpose of storing data in distributed database systems. [9] As stated by the CAP theorem in the Figure 3.1, it is impossible to achieve Consistency, Availability and Partition tolerance at once, limiting a distributed database to having only two of these properties at once. This led to the adoption of the Basically Available, Soft-state and Eventual consistency (BASE) model adopted by most of the NoSQL databases. This model does not provide strong consistency. Instead, it is based on eventual synchronisation, where, given enough time, the data becomes consistent. [8]

Furthermore, with a different approach to the database, the view on the database normalisation has also changed. For example, in the document database MongoDB, the usual normalisation we could see in the SQL databases is not applied. Instead, we store all related data within one document. Using this approach, we can store all the data within a single entity, which allows us to query the necessary data without any costly joins or aggregation. This way, we also get all the essential information at once without the risk of the related entities not being synchronised in time. [10]

3.3 Graph databases

Graph databases are part of the NoSQL database category. They allow us to store a graph structure with completely different nodes with close to no predefined structure. Each node can have different parameters and can be connected to any number of other nodes. In many graph databases, the edges can also have various attributes.

3.3.1 Native graph database

[11] Not all graph databases approach the data in the same way, and there can be a considerable difference in their processing speeds. Two properties of the graph database cause this difference. First is the way the database stores the graph structure. It can be stored natively as graph storage. This way, the data are stored optimally, ensuring the nodes and relationships are kept close to each other. Or the graph structure can be stored in a non-native way. For example, in a relational model, object-oriented model, or any other general-purpose storage.

The second property is the processing engine. Based on how the engine works, we can either have a native graph processing engine, which employs the use of index-free adjacency, where connected nodes point to each other directly.

To be able to say a database is a native graph database, we need to satisfy two conditions. The database has to have native graph storage and native graph processing. On the other side, we have an emulated graph processing, where the engine only behaves as a graph from the outside, and on the inside, processes the queries in various other ways.

3.3.2 Traversal problem

Usually, when we traverse a graph, we have to take into consideration various variables. For example, when we wish to determine if two nodes have a path between each other, we might want to use Breadth first search (BFS) or Depth first search (DFS) algorithms. Both of these algorithms have their own advantages and disadvantages. They have pretty much the same performance when we want to find all the existing paths between these two nodes. However, when we only need to determine if there is at least one path, it matters where we look first, and the difference in performance can be vast, depending on the density of the graph and usual path length.

Unlike relational databases, the graph database is meant for graph-structured data, especially a native graph database. Thus, when we use the graph database to traverse a graph structure, we do not need to extend our database with additional procedures and views just to be able to work with the graph. We also generally get a performance improvement, as a graph database has at least some optimisations compared to just using a relational database. The increase is most significant when using a fully native graph database.

Furthermore, as the database is made to work with graph-structured data, it comes with a query language intended for graph structure as well. This query language allows pattern matching based on the graph structure. As the query language was developed with the graph structure in mind, it also accounts for cycles during traversals. This way, we do not have to worry about self-referencing cycles as well as other cycles in the graph, and the database takes care of everything for us.

When we further analyse the graph traversal problem, we come to the conclusion that the index-free adjacency provided by the native graph database is not a sufficient solution for large database systems. Although the index-free adjacency significantly improves the speed of traversals in comparison to the usual relational database system, for long path traversals, it is still insufficient. When we look at the dataset provided, it commonly contains data lineage paths with tens of steps. Once we take into consideration that each node can have hundreds or thousands of relations, we get an exponentially growing number of paths we need to visit to build the entire data lineage path.

According to K. Kusu and K. Hatano [12] who have examined the traversal problem with recurrent paths on large graph datasets. In this article, they have come up with an approach to increase the efficiency of the traversal on graph databases. With some modifications, this approach could also be applicable to relational databases. The basis of the approach is storing the information about nodes reachable over multiple steps of the recurrent path as an attribute of a node.

Chapter 4

Requirements analysis

In this chapter, we would like to focus on what has already been achieved with the use of OracleDB, what can be improved, and which graph database would be a good fit for our needs.

4.1 Requirements

The main goal is to visualise extensive data lineage within a reasonable timeframe. We need to preprocess the data in various ways to achieve this goal. Some visualisations require only the larger picture information about higher-level nodes, such as data flow between databases and systems. It would be quite time consuming to calculate the paths between these high-level nodes for every visualisation. Furthermore, we also need to find an approach to visualise the detailed data lineage for extensive analysis. To achieve this goal, we need to preprocess the data in various ways to shorten the time necessary for visualising each case.

After consultation with the DWH management and data analytics from the data provider, "Česká spořitelna", we have gathered three main targets. We need to be able to visualise the data flow on the top level of databases and schemas for general purposes and basic decision making. Next, we need to be able to visualise the data flow on the level of tables for the purposes of analysing existing as well as new in-development systems. Lastly, we have the need to visualise the bottom-most level for the detailed analysis of dataflows.

In most cases, we will not need the information about data transformations.

Thus, we can skip the transformations and link the tables connected directly to save processing time. However, in some cases, we will need to focus on the detailed path, including all the transformations. Due to this need, we cannot entirely remove all transformations, and we still need to be able to display them.

Moreover, many times, while examining the data lineage on the table level, we will not need the full path. For example, the information that the data originates from a certain database or schema will be sufficient for the data analyst to stop further examinations. This way, we could save some processing time and make the result data lineage visualisation easier to read.

4.1.1 Use cases

To paint a better picture of the desired results, we would like to describe a few general use cases for which we would like to use the visualisations of the data lineage.

The first use case, which requires the least work, is to examine the exact data flow between two tables in a database. This information can be used for detailed analysis of the data flow once we have the general knowledge about the data and we need to know further detailed information. This is the case for the data analyst who needs to know the exact data flow between two tables in a database.

Another use case is to examine a data flow to a table. This can be used for determining the quality of data. For example, we can track the sources of changes in a table containing critical data and determine the weak links in the data lineage.

A similar use case to the last one is tracking outgoing data flow. In this case, we need to track the uses of information. For example, when we have personal information about clients, we need to make sure it will not be used in any publicly accessible parts of our system. We can then ensure the proper security measures are in place.

The last use case is to understand the system while making modifications or transferring it to newer technology. In this case, the data lineage can act as documentation for the system. We can then use it to determine the changes that need to be made to the system.

These are only a few from the multitude of examples for which we can use the data lineage visualisations.

4.2 Current solution

At the start of this all, we have the manta tool and its export with nodes, edges, their attributes and resources. Thus, the first thing that needs to be done is to import the dataset into the database. Once the graph is imported, we also clean it up from unnecessary nodes. This is done by removing the bare minimum of nodes that have no added values, such as nodes with an unknown resource type. After that, the nodes and edges are enriched with more information precalculated from the graph to allow faster traversal.

4.2.1 Building the base entitites

As they are exported from Manta, the base entities are inserted into tables within the database matching the structure of the exported files. As the OracleDB is a relational database, the base tables also have relations according to the imported structure, with nodes having constraints to their parent node as well as edges being connected to the source and target nodes of the relation they signify. Both nodes, as well as edges, also have constraints for the resource marking the source system of the scanned object.

Once the base entities are imported as provided, the entities are then rebuilt and enriched with further information. The core of the optimisation is to precalculate the path from the root node to each node. There is not only one path but three. One path consists of node types, one has the names, and the last has node identifiers. Next, each node has a level, which signifies the depth (number of parent nodes preceding it). Each node also has information about its root node, as well as information about the so-called master level node, which will be described further in this chapter. Finally, there is also the count of target and source nodes, which helps stop the traversal earlier and thus saves a little bit of the precious time needed to traverse the graph.

Thanks to these optimisations, we can directly access any of the parents of a node at any given time. Thus, saving a lot of time, especially when would have to find a parent over multiple steps, which would require numerous joins, which is quite expensive.

4.2.2 Master level

The master level is a level determined by an analyst who has in-depth knowledge about the scanned systems. This level should be understandable to the everyday manager without the requirement of in-depth knowledge of the system or technical knowledge. To this level, we are trying to aggregate the connections which are currently at the bottom-most level in the export we receive and are next to impossible to follow for any human being. Once aggregated, the data lineage can help determine the provenance of data in various systems and help determine the type of data that can appear in said systems, as well as the impact any changes can have on data downstream.

As of now, the data lineage data gathering is done through complex scripts with hard-coded rules of node types that are to be skipped or displayed. These rules need to be set case by case, and there is no guarantee they won't need to change once the tracked data change their structure. Furthermore, as these rules are created case by case for each data lineage diagram, there is considerable time required to create each diagram.

4.3 Selection of graph database

During the search for the suitable database, we have, among other things, consulted "An empirical comparison of graph databases"[13], where the Node4j database particularly piqued our interest. Especially after the considerable improvement Neo4j has made since the publication of the comparison in 2013.

Since then, Neo4j has introduced features such as APOC a heavily optimised library of procedures and functions or the Bolt binary protocol with native drivers for multiple programming languages.[14] The APOC library is also open source, which allows it to grow and become more optimised over time while its userbase grows.

Furthermore, Neo4j also has a powerful Graph Data Science (GDS) library, which introduces many features useful for data analysis, such as pathfinding procedures which are heavily optimised for massive scale, as well as parallelisation. [15]

We have also considered other graph databases, such as Amazon Neptune, Redis or Sparksee (formerly known as DEX). However, Neo4j has prevailed over other candidates, especially thanks to its large userbase and well-developed

documentation. Some of the other databases also introduced limits on free licences to one million nodes or below, which is way below our needs of tens to hundreds of millions of nodes, which also factored in our evaluation.

We have also disqualified all of the in-memory and non-native graph databases. We need to focus on the processing efficiency, as the datasets considered are rather sizable, and our resources are quite limited for the purposes of this proof of concept.

Chapter 5

Metadata storage

The first thing we need to do is process the data exported from Manta, as it is described in the Section 2.1. We need to import the data into our database and aggregate the edges into a higher layer.

5.1 Import

The first thing we need to do is get the source dataset containing the graph we need to further process. We can either export this graph from Manta directly or use the already exported version within the relational database used up until now. This way, we can use any enhancement developed in the relational database. Thanks to this option, we are also not limited to developing our aggregation scripts and procedures within the graph database, and we can use other resources as well. The export from the relational database is highly similar to the Manta tool. Thanks to this similarity, we can import the data with minimal changes to the importing script itself.

Once we have exported the CSV files, we can proceed to import them into our Neo4j database. At first, we have used the built-in function ”load csv”, as can be seen in the Figure 5.1. However, that only worked on the smaller datasets we used for testing purposes. Once we tried to import the working dataset, the import became exceedingly slow.

Thus after some research, we have decided to use the APOC library. Namely, the ”apoc.periodic.iterate” and ”apoc.load.csv” functions, as can be seen in the Figure 5.2. Thanks to this improvement, we have been able to shorten the time it took to import our files from hours to minutes.

Once imported, the dataset for this proof of concept contains over ten million nodes and over twenty million relations.

```
load csv from "file:\\\edge.csv" as line
create (:EdgeL0 {edge_id:line[0], from:line[1], to:line[2],
                 type:line[3], no:line[4]});
```

Figure 5.1: Native Cypher import

```
CALL apoc.periodic.iterate(
    'CALL apoc.load.csv("edge.csv") yield map as line return line',
    'create (:Edge:L0 {edge_id:line.EDGE_ID, from:line.SOURCE_NODE_ID,
                       to:line.TARGET_NODE_ID, type:line.EDGE_TYPE,
                       resource_id:line.RESOURCE_ID});',
    {batchSize: 10000, iterateList: true, parallel: true});
```

Figure 5.2: APOC import

5.2 Building the graph

With all the data imported into our database, we have set to build the graph structure. At first, we have attempted a more comfortable solution, using Python scripts and the bolt connector. This approach, however, was not as fast as we wished it to be. The native bolt driver improved the connectivity considerably with data streaming and other optimisations, but it was still easier to do all the operations directly within the database.

5.2.1 Nodes

To better distinguish our data nodes, we chose to use the node type as a label. Thus, being able to quickly filter them and display them with separate colours, thus easily distinguishing the types of data sources while displaying our data paths.

We have found it to be quite an arduous process to dynamically set the label of a node within a native Cypher (the Neo4j query language). The only possible answer we have found was to split the nodes into groups by type. Afterwards, create a query string while concatenating the type with the rest of the query string and then evaluating the query string.

Thus, we have tried to process the data with python scripts, but that was infeasible due to the inefficiency, as mentioned above. After introducing the APOC library, we were able to make the process more efficient. However, we still ended up having to split up all the nodes into groups separated by type first. Nevertheless, the ”apoc.periodic.iterate” function was also beneficial in this case.

The next part was adding the additional parameters to our nodes. This was the second reason we were keen on using Python scripts at first. We have met with an issue where we needed to set property by a dynamic name. This issue was easily solved within a Python script, but it was relatively time-consuming. Thus, we were forced to turn to the APOC library again. After some research, we have found the ”apoc.map.setKey” function, which allowed us to modify an object by a variable key. Thanks to this, we could swiftly update all newly created nodes with additional attributes.

5.2.2 Edges

Once the actual nodes were prepared, we have set to add the edges, for which we have used the same approach as with building the nodes themselves. At first, we have attempted to use a Python script, which allowed us to do all the necessary operations comfortably. However, this approach has proven to be too time-consuming. Thus, we have set to optimise the process with the APOC library.

At first, we have separated all the edges imported to groups by type in the same way as with the nodes. Afterwards, we have used the iterate function from APOC and created the edges between nodes. However, once we tried to run this short script, we have met with an issue. The script was never-ending. It took us considerable time debugging the script and further testing until we noticed the create operation was locking nodes, and our script ended with a deadlock. Once we realised the issue, we switched to synchronous iteration, and the script finished successfully within a few minutes. The final script can be seen in the Figure 5.4. We are aware of the possibility of optimising this script further and adding the edges at the time of import, thus saving the time needed to load the edges from our database.

```
match (n:Edge:L0) with collect(DISTINCT n.type) as types
unwind types as type
call apoc.periodic.iterate(
    'match (e:Edge:L0 {type:\'' + type + '\'}),
           (f:Node:L1), (t:Node:L1)
     where f.node_id=e.from and
           t.node_id=e.to return e,f,t',
    'merge (f)-[:' + type + ' {resource_id:e.resource_id}]->(t)',
    {batchSize:10000, parallel:false})
yield total
return total;
```

Figure 5.3: Add basic edges

The next step in our graph-building process was to add the edges to our parent nodes. This task was rather straightforward. All we had to do was match all nodes with their parent node by the node ids already present and create an edge between these two nodes.

5.2.3 Modifications

To make the orientation among the nodes easier, we have chosen to add a path string to each node containing the names of all its parents, creating a path from one of the root nodes to its imminent parent. This way, we can identify the current schema or project the node is part of without the need to load up all the parent nodes.

Initially, this task was resolved as part of creating the parent connections, as it was relatively effortless to go through all of the nodes through a depth-first search and, while adding the connection, add the path as well. However, after moving the edge building process from a python script, we were forced to split this process into two queries, as the depth-first search did not allow us to iterate through the nodes while adding edges.

In the end, we have managed to find the ”apoc.text.join” function, which in combination with the ”collect” function, allowed us to build the path directly in the Cypher script, as can be seen in the Figure 5.4.

```
call apoc.periodic.iterate(
    'match p = (n:Node)-[:PARENT *1..100]->(m)
        return n, apoc.text.join(
            apoc.coll.reverse(collect(m.name)),
            ".") as path',
    'set n.path = path',
    {batchSize:10000, parallel:false}
);
```

Figure 5.4: Add paths from root to each node

5.3 Filtering

Since the graph may contain unnecessary nodes, duplicates as well as nodes that were scanned with error, we need to filter out these nodes. For most of these nodes, we can directly delete them according to preset rules. We can set these rules to match custom name patterns, such as searching for nodes containing phrases like ”copy” or ”dummy”. We can also match all nodes which are scanned with a connected resource named ”Unknown”. This resource marks nodes with unknown origins. We can directly delete these nodes, as we will not need them for any further analysis, and they would only take up space and take up processing time during further aggregation.

We can also encounter uninteresting nodes, such as logs or other similar tables. We can not delete these nodes, as there might be cases where we need to display or examine them. However, we do not want to use them in our aggregations as they could create unnecessary paths. We can thus mark them with a custom label, in our case a simple ”Skip”, and then, later on, we just filter these nodes out during aggregations when these nodes are not needed. Since we will not delete these nodes, we can display them when needed directly or use them in selected aggregations where it makes sense.

5.4 Aggregation

Once we have built the graph within our database, we have set to aggregate the structure to be easier to read and understand for an average human being. To start with, we have set to aggregate connections at the bottom-most level, meaningless for a human mind. We have used the so-called master level nodes selected by the data analyst before, as mentioned in the Section 4.2.2.

```
unwind edgeTypes as edgeType
unwind nodeTypes as nodeType1
unwind nodeTypes as nodeType2
call apoc.periodic.iterate(
    "match (f:`" + nodeType1 + "`)
            <-[:PARENT *1..100]-
        ()-[:" + edgeType + "]->()
            -[:PARENT *1..100]->
        (t:`" + nodeType2 + "`)
    return f, t;",
    "merge (f)-[:" + edgeType + "]->(t)",
    {batchSize:10000, parallel:false}
)
yield total
return total;
```

Figure 5.5: Horizontal aggregation

5.4.1 Vertical aggregation

To aggregate relations to this level, we first tried the basic match within Neo4j. This approach, which was sufficient for testing purposes, turned out to be quite lacking once we had introduced it to the production dataset. For one, we needed more than one type of node to be connected as opposed to only a single node type used for testing. Thus, we needed to iterate through all the types of our master-level nodes. Therefore, we have once again introduced the 'apoc.periodic.iterate'.

We can see the final result in the Figure 5.5. We have first gathered the edge types and list of master node types, which we have omitted in this example as it would take too much space.[1] Once we had all the types, we used an unwind, which works in the same way as iteration over the list and went through all of the types, the edge types, as well as the node types of both starting and finishing nodes. Afterwards, we have used the basic match on our path structure and added the aggregated edges. The basic match selects two of the master-level nodes, then matches all child nodes under them and checks if any of these child nodes have a connection between each other in the right direction.

This way, we have successfully aggregated the relations into the master level.

[1] The full script can be found within the attached full_build.cypher file

Once connected, we have also separated these master-level nodes into four groups, Reports, Tables, Transformations and Files, according to what best describes the master-level nodes. This way, the manager does not need to know the meaning behind a "Tableau Data Source" and only needs to know that it stores data and acts the same way as a table for our data lineage purposes. This separation has been done by a simple label assignment over all of the master-level nodes.

5.4.2 Horizontal aggregation

Once we had built the connections between the master-level nodes, we were able to visualise the data flow in the built-in Neo4j Browser. However, this level of aggregation was still insufficient. In order to track the provenance of data, we had to go through multiple transformations just to arrive at a source table. Thus, after consultation with our data analyst, we have set to skip these transformations, aggregate the data horizontally and connect tables directly to each other without the need to go through many transformations.

Once we have started analysing the master-level nodes, we have found that most of the nodes had a connection to a logging database. This connection has led to problems with displaying the data lineage, as it forced us to load many unnecessary nodes. Thus we have removed all logging related nodes from our master level for the purposes of this proof of concept.

The first idea to horizontally aggregate our nodes was to use a similar approach as with vertical aggregation. We wanted to use a match statement over multiple relations and tables and then iterate over the results. This approach has proven not to be feasible, as the match statement matches all existing paths and stores them in memory.

Once the previous attempt failed, we turned for help to the APOC library, which has helped us many times in the past. We have found two procedures that could be potentially helpful.

The first APOC attempt was with the "apoc.path.spanningTree", as can be seen in the Figure 5.6. This procedure is optimised for finding one or a selected number of paths between one and a list of other nodes.[16] We have selected all our table nodes as starting as well as ending nodes for our path. We have added a condition for removing self-referencing relationships, as they are irrelevant for our purposes. Then we used the procedure with label fil-

```
call apoc.periodic.iterate(
    "match (n:MasterTable), (m:MasterTable)
        where n.node_id <> m.node_id
        return n, m;",
    "call apoc.path.spanningTree(
            n,
            {
                relationshipFilter:'DIRECT>',
                labelFilter: '+MasterTransformation|/MasterTable',
                endNodes:[m],
                limit: 1
            }) yield path
        merge (n)-[:MasterDirect]->(m)",
    {batchSize:10000, parallel:false}
)
yield total
return total;
```

Figure 5.6: The first APOC attempt for horizontal aggregation

```
call apoc.periodic.iterate(
    "match (n:MasterTable) return n;",
    "call apoc.path.expand(
            n,
            'DIRECT>',
            '+MasterTransformation|/MasterTable',
            1,
            10000
        ) yield path
    merge (n)-[:MasterDirect]->(last(path))",
    {batchSize:10000, parallel:false}
)
yield total
return total;
```

Figure 5.7: The second APOC attempt for horizontal aggregation

tering over transformation nodes, with the terminal node being a table. We have also limited the relationships to direct for the purposes of this proof of concept.

We have also pondered on options to optimise the query in the Figure 5.6. One option we could come up with was to add all the table nodes to our end-

node list. However, this option has proven to be unfeasible, as we could not limit the number of returned paths, which only made the query slower.

The second APOC procedure we have attempted to use was the ”apoc.path.expand” in the Figure 5.7. This procedure focuses on expanding the path according to provided filters. However, this procedure has proven to be an unfeasible option as well since the procedure does not allow for exempting the starting node from the traversal node filter. [17]

After further analysis of this subject, we have met with a recommendation to write a server-side extension for any complex high-performance operations.[18]

5.4.3 Neo4j plugin

Based on the abovementioned recommendation, we have set to explore the world of Neo4j custom plugins. The essential requirement for its creation was java, as the Neo4j is implemented in java. The next thing we needed was to set up the environment with maven and required packages. This process is well described within the neo4j documentation[19]. Thus we will not be going into detail, as it is slightly out of the scope of this thesis.

Once we had the core of the java plugin, we have added a procedure accepting five arguments, labels of the starting, traversal and ending nodes to filter by, as well as the type of relationship we would traverse over and lastly, the relationship type of the newly created aggregate relationships instead of the existing paths.

To make our procedure as efficient as possible, we have made it fully in memory and compact. Thus, at the start of our procedure, we gather all the starting, ending and traversal node identifiers. Afterwards, we load all the relationships between the gathered nodes. From these relationships, we only select the identifiers of starting and ending nodes. We then group these relationships by starting node identifier. We save it to a HashMap[20] structure, where we map the starting node identifier to the list of all related ending node identifiers to improve lookup performance. This way, we have easily accessible relations for all our nodes. Based on the Java documentation[20], we should achieve the optimal constant-time retrieval performance, as the hashmap was constructed with a known initial size, which never changes throughout the procedure.

Next, we create another HashMap for saving the list of accessible end node

```
dfs(currentId) {
    if (ifAlreadyVisited(currentId)) {
        return
    }

    setVisited(currentId)

    connectedNodes = getRelationships(currentId);
    for (connectedNodeId in connectedNodes) {
        dfs(connectedNodeId);
        addResultsFromConnected(currentId, connectedNodeId);
    }
}
```

Figure 5.8: The DFS pseudocode for finding all paths

identifiers for each of the nodes, including the traversal nodes. Afterwards, we run a DFS algorithm[21] from each of the starting nodes gathered at the beginning. This way, we only have to visit each relation at most once. The simplified pseudocode version of our DFS algorithm can be seen in the Figure 5.8, and the full implementation can be found in the attached CD within the "src/plugin" folder.

Once we have all the paths aggregated into simple relations, all we have to do is to create all the new relations in our database. Since we are working with larger datasets, we have to separate the process into multiple transactions. We create a new transaction for each of the starting nodes and commit it once we add all the relations starting from the said node.

After we have completed the code, we have packaged it into a Java runtime environment (JRE) plugin, which can be plugged into any Neo4j server and used for aggregating large dense graphs. However, this plugin is made only for the purposes of this proof of concept and for further, more general use, it would need further improvements, such as allowing multiple label types for relationships as well as nodes, as well as other improvements and possible performance optimisations.

However, after further investigation of the graph structure, we have found that there can be cycles within the data flow. Due to these cycles, we were forced to reexamine our DFS algorithm, as the original algorithm would not connect all the nodes properly. For example, with the graph structure displayed in the

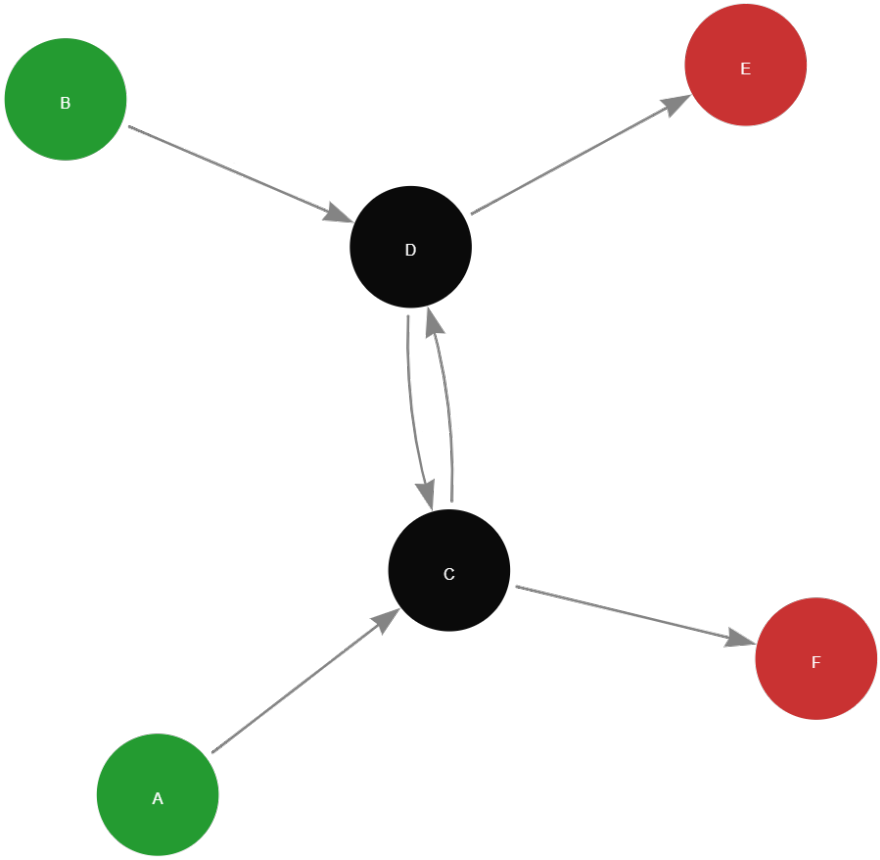

Figure 5.9: The graph structure with cycles

Figure 5.9, where we have starting nodes A and B, traversal nodes C and D and finally terminal nodes E and F. The original algorithm would start in node A and follow through the node C, D and E. Once reaching a terminal node, it would go back to node D and mark it as it has access to node E. Afterwards, it would return to node C and mark it with access to node E. Then, it would follow to node F, and mark node C with access to node F. Once node C is completed, it would add the information about access to nodes E and F to node A. After all edges for starting node A were calculated, the algorithm would start from node B, following to node D, which would appear as calculated and return only access to node E, leaving node F unconnected.

Due to this problem, we have tried to use a simple DFS algorithm without storing the accessible terminal nodes in each of the traversal nodes. This way, we would run a simple DFS algorithm from each of the starting nodes.

Thanks to the in-memory processing, the difference in processing time was not significant. Thus, we have decided to use the simple DFS algorithm without any further attempts to make it more efficient, as it was not necessary for the purposes of this proof of concept.

Thanks to this plugin, we have been able to create over twenty million aggregated relationships within approximately 2 minutes. To compare the progress with the first APOC implementation in the Figure 5.6, which we stopped after over twelve hours of processing which created a little over one hundred and forty thousand relationships. At this rate, it would take over two months to process all of the horizontal aggregations using the APOC library.

5.4.4 Reachability

In order to be able to gather the data lineage for more complex cases, we need to precalculate the reachability of certain nodes to limit the amount of data we have to process. The preprocessing is done by another custom procedure not dissimilar to the first procedure in our custom extension. This procedure loads up all the nodes matching a selected label and their relations of a selected type. It then traverses the graph from each of the nodes and saves the identifiers of all the nodes it has visited. After all the accessible nodes for a starting node have been gathered, we create a relationship between the starting node and each visited node. We have been forced to create the relationships for each starting node separately, as we are limited by the ram size, and the graph is too extensive for us to store all the newly created relationships within RAM.

Once we have all of the accessible nodes for all of the starting nodes, we can easily examine any relationships within the data path on higher levels and gather the lower levels with only the necessary data which relates to our target node. To be able to gather details on the bottom-most level as well as on the master level, we have preprocessed both these levels in the same way. However, the preprocessing was done only on the data nodes, as most often, we do not need the procedures in between. As for the edge cases, where we need the full path with procedures included, we can gather the paths in between our data nodes to complete the detailed path. This process should not be too time-consuming, as the transformation nodes do not have as many relationships as the data nodes.

5.4.5 Path limiting

To limit the number of nodes within the path, we have also added further aggregated relations. These relations connect the master level data node and the top-level data node on the border between top-level nodes. This way, we can traverse the path and stop at the border once we leave the schema or database. We can also specify how many borders we are willing to cross while gathering all the nodes in our data lineage path. This limit was introduced because some tables have data gathered throughout a large part of the system, and the path branches rapidly, which can be hard to visualise. It is much easier to visualise and comprehend when we split the path into parts.

We can also use the combination of path limiting, traverse the path mostly over top-level nodes and then load up selected parts of the path with the help of preprocessed reachability relations. This way, we can load up only the nodes that relate to the data node we are currently examining without the need to traverse the full path at the examined level. This approach can make data lineage analysis much more efficient, as we do not need to load anything else than we are currently examining.

Chapter 6

Visualisation

We have found various options for visualising our data lineage. Each option has its pros and cons, and we would like to describe them in this chapter.

6.1 Neo4j Browser

The Neo4j Browser application has a built-in basic visualisation tool. It allows the user to browse through the data nodes. It is ideal for a developer, as it can be used for writing detailed queries. However, it requires basic knowledge of the Cypher language for its use. Thus, it would not be suitable for managers without prior knowledge about graph databases. On the other hand, it would be ideal for use-cases such as developers trying to find all tables which can be affected by changes in a selected table. Once all aggregations are finished, a simple query could achieve this task.

The Neo4j Browser also allows users to administer a local version of our graph database. This way, any user can wither run the aggregation scripts on their device as well as load a backup of the already ready database and work in a fully offline mode.

The basic built-in visualisation tool also limits the number of nodes it will visualise. Furthermore, it also loads up all the existing relationships of loaded nodes, which can be chaotic, especially when introducing all of our aggregated relations. Thus, it is not the ideal solution for most of our use cases.

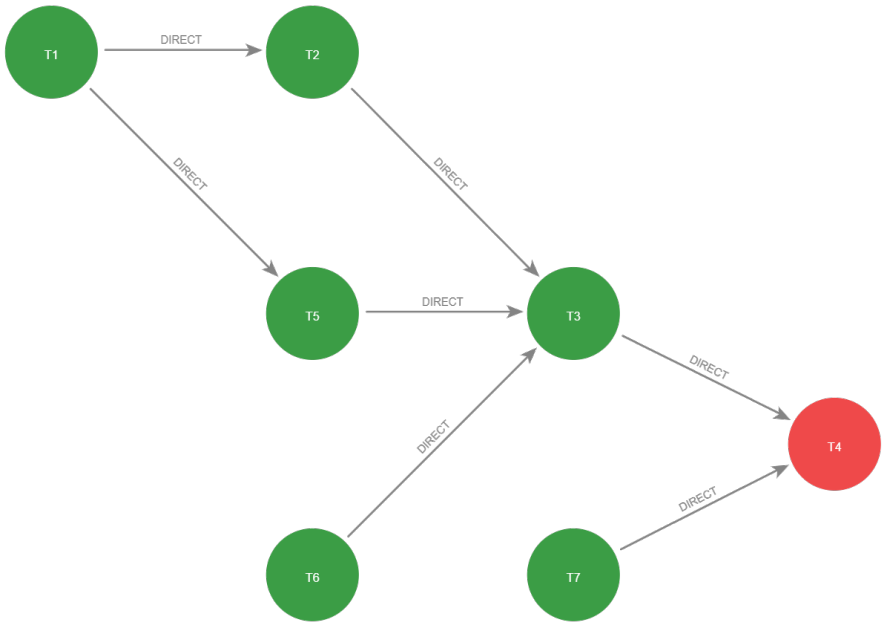

Figure 6.1: Data lineage example on master level with horizontal aggregation (green - tables, red - table for which we need the data lineage)

6.2 Neo4j Bloom

The Neo4j Bloom is a visualisation tool that freely interacts with the Neo4j graph data platform. It allows the user to connect to a graph database and visualise its contents. Furthermore, it also supports more natural-like language, which is much friendlier to business users, allowing for simple querying and filtering. It also supports custom queries written in Cypher, which are then executed with custom parameters with support for the nature-like language.

Another advantage of the Neo4j Bloom is the ability to display nodes topologically, which considerably helps with orienting in the data lineage flow. It also allows for custom rule-based styling, which helps to distinguish different types of nodes as well as specific essential nodes within the graph.

For example, we can visualise the following data lineage in the most commonly used view, where we display only tables. The visualisation can be seen in the Figure 6.1, although these images are anonymised parts of the data lineage, as the original dataset contains proprietary data. There, we can see aggregated edges between tables signifying direct data flow.

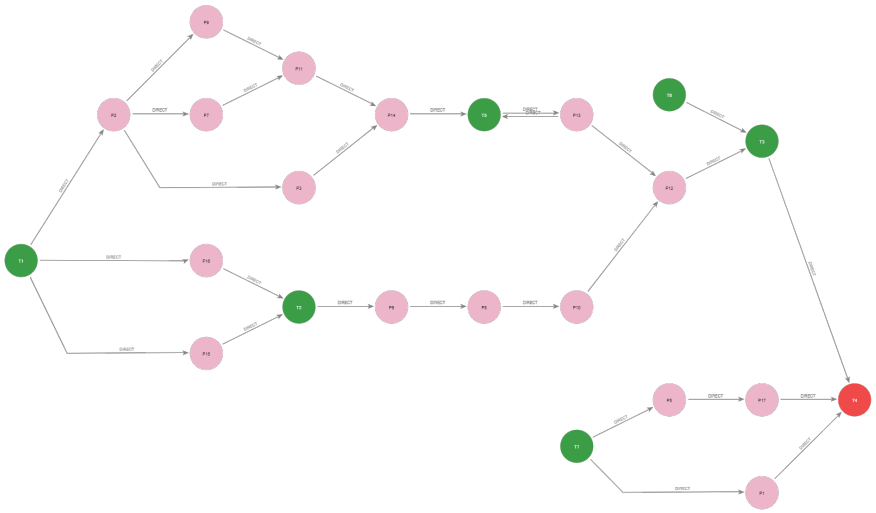

Figure 6.2: Data lineage example on master level with procedures
(green - tables, pink - procedures, red - table for which we need the data lineage)

In the second example, the Figure 6.2, we can see the visualisation of the data flow of the same dataset without the horizontal aggregation, which hides procedures transforming the data along the datapath. In this view, we can further analyse data transformations between tables if needed.

It is also possible to visualise the top level of our data lineage, in this case, the schema level. It is to be used for the larger picture visualisation, where we want to find associations between databases or dependencies in system deployment. This visualisation can be seen in the Figure 6.3. It can also be visualised without the horizontal aggregation, giving us a similar graph as the master level.

On top of that, we can also combine the abovementioned views. We can start at the top view with schemas and expand the connections in vertical as well as horizontal directions. This way, we can leave less important tables hidden under a schema and display only the parts that are currently of interest. Moreover, we can also combine both expansions in one view, as you can see in the Figure 6.4, where we have expanded schemas S1 and S2. Afterwards, we have also expanded connections between tables T1 and T5 horizontally, displaying dataflow in procedures connecting these two tables. Finally, we have expanded the data lineage visualisation to the bottom-most level between

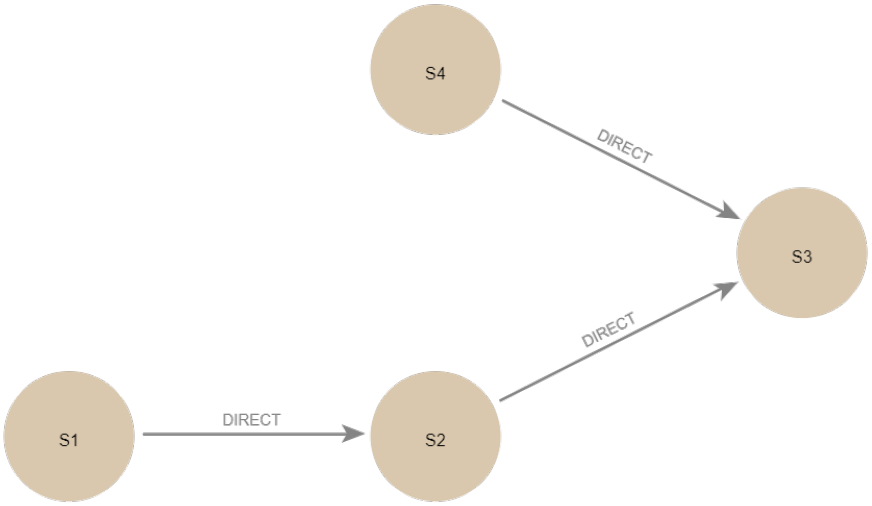

Figure 6.3: Data lineage example on top level with horizontal aggregation (brown - schemas)

tables T6 and T3, displaying columns connecting these two tables.

Furthermore, we can also use the aggregated relations on the edges of schemas. We can stop the path expansion using these relations once we cross the border between schemas. We can also set particular schemas for which we want to display the detail and skip the rest. Once the path is expanded, we can further expand the details we need for our specific issue. The resulting example of such a path can be seen in the Figure 6.5.

6.3 External tools

The Neo4j Bloom also offers the option to export selected nodes and relationships to simple CSV files. This option allows us to prepare the data lineage within desired borders and then export it for further processing. The Neo4j Desktop also enables us to export queried graph structures, which is helpful in loading larger graphs that would be hard to render within the interactive environment.

Once we have the exported graph, we can use our custom parser, which we have created for the purposes of this thesis, which parses the graph into a PlantUML source file. We can see the exported source code in the Figure 6.6 as well as the exported image in the Figure 6.7. The example contains the

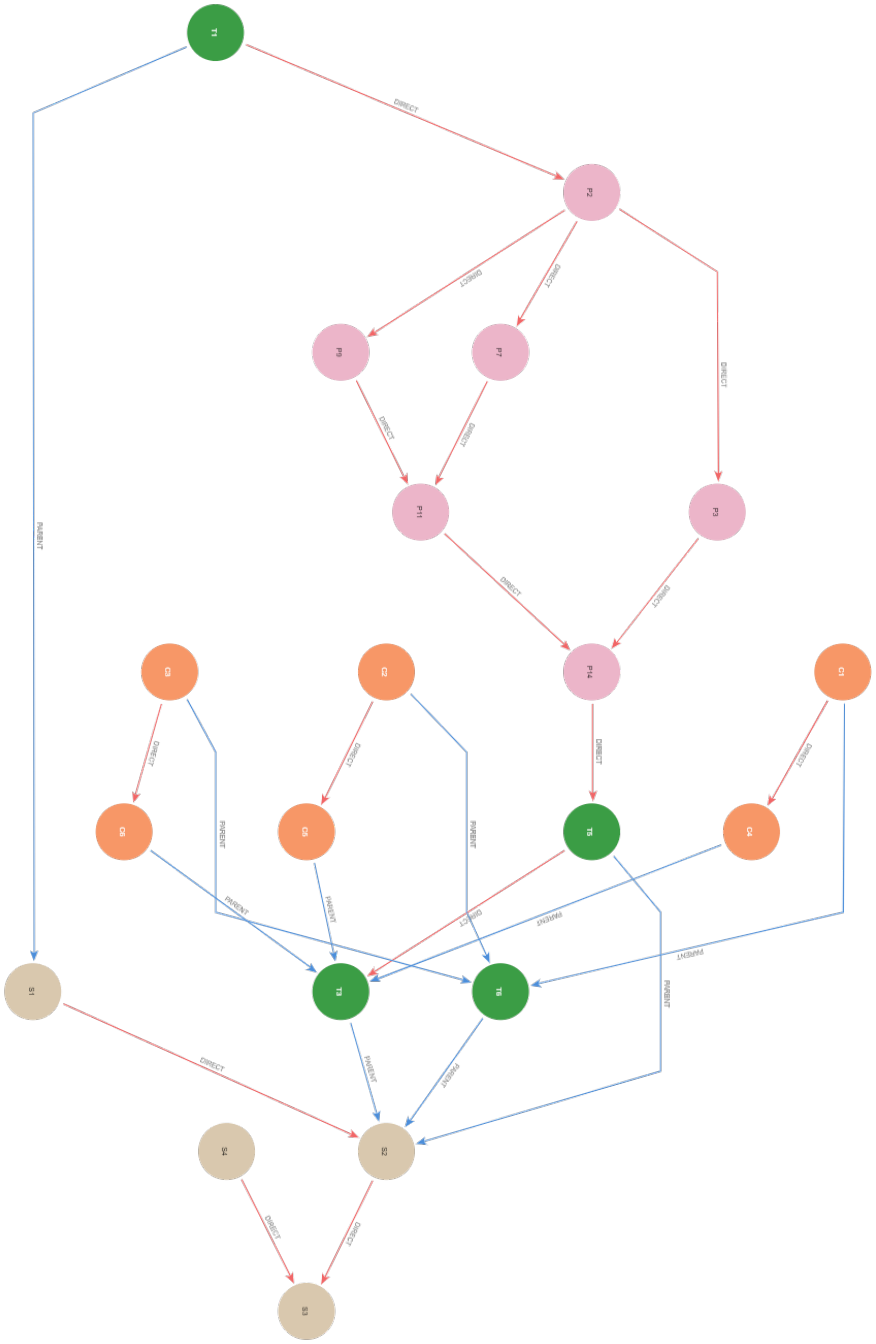

Figure 6.4: Combined data lineage example on multiple levels (brown - schemas, green - tables, pink - procedures, orange - columns blue edge - parent, red edge - direct data flow)

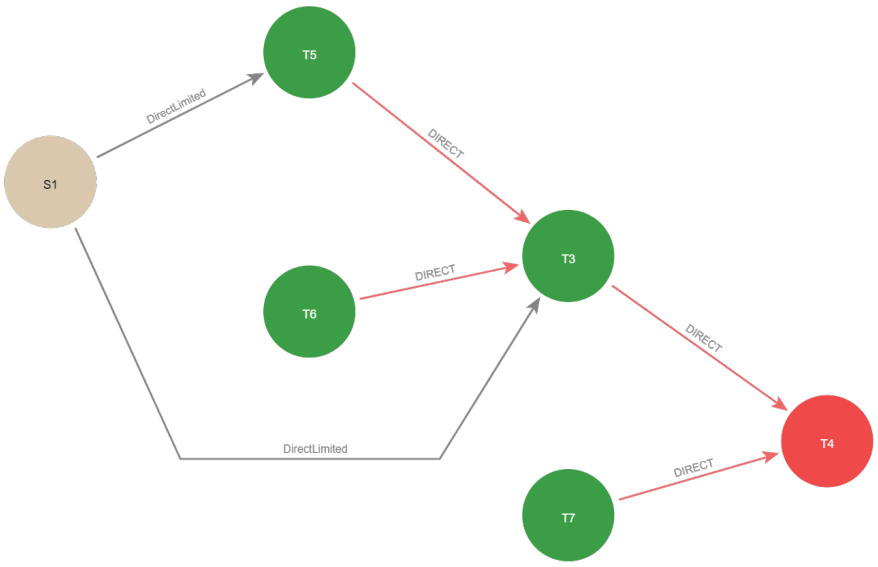

Figure 6.5: Combined data lineage example on multiple levels
(brown - schemas, green - tables, red - table for which we need the data lineage
red edge - direct data flow, black edge - limited direct dataflow at the border
of two schemas between schema and table)

same dataset as the Figure 6.4. It is also possible to display further details for each node, as well as to add custom information to each node. However, this detail is sufficient for the illustrative proof of concept purposes of this thesis.

To generate the image from the PlantUML source code, we use a free PlantUML service[22]. This service, however, stores the generated image publicly. Thus it will not be possible to use this publicly available version for production purposes. In the future, it would be better to use a local service or command-line tool for generating the resulting image in the desired format. The PlantUML also has an option to generate the other formats, such as SCXML, VDX or XMI, which can be further processed and enhanced. Once we set up the local service, command-line tool or library, we can integrate the generation into the Neo4j database itself as another procedure. This way, we can generate these diagrams on demand on any layer desired with simple parameters.

```
@startuml
skinparam rectangle {
 roundCorner 25
}
left to right direction
title export_example
'nodes
rectangle "T1" as 0 #FFE081
rectangle "T3" as 2 #FFE081
rectangle "S1" as 4 #D9C8AE
rectangle "T5" as 5 #FFE081
rectangle "T6" as 6 #FFE081
rectangle "S2" as 8 #D9C8AE
rectangle "S3" as 9 #D9C8AE
rectangle "S4" as 10 #D9C8AE
rectangle "C1" as 11 #F79767
rectangle "P2" as 13 #4C8EDA
rectangle "P3" as 14 #4C8EDA
rectangle "P7" as 17 #4C8EDA
rectangle "P9" as 19 #4C8EDA
rectangle "P11" as 21 #4C8EDA
rectangle "P14" as 24 #4C8EDA
rectangle "C2" as 28 #F79767
rectangle "C3" as 29 #F79767
rectangle "C4" as 30 #F79767
rectangle "C5" as 31 #F79767
rectangle "C6" as 32 #F79767
'relationships
5 --> 2
6 --> 2
0 -* 4
5 -* 8
6 -* 8
2 -* 8
32 -* 2
31 -* 2
30 -* 2
0 --> 13
13 --> 14
13 --> 17
13 --> 19
19 --> 21
21 --> 24
17 --> 21
14 --> 24
24 --> 5
10 --> 9
4 --> 8
8 --> 9
29 -* 6
28 -* 6
11 -* 6
11 --> 30
28 --> 31
29 --> 32
@enduml
```

Figure 6.6: PlantUML data lineage source

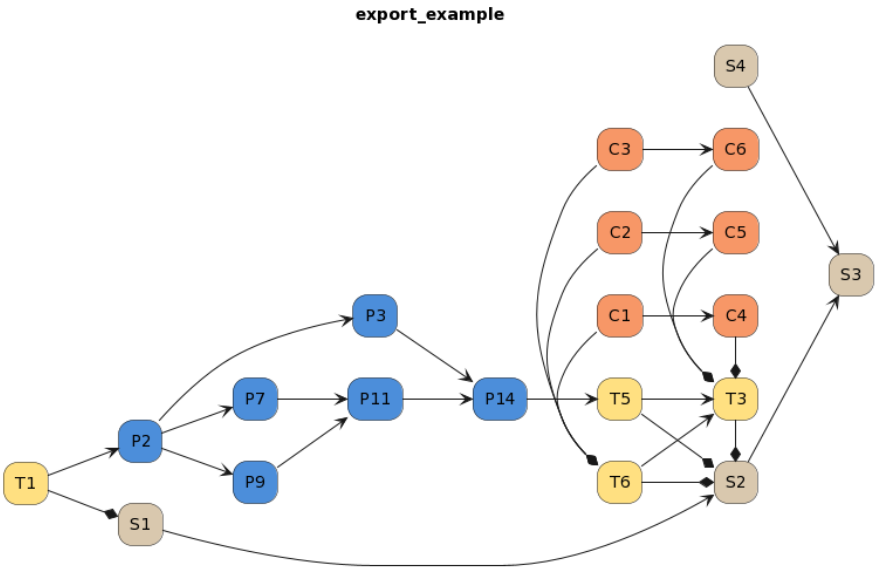

Figure 6.7: PlantUML data lineage diagram
(brown - schemas, yellow - tables, blue - procedures, orange - columns)

It is also possible to use the graph export to generate a FlowChart diagram, which also allows for integration to Confluence, which is widely used for documentation. The confluence integration also allows binding custom actions on the diagram elements. This way, we could view detail of any relation upon clicking on it. However, this would require us to either generate all the diagrams beforehand or generate them on demand.

To generate such a vast amount of diagrams would require us to automate the process entirely, which would require further development and integration. However, with some more work, we could generate the diagrams on demand by specifying the node of interest, whether we want its data source nodes or destination nodes, and the layer we want to display. We could also generate details of lower layers specified by two nodes of a higher layer and a target. This way, we could create a fully interactive tool that would require minimal user experience and could be operated by a simple double-click of a mouse to display the details of any relation and a simple full-text search for a node of interest.

It is also possible to export the graph into any other visualisation tool. However, since the graph in question contains hundreds of millions of nodes and

relations, it would be advisable to export only sections or selected layers of the graph. It would also be possible to connect any third-party visualisation tools which support the Neo4j integration, such as Perspectives[23].

Chapter 7

Evaluation

After implementing our solution based on the Neo4j graph database platform, we have come up with a few options for improvement. First and foremost, we would like to mention the importance of in-memory preprocessing of reachability, as well as horizontal aggregations. Since databases were never meant for calculating large operations with heavy performance requirements, the resulting time needed for these operations is often relatively high. Thus, our first and most important recommendation is to preprocess as much information as possible outside of the database.

Another great option is to use a custom in-memory procedure, which allows us to process the data in parts without the need to move them outside of the database. This approach might be problematic for larger datasets, as we could be limited by the RAM size of the machine. This limit can be bypassed by modifying our current algorithms to allow for problem division and solving smaller parts separately. However, this approach would overall require increased processing time.

Next, we would like to note that the most significant advantage of the graph database is the index-free adjacency which simply can not be achieved with the usual relational database. The index-free adjacency grants us high-efficiency improvement while traversing a graph, which is an operation that is the core of our data lineage visualisation use case. This advantage appears to place an unscalable barrier between the Neo4j and our usual relational database used in the original attempts to visualise the data lineage. This core difference between these databases projects into an exponential difference in time

needed for visualising the data lineage. The visualisation based on the Neo4j database we have implemented as part of this thesis can gather and visualise the smaller diagrams containing hundreds of nodes within seconds, as opposed to the attempts based on the relational databases, which take minutes to complete. The larger datasets with thousands of nodes then display even vaster differences in performance.

Furthermore, some operations are completely unfeasible based on the current solution without the preprocessing done on the Neo4j database. For example, displaying detailed information about the relation between two tables with hundreds of nodes distancing this relation from the target table in question, where we try to limit the details to only nodes based on the information which affects the target table, is not feasible. This operation would require us to gather the full bottom-most path over the entire path removed by hundreds of other nodes just to determine which nodes can eventually reach the target table and which do not. This problem could be resolved with the aforementioned preprocessing, which is not database related. Although, it would be advisable to split the relations into different tables by type, as there can be a vast amount of the preprocessed relations, which could significantly extend the time it takes to lookup a relation for each node within the relational database.

On top of the abovementioned issues, the current problem with the solution based on a relational database is its need to write extensive custom SQL queries for each visualisation, which takes hours or longer. This work also requires a skilled engineer with advanced SQL knowledge as well as elementary knowledge of the data lineage being visualised. This requirement might prove to be a significant problem, as it could potentially raise the cost of creating these visualisations. This issue could be resolved with the use of some advanced visualisation tool, which allows a business user to interact with the data visualisation and query the needed information on demand without the need for any extensive training. Any tool similar to the Neo4j Bloom should suffice. However, we will leave the research of these tools for future exploration since it is unnecessary for the purposes of this thesis, and the Neo4j Bloom provides sufficient options for our visualisation based on a graph database.

It is also possible to further enhance the data lineage with information from

other sources and add it to the visualisations. This enhancement could give us the full business data lineage. It should be relatively easy to implement it into the project, as the Neo4j database does not have a rigid structure, and it can be modified at any given step of the process. However, it would require considerable effort to gather all the information for the business data lineage to be complete. On the other hand, once the information was gathered and introduced into the metadata storage, the value of our visualisations would grow significantly.

Conclusion

In this thesis, we have analysed the options for visualising data lineage of various levels and granularities. We have met with the issue of visualising extensive data lineage and keeping the result easy to understand and comprehend. We have decided to attempt to create a proof of concept for dynamic data lineage visualisation, which was met with radiant support from the side of our partners from Profinit and Česká Spořitelna. We have also analysed the current attempts for a solution using a relational database, namely Oracle. At the time we started this thesis, all the results were unsuccessful. However, during the development of this thesis, the other party working on the project with the relational database managed to achieve a successful outcome, as they are mentioned in the Section 4.2. However, these results were achieved with the use of hardcoded SQL scripts, which required considerable time for each visualised data lineage diagram. These scripts also required a skilled data analyst to cherry-pick the nodes to visualise as well as the nodes to skip.

We have also compared the advantages and disadvantages of SQL relational databases and graph databases. Furthermore, we have focussed on the details of graph processing on various databases, as well as the level of the nativity of a graph database. Further, we have selected the Neo4j database as the best representative, as it is the most widely used graph database. Moreover, the Neo4j is also an industry-leading database system, which is fully native, giving the best performance while working with a graph-structured dataset. On top of that, the Neo4j also has a sizable open-source community developing

custom methods with an emphasis on performance.

Once we have selected the Neo4j database, we have worked on various approaches for building the metadata storage. At first, we started with python scripts, which turned out as a failure due to their performance issues. When testing this approach on a smaller dataset, we have been rather successful, but once we introduced it to the dataset provided by our partner Česká Spořitelna, we met with severe problems, and the scripts would finish only after a rather long time. Due to the abovementioned performance issues, we have decided to move forward with the use of native Neo4j queries. We have developed basic scripts for building the initial stage of the metadata storage. However, the native neo4j scripts were insufficient for the amount of data we had to work with. Thus we have decided to look for further options for improving our processes. We have finally succeeded in optimising the scripts for building the base of the metadata storage with the use of APOC library developed by the opensource community surrounding Neo4j. This library helped us to increase the performance of our scripts significantly.

Once we have built the base of our metadata storage, we have set to aggregate it horizontally and vertically to enable querying of various types of data lineages. We have aggregated new paths while skipping unnecessary nodes such as various transformations. We have also aggregated the most basic level data lineage to higher levels. This way, we have gained the option to query different data lineages containing various levels of granularity and also to query the data lineage for specific use-cases. At this point, we have reached the stage where we could query basic data lineages using the Neo4j database in seconds, while the approach using the SQL database took minutes to complete processing the query.

However, even though we could query the common data lineages rather fast, the more complex data lineages were still too extensive to query even with the Neo4j database, even though it gave us fully native graph processing. After meeting with the more prominent data lineages, which were harder to query, we have decided to look for further options for optimising our metadata storage. After further analysis of the Neo4j database, we have decided to use a custom procedure. We have thus created two different procedures for aggregating data. One for skipping transformations and other unnecessary nodes, which created direct connections between data nodes and the second

one, which created aggregated connections between each reachable node. This way, we have increased the storage needed for the entire metadata storage. On the other hand, we have gained information about all nodes which are part of the queried data lineage almost instantly. Thanks to this preprocessing, we have managed to query even the most complex data lineages within seconds.

Once we have completed the requirements for fast data lineage querying on multiple levels and granularities, we have decided to build a proof of concept for dynamic data lineage visualisation. We have decided to use the Neo4j Bloom application, which is a complex data analysis tool that seamlessly integrates with the Neo4j database. We have also tried the built-in visualisation tool in the Neo4j Dekstop application, but it was insufficient for our needs. With the neo4j bloom, we have gained the native-like query language, which is easy to use for users without extensive knowledge and experience with database systems. We can also extend this language with custom Cypher queries for querying specific types of data lineages as well as displaying detail of aggregated relations. This way, any user can dynamically expand the data lineage with the data they require without the need to go through unnecessary parts and details, which would only confuse the result.

Since the dynamic querying did not provide us with the option to save the resulting data lineage with its full information, we have decided to create another tool for exporting the data lineage into a static format. After analysing the options for this tool, we ended up choosing the PlantUML tool. This is mainly a UML visualisation tool, which supports mainly the UML generation from a source code. To generate the PlantUML source code, we have used a custom Python script, which consumes CSV files easily exported from the Neo4j Bloom containing the current data lineage. This script also allows us to generate the PlantUML source code for the data lineages of various levels and granularities. We can also generate an export of a specific data lineage directly from the Neo4j database and then use it to generate the PlantUML source code. Once we have the PlantUML source code, we can then generate the resulting graph in the desired format, such as SVG image or many other formats.

We have also devised a solution for generating these exports on demand for specified data lineage, granularity and level. This way, we could make a tool

that dynamically loads the data lineage as we go through it and generate the PlantUML source code and resulting graph. This way, we could make the tool more user-friendly and save time for the user. The user could easily go through the graph by simply clicking on an aggregated relation to get a more detailed visualisation of said relation.

In conclusion, we have successfully managed to devise a process for visualising static as well as dynamic data lineages of different levels and granularities. We have managed to optimise this tool to visualise the data lineages efficiently on demand. In the end, we have also devised further options for the development of this proof of concept, and we have met with a lot of positive feedback from our partners.

Bibliography

[1] Haerder, T.; Reuter, A. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.*, volume 15, no. 4, dec 1983: p. 287–317, ISSN 0360-0300, doi:10.1145/289.291. Available from: https://doi.org/10.1145/289.291

[2] About us: Automatically scans your data environment. Feb 2022. Available from: https://getmanta.com/about-us/

[3] Ikeda, R.; Widom, J. Data Lineage: A Survey. Technical report, Stanford University, 2009. Available from: http://ilpubs.stanford.edu:8090/918/

[4] Sharma, V.; Dave, M. Sql and nosql databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, volume 2, no. 8, 2012.

[5] Date, C. *Database in Depth: Relational Theory for Practitioners*. Theory in practice, O'Reilly Media, Incorporated, 2005, ISBN 9780596100124. Available from: https://books.google.cz/books?id=TR8f5dtnC9IC

[6] Hovad, J. *Metody pro práci s grafy v databázi*. Dissertation thesis, FIT VUT v Brně, 2011.

[7] Khan, W.; Ahmad, W.; et al. SQL Database with physical database tuning technique and NoSQL graph database comparisons. In *2019*

IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), 2019, pp. 110–116, doi:10.1109/ITNEC.2019.8729264.

[8] Kong, C.; Gao, M.; et al. ACID Encountering the CAP Theorem: Two Bank Case Studies. In *2015 12th Web Information System and Application Conference (WISA)*, 2015, pp. 235–240, doi:10.1109/WISA.2015.63.

[9] Zhou, Y.; Chen, Q.; et al. A Distributed Storage Strategy For Trajectory Data Based On Nosql Database. In *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, 2019, pp. 3487–3490, doi:10.1109/IGARSS.2019.8900482.

[10] Kanade, A.; Gopal, A.; et al. A study of normalization and embedding in MongoDB. In *2014 IEEE International Advance Computing Conference (IACC)*, 2014, pp. 416–421, doi:10.1109/IAdCC.2014.6779360.

[11] Robinson, I.; Webber, J.; et al. *Graph databases*. O'Reilly, 2015.

[12] Kusu, K.; Hatano, K. Recurrent Path Index for Efficient Graph Traversal. In *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 6107–6109, doi:10.1109/BigData47090.2019.9006295.

[13] Jouili, S.; Vansteenberghe, V. An Empirical Comparison of Graph Databases. In *2013 International Conference on Social Computing*, 2013, pp. 708–715, doi:10.1109/SocialCom.2013.106.

[14] Neo4j open source project. Nov 2021. Available from: `https://neo4j.com/open-source-project/`

[15] The neo4j graph data science library manual V1.8 - neo4j graph data science. Available from: `https://neo4j.com/docs/graph-data-science/current/?ref=desktop`

[16] Apoc.path.spanningTree - APOC documentation. Available from: `https://neo4j.com/labs/apoc/4.1/overview/apoc.path/apoc.path.spanningTree/`

[17] Apoc.path.expand - APOC documentation. Available from: `https://neo4j.com/labs/apoc/4.1/overview/apoc.path/apoc.path.expand/`

[18] Reference - OGM library. Available from: https://neo4j.com/docs/ogm-manual/current/reference/#reference:session:basic-operations

[19] Setting up a Plugin Project - Java Reference. Available from: https://neo4j.com/docs/java-reference/current/extending-neo4j/project-setup/

[20] Jan 2022. Available from: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

[21] Černý, J. Available from: https://web.archive.org/web/20070617093712/http://kam.mff.cuni.cz/~kuba/ka/pruchod.pdf

[22] PlantUML web server. Available from: https://www.plantuml.com/

[23] Graph database browser: Explore amp; analyze: Tom Sawyer Software. Available from: https://www.tomsawyer.com/graph-database-browser

Appendix A

Contents of CD

- `readme.txt` ..... the file with CD contents description
- `bin` ..... the directory with compiled programs
  - `custom-plugin.jar` ..... Neo4j plugin for aggregating paths
- `src` ..... the directory of source codes
  - `export_to_plantuml` .. Directory with export to PlantUML (including example)
  - `plugin` ..... Custom Neo4j plugin for aggregations
  - `scripts` ..... Scripts to query and build the metadata storage
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the Diploma thesis in PDF format