CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

Engine for pattern detection in graph database used as metadata storage for data lineage

Bc. Lukáš Jarrah
Open Informatics - Data Science

May 2021
Supervisor: Ing. Michal Valenta, Ph.D.

Acknowledgement / Declaration

I want to thank my supervisor Michal Valenta, PhD., for his valuable suggestions and my Manta colleagues for their provided insights and expertise that greatly assisted the research. Besides, I wish to thank RNDr. Petr Olšák for sharing the TEX template and advice.

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the “Guideline for adhering to ethical principles when elaborating an academic final thesis”.

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

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs (“software”), in accordance with Article 2373 of the Act No. 89/2012 Coll., the Civil Code, I hereby grant a nonexclusive and irrevocable authorisation (license) to use this software, to any and all persons that wish to use the software. Such persons are entitled to use the software in any way without any limitations (including use for-profit purposes). This license is not limited in terms of time, location and quantity, is granted free of charge, and also covers the right to alter or modify the software, combine it with another work, and/or include the software in a collective work.

In Prague on May 14, 2021

. . . .

Abstrakt / Abstract

Datové toky jsou tradičně zkoumány prostřednictvím zobrazení *data lineage* v grafickém uživatelském prostředí. Tato práce navrhuje inovativní přístup založený na automatické analýze vzorů datových toků za pomoci vyvinutého nástroje *rules engine*. V publikaci je předložen průzkum grafových databází se zaměřením na technologii Neo4j, která v Manta platformě slouží jako úložiště metadat. Práce nabízí detail konkrétních obchodních příležitostí pro obhájení přidané hodnoty vyvíjeného projektu. Dále je v práci navržen a implementován prototyp vyhodnocovacího nástroje. Jeho hlavní část zodpovídající za spuštění pěti vybraných pravidel je popsána detailněji se zaměřením na její výstupy, které jsou v podobě reportů následně srovnávány s manuálním zkoumáním v uživatelském prostředí Manty. Celý prototyp je na závěr vhodně otestován a ověřen s ohledem na jeho relevanci vůči různým typům zákazníků. Speciální pozornost je věnována výkonnosti konkrétních databázových dotazů napsaných v jazyce Cypher. Práce také kalkuluje s eventuálními možnostmi rozšíření a navrhuje další vylepšení, která by mohla být do tohoto nástroje v budoucnu implementována.

Klíčová slova: Data lineage, Datové toky, Grafová databáze, Neo4j, Cypher

Překlad titulu: Nástroj pro detekci vzorů v grafové databázi sloužící jako úložiště metadat pro zpracování data lineage

The traditional way of data flows inspection is a visual representation of *data lineage* in the graphical user interface. This work proposes an innovative approach to automatically analyzing data flows patterns by the developed *rules engine* tool. The thesis contains research of graph databases, aiming mainly on Neo4j, which is used as underlying storage in the Manta platform. To justify the project's business value, the work also includes a detailed list of specific use-cases. On this basis, the prototype of the rules engine is designed and implemented. The core part responsible for the execution of five selected rules is described, and the result reports are demonstrated and compared with manual inspection in the Manta user interface. Subsequently, the project is appropriately tested, focusing on performance testing of particular graph queries implemented in Cypher language. Eventually, the provided value of the rules engine for various types of customers is verified. Besides, the thesis introduces suggestions for future extensions and enhancements.

Keywords: Data lineage, Data flows, Graph database, Neo4j, Cypher

/ Contents

Tables / Figures

Introduction

The constantly growing amount of produced data from various sources led to the inevitable discovery of ways to save and organize data effectively. Database systems, which allow persistent storage and simple data access, are categorized into *relational* and *NoSQL* databases. The latter group example are *graph databases*, which provide a way to represent connected data based on graph theory.

Data lineage, which uses graph databases as an underlying storage, represents the map of data flows in a specific environment. In recent decades, large companies have recognized the importance of data possession, thus they have been collecting the majority of produced and controlled data. This often leads enterprises to the gradual expansion of their data warehouse consisting of many unmaintained systems, in which ensuring data reliability or security becomes unfeasible. Data lineage tools help enterprises to monitor their systems and create a visual representation of their end-to-end data journeys.

Although having a detailed description of data flows boosts confidence in own data, examining the lineage map in a graphical user interface has two main drawbacks. Firstly, it still requires a lot of manual effort because the production graphs typically consist of tens of thousands of objects, and secondly, human errors do happen from time to time. These reasons brought the idea of building a tool, further called a *rules engine*, that would automatically analyze the data lineage and create a report with detected graph patterns. The input to the engine is a set of user-defined rules, which deliver information leading to business value growth in terms of data reliance, security issues, and last but not least, recommendations for simplification of systems and processes.

A Gartner study describes a recent trend of ensuring user-friendly access for business leaders to valuable data [1]. The rules engine can transform the passive data lineage system into the active tool, which may help end-users discover important patterns in their data flows with minimal effort and less technical knowledge required. Eventually, it results in an efficiency increase and better resource planning for the entire company.

Objectives

The thesis's first objective is to research current practices of data storage in graph databases and describe the data lineage functionality in the Manta Tools project. Next, it is essential to collect customer requirements of rules with lineage patterns and analyze the business use-cases. The key aim of this work is to design and implement a rules engine prototype consisting of configurable rules detecting the selected graph patterns in a data lineage. Finally, the author should analyze the effectiveness of implemented algorithms and propose methods for more efficient evaluation.

Structure

To achieve the goals, the thesis is structured as follows. The theoretical part starts with an introduction to databases, focusing on NoSQL systems, particularly graph databases. On that account, the general graph theory is presented. In the next chapter, the data lineage concept and use-cases are explained. One of the popular data lineage products is Manta, therefore its platform architecture and metadata storage structure are discussed.

The practical part of the thesis begins by collecting functional and non-functional requirements of the application from the perspective of integration with the current Manta platform. At this stage, the author introduces the intended objectives to achieve. This section also contains detailed descriptions of particular rules with graph patterns and how they can increase business value.

The subsequent chapter contains a research of the selected graph database, Neo4j in particular, with data querying and implementation details. Besides, the rules engine representatives are analyzed, and the author considers the potential usage of available tools for the implementation part of the thesis. Then the architecture for the module containing the planned prototype is designed.

During the implementation stage, the author discusses the entire development process of the application prototype. The focus is put into the explanation of the logic of single rules and flexibility achieved by input parameters configuration.

The evaluation chapter presents the ways of software testing with the main emphasis on Neo4j testing tools. Primarily this section contains the evaluation of the performance of implemented rules. Besides, it includes an in-depth description of particular database queries implemented in Cypher language and proposed improvements leading to higher efficiency.

Finally, the author summarizes the fulfillment of all gathered requirements. He also suggests general ways to enhance the application in terms of architecture and performance, leading to simplifying future development.

Chapter 1
Theory of graph databases

This chapter should provide an introduction to the theory of graph databases. First of all, the ways of data storage into databases of several types are explained. The subsequent section focuses on graph databases, in particular. Primarily the graph model comprised of nodes and edges is presented. Then there are described methods of physically storing the graph model in graph databases and how to benefit from database indexes.

1.1 Outline of databases

With the growing amount of produced data from various computer systems, it has been called for ways to organize and store data persistently, effectively, and easily accessible. During the 1970s, it became popular to save data into *relational* databases. Although relational databases are still widely used, it was realized that they might not be optimal for specific tasks. Around the 2000s, people started to use new emerged modern *NoSQL* databases, which became popular. Specifically, it was discovered that for navigating connected data, it is convenient to use the concept of from graph theory and store the data into newly arisen *graph databases* [2].

1.1.1 Data storage

A database (DB) is a logically organized collection of related data typically stored electronically in a computer system. Data may have metadata stored together with them and should be self-describing. A software system enabling access to a database is called Database management systems (DBMS), whose core functionality is the storage, retrieval, and update of data. Database systems are responsible for data sharing, reliability, integrity, reusability, unified interface, security, administration, and maintenance [3].

There are mainly three layers of data models, and each one has a different purpose and is meant for a specific audience target. The data models define how the data are stored in the database and set relationships between data items. Moreover, data layers ensure consistency in naming conventions and help communicate within organizations [4–5].

- **Conceptual model** – Describes high-level real-world entities and relationships between them. The purpose is to organize, scope, and define business concepts and rules. It models information gathered from business requirements, *what* the system should do.
- **Logical model** – Specifies *how* conceptual components are represented in data structures but are independent of technology (DBMS). It includes entities, attributes, and relationships between them.
- **Physical model** – Describes *how* logical structures are implemented using a specific technology, so the conventions and restrictions of the used (DBMS) must be considered. The model is typically created by developers.

Databases may be categorized based on many different features. Modern database systems can be classified into two types – *relational* and *NoSQL* databases.

1.1.2 Relational databases

Relational databases are based on the relational data model and typically use SQL for querying and maintaining the database. They use structure, allowing us to identify and access data in relation to other data items in the database. Data in a relational database are often organized into tables. Tables have columns defining attributes and rows containing data records representing instances of that type of entity. Every row has a unique primary key used for identification. Understanding the relationships between the data is achieved by joining tables.

Transaction management must satisfy a few properties to guarantee data validity despite errors, power failures, and other problems that may happen during database transactions. These four properties created the acronym ACID [3, 6].

- **Atomicity** – Either all tasks within a transaction are performed, or none of them are, which means that partial executions are not allowed.
- **Consistency** – Any transaction will bring the database from one consistent state to another, therefore half-completed transactions are not allowed.
- **Isolation** – Transactions are independent, so the resulting state is the same whether the transactions are executed sequentially or in parallel.
- **Durability** – Once the transaction has been committed, the database will remain in the resulting state, even in the system failure, power loss, and other types of system breakdowns.

1.1.3 NoSQL databases

NoSQL databases use different ways of data storage other than the tabular relational database approach. They aim to provide features such as data distribution, simple horizontal scaling, design simplicity, replication support, simple API, eventual consistency, high availability, and more. In contrast to the traditional ACID properties, some NoSQL databases have loosened the requirements for strong consistency and data accuracy to achieve other benefits. The new acronym BASE consists of the following properties [6–7]:

- **Basic availability** – The system guarantees availability at all times, so there will be an immediate response to any request. However, the requested data may be in an inconsistent or changing state.
- **Soft state** – Because consistency is not guaranteed, the system state could change over time even without the new input, thus the state of the system is always soft. After some time, we only have some probability of knowing the state, but not assurance.
- **Eventual consistency** – Once the system stops receiving input, it will eventually become consistent. The data will propagate to every database machine sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

The particular suitability of a specific NoSQL database depends on the actual problem. NoSQL databases may be classified by a data model into a few main categories:

- **Key-value stores** work as a simple hash table using key-value pairs. Key is the unique identifier, and value can be any object. Therefore the values may only be accessed via keys, so they are not appropriate for complex data nor complex queries.
  Examples of representatives are Redis[1] and Amazon DynamoDB[2].
- **Wide column stores** use tables, rows, and columns, but columns' names and format may not be strictly the same for every record. Multiple columns may be grouped into column families that are used and stored together for more efficient querying.
  Examples of representatives are Apache Cassanda[3] and Apache HBase[4].
- **Document stores** use hierarchical tree structures such as JSON, XML, or YAML. Every document has a unique key, and documents are organized into collections. They are suitable for structured documents with a similar schema.
  Examples of representatives are MongoDB[5] and OrientDB[6].
- **Graph databases** store the data in the form of nodes and edges with various properties attached to them. The section 1.2 will provide a more in-depth description of graph databases.
  Examples of representatives are Neo4j (explained in detail in section 4.1), Memgraph[7], and JanusGraph[8].

1.2 Graph databases

A graph database (GDB) is a type of NoSQL database designed for convenient representation and effective querying of connected data. A key concept of the system is based on graph structure. This section consists of a brief overview of graph theory, ways of graph database representation, and methods of indexing those structures.

1.2.1 Graph

A graph is an ordered triple

$$G = (V, E, \varepsilon) \tag{1}$$

where

- V – a set of nodes, also called vertices
- E – a set of edges, also called relationships
- ε – the mapping function

$$\varepsilon : E \to \{\{x, y\} | \, x, y \in V, x \neq y\} \tag{2}$$

that maps every edge to:

  - an unordered pair of vertices for an *undirected* graph
  - an ordered pair of vertices for a *directed* graph

[1] `https://redis.io/`
[2] `https://aws.amazon.com/dynamodb/`
[3] `https://cassandra.apache.org/`
[4] `https://hbase.apache.org/`
[5] `https://www.mongodb.com/`
[6] `https://www.orientdb.org/`
[7] `https://redis.io/`
[8] `https://janusgraph.org/`

If the graph contains two different edges *e1*, *e2*, for which equation (3) holds, both edges *e1* and *e2* are called *parallel*.

$$\varepsilon(e1) = \varepsilon(e2) \tag{3}$$

A graph containing parallel edges is called a *multigraph* [8].

1.2.2 Data model

A GDBs are based on graph theory and uses nodes, edges, and properties to represent and store connected data. This data have valuable information in relationships of particular entities, which may be more important than the discrete information about single entities. Those relationships, which may include information about direction, labels, and properties, are then used for effective traversals. This is preferred to complex querying over linked tables in relational databases, which can be difficult to understand and hard to optimize [2, 9]. A graph is called a *property-graph* if its edges carry labels information and optional properties [10]. This general-purpose structure allows us to model many scenarios, such as fraud detection, recommendation engines, social networks, supply chain mapping, and many more.

Figure 1.1 shows the example of a simple social network. The graph consists of:

- nodes representing six entities:
  - labeled *PERSON* with properties *name* and *age*
  - labeled *BOOK* with property *title*
- directed edges representing seven relationships representing connections between those entities:
  - labeled *KNOWS*, *LOVES*, *HAS_READ* and *HAS_WRITTEN*, some of them also containing relationship properties *since*

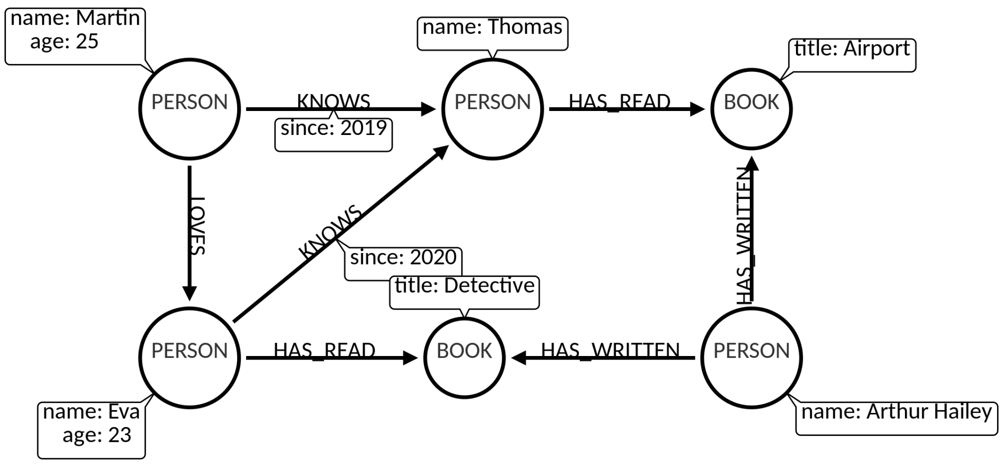


Figure 1.1. An example of a social network graph

A graph representation like this may contain millions of nodes and relationships and is convenient for database queries using relations, for example:

"Find the name of all persons that have written at least two books."

"Find the average age of people that have read any book from the person named Arthur Hailey."

1.2.3 Physical storage

Methods of physical storage differ among particular GDB. They are usually implemented as an extra layer over another NoSQL database type, a key-value, or a document store. Below there can be found the basic data structures for graph representations [11–12].

- **Adjacency list** – Nodes are stored as an array of records or objects, and every vertex stores a list of adjacent vertices. To improve access time to vertices, hash lists or hash tables are used instead of linked lists.
- **Adjacency matrix** – A square matrix in which the rows represent source nodes and columns represent destination nodes. If there exists an edge from node m to node n, then the matrix element at position *(m,n)* will have the value *true*, otherwise *false*. Instead of using boolean values with simple edge existence information in the adjacency matrix, it is possible to store other information, e.g., cost or number of edges.
- **Incidence matrix** – A rectangular matrix in which the rows represent the vertices and columns represent the edges. Every edge has value *-1* at the source node, *+1* at the target node, and *zeros* elsewhere.

Table 1.1 compares those graph representations in terms of the computational complexity of basic graph operations. Symbol $|V|$ represents number of nodes and $|E|$ represents number of edges in a graph. Obviously, different operations may benefit from various representations. Therefore GDBs use more ways of data storage representation for efficient execution of various tasks.

Graph operation	Adjacency list	Adjacency matrix	Incidence matrix
Storage size	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add node	$O(1)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add edge	$O(1)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Remove node	$O(\|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Remove edge	$O(\|V\|)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Are nodes m and n adjacent?	$O(\|V\|)$	$O(1)$	$O(\|E\|)$
Find all neighbors of node m	$O(\|V\|)$	$O(1)$	$O(\|E\|)$

Table 1.1. Comparison of graph data structures in terms of time complexity of basic graph operations [11]

1.2.4 Indexing

GDBs, just as relational databases, typically use indexes allowing faster database querying to allow more efficient data searching. Graph indexes are of two kinds – *value-based* and *structure-based*.

The first category, *value-based* index, allows efficient retrieval of nodes or edges by their properties. Those are used especially when finding the starting nodes of a given query. The indexes may be implemented by data structures such as binary trees or hash tables for every indexed property [9, 13]. When querying multiple properties, the results from more indexes may be combined. A typical index hash table contains key-value pairs consisting of:

- Key – particular property value
- Value – pointers to the graph nodes having given property value

The second kind, *structure-based* index, aims to extract and index structural properties of graphs in a database and allow fast querying of database patterns regardless of node and edge properties. A simple example may be finding all triangles in a graph, which means all triples of nodes having edges connecting all of them. Note that it is impossible to create a set of ideal structure indexes allowing efficient querying of any chosen pattern. However, a useful feature set improves the filtering power by reducing the number of candidate graphs. Unlike value-based indexes, structure-based indexes nowadays are not so developed and are yet rather the subject of research [9, 14].

Chapter 2
Data lineage

This chapter presents the term *data lineage* and illustrates how businesses may benefit from detailed knowledge of their data flows through presented use cases. The following section describes the architecture and components of a Manta product, one of the representatives of data lineage tools allowing the creation and visualization of data flows. In the last section, the metadata storage structure in the Manta ecosystem is explained in detail.

2.1 Data lineage

In recent decades, companies have discovered the importance of collecting and storing produced data because it can help them better understand their internal processes, customer behavior, demands, etc. Therefore keeping historical data may serve as a competitor's advantage. Companies typically store as much data as possible to avoid losing critical information because it may be difficult to distinguish between useless and potentially important data. With a constant increase in produced data in any organization, it becomes harder for companies to maintain all data in their ownership and monitor data flows within the company. *Data lineage* tools can help companies understand their data and data journeys in depth.

2.1.1 Description

Data lineage is an end-to-end map of the data flows, stating where data are coming from, where it ends, and what data transformations are applied as it flows through multiple processes and systems [15]. It allows understanding where the specific piece of data starts and when and where it separates and merges with other data. Often, this data manipulation process is not transparently documented but is kept in the minds of professionals or, in the best-case scenario, on local computers in the form of Word or Excel documents [16]. Although there are several ways of representing data lineage, visual representation in the form of a map is the most common as it allows a simple overview of the data flows [17–18].

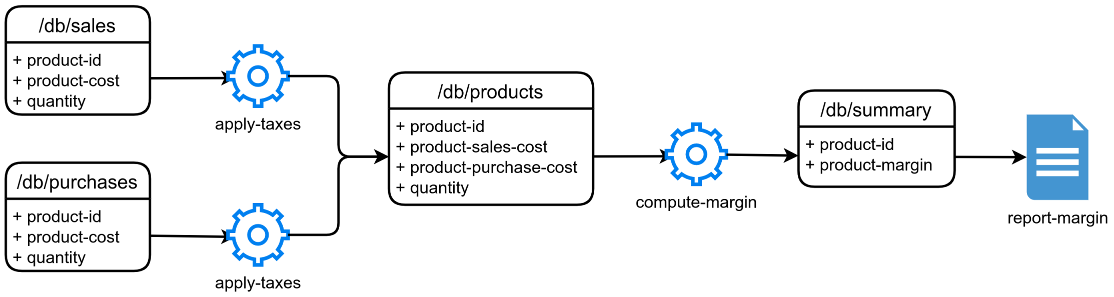


Figure 2.1. An example of a data lineage

Figure 2.1 shows a simple example of data lineage. There are two input database tables at */db/sales* and */db/purchases*. In the next step, both tables are transformed and joined into a single table */db/products*. This table is transformed by the script *compute-margin* resulting in a table */db/summary*. From there, the data are exported into the *report-margin* output file.

Up to date, there are many data lineage tools on the market. Examples of representatives are Talend Data Catalog[1], Octopai[2], Collibra Data Linage[3], SentryOne Document[4], Ovaledge[5], and last but not least, Manta, which will be discussed in detail in section 2.2.

2.1.2 Use cases

Even though understanding a data flow can help companies in many ways, there is still a significant number of enterprises that do not have their data lineage under control [19]. Without a deep understanding of data pipelines and agile development requirements, every change to the environment carries a high risk of broken releases [18]. Moreover, data in any company are tied by data protection laws, and lineage allows improvement of their data governance and fulfills compliance regulations [20]. In general, there are two most common concepts of information provided by data lineage.

The first type of question is to know the data's origin or provenance – the earliest instance of the data [17]. Imagine a manager in a large company that needs to make an important decision. Hence, he asks for a very important number, e.g., the main product's total margin. Before presenting this final number to stakeholders, he would like to ensure that this value is accurate, so he needs to know how exactly the number was calculated. He needs information on the origin of the number and how this number transformed during the chain through the systems [21]. In the end, the manager or BI team no longer needs to fear having to prove data accuracy in their reports [22].

The second type of question is *how* and *why* data has changed since the last time [17]. Imagine a customer coming to a bank with the intention of getting a mortgage. After filling in the necessary documents and waiting for a few minutes, the banker tells the customer that he cannot get a mortgage. Still, the reason why his application was rejected is unknown to both him and the banker. After a few months, the customer submits a mortgage application form again, and now he gets the mortgage. But the banker has no idea what has changed since the last time. All input information did not change, but the evaluating "black-box" engine probably used different transformations resulting in a different credit score. With access to the complete data lineage, any authorized person in a bank can see customer data flow through the company systems and understand the whole process in detail.

Nevertheless, it does not end up with only these two types of acquired knowledge. The companies may use the lineage to satisfy data governance, identify any issues in their particular data solution, find and secure the most crucial data object in the data warehouse, plan a parallelization of internal processes, and many more [18]. The concrete examples of how to extract valuable information from a data lineage is discussed in section 3.4.

[1] `https://www.talend.com/products/data-catalog/`
[2] `https://www.octopai.com/`
[3] `https://www.collibra.com/data-lineage`
[4] `https://www.sentryone.com/products/sentryone-document`
[5] `https://www.ovaledge.com/`

2.2 Manta product

One of the popular tools for data lineage is the Manta[1] platform (also called Manta Flow) developed by American-Czech company Manta Software, Inc. that was established in 2016. At the first stage, the data lineage is constructed by using scanners to connect to various parts of the environment, automatically gather all metadata by semantics analysis, and reconstruct complete lineage. Currently, many technologies are supported – databases (e.g., Oracle or Microsoft SQL), data integration technologies (e.g., Talend or Apache Pig), programming languages (e.g., Java or C#), and reporting tools (e.g., Tableau or Cognos). The lineage and all parsed metadata are saved in the graph database and versioned. In the next step, the customer may visually discover data lineage, adjust the level of detail as he needs and start exploring data flows in his entire system. A typical Manta customer is a large enterprise having a vast amount of data in a data warehouse (DWH) consisted of many resources [19, 23].

2.2.1 Platform architecture

Manta Flow is a client-server based application made up of three main components:

- **Manta Flow CLI** – Client command-line Java application performing extraction of metadata from databases, DDL scripts, storage files, and other sources. All metadata are analyzed, transformed into the representation of nodes and edges, and uploaded to the Manta Flow Server.
- **Manta Flow Server** – Server Java application processing metadata received from the client and storing them into the internal metadata repository. The server's data may be visualized by Manta UI, exported into supported systems, or consumed by third-party applications via API.
- **Manta Admin UI** – Server Java application providing a graphical interface for installation, configuration, updating, viewing logs, or process monitoring. This component allows easier maintenance of Manta Flow, so the end-user may perform most of the basic tasks from GUI instead of running specific shell or batch scripts.

[1] `https://getmanta.com/`

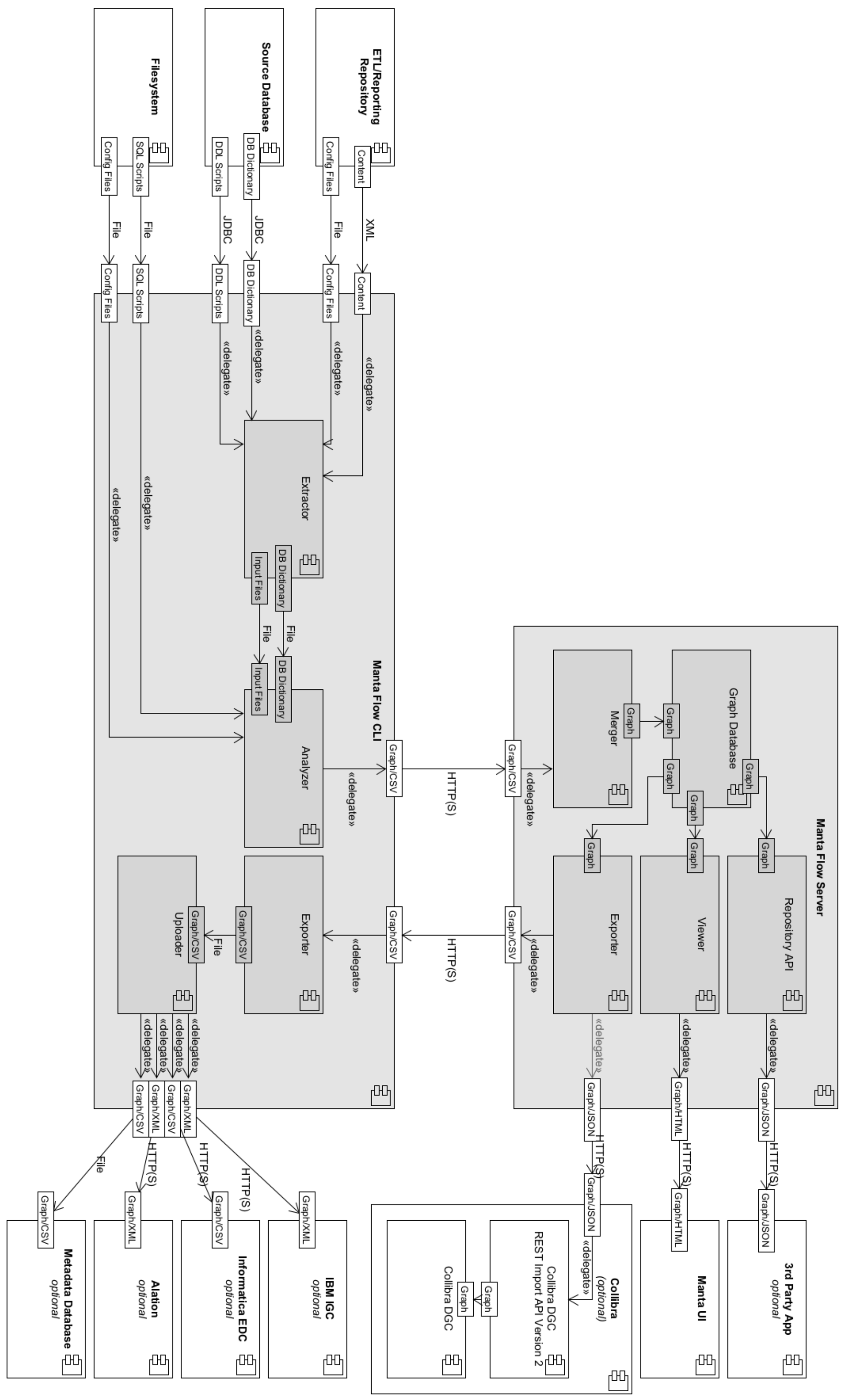


Figure 2.2. Architecture diagram of Manta Flow, components, and interactions with third-party resources (2021, February)

2.2.2 Components of Manta Flow Server

Figure 2.2 illustrates current Manta Flow's architecture, components, and interactions with third-party resources. In the figure, there can be seen data export to various technologies – Collibra DGC, IBM IGC, Informatica EDC, and Alation. The purpose of this section is to describe the main components of the Manta Flow Server.

- **Graph database** – A graph database is the storage of metadata received from Manta Flow CLI. At the time of writing this thesis, the underlying graph database is TitanDB[1], although the migration of unmaintained TitanDB to Neo4j is currently going on. A detailed description of the new Neo4j database may be found in section 4.1.
- **Connector** – The connector allows connection to the graph database and performs database queries. The module provides a set of basic database operations and flows traversals; below there are a few examples:
  - *get adjacent nodes and edges*
  - *get the path to the root*
  - *get the subtree*
  - *traverse the data lineage*
- **Merger** – The purpose of the merger is to save received metadata from Manta Flow CLI into the metadata storage. During the complex transactional process, new database objects are created and controlled to avoid duplications and inconsistencies. Furthermore, the merger keeps track of different versions of his data throughout different flow analysis runs [24].
- **Viewer** – The viewer provides data from a graph database to the graphical client interface implemented in JavaScript. The typical task is to provide the data lineage from objects selected by the end-user.
- **Exporter** – Exporter's responsibility is to extract the stored metadata and enrich them based on the particular third-party tool. The format of exported data may be technology-specific or a simple CSV file.
- **Pubic API** – Public API, also called Repository API, is an HTTP-based interface for Manta Flow Server. It allows the customer to call provided operations with custom parameters on their lineage using REST API. A few examples of allowed operations:
  - *find node by the name or path*
  - *import CSV*
  - *traverse the data lineage*
  - *get number of nodes and edges in the database*
  - *get nodes with the most or fewest edges*

  The purpose is to allow customers to directly query stored metadata by HTTP requests, which let them create new use-cases and enhance their processes. The API provides a way for advanced customers to work programmatically with data lineage within their environment and either analyze it for particular development cases or directly integrate with other solutions [25].

2.3 Metadata storage structure

The logical data model of the Manta repository is a *property* graph. Entity and relationship properties have a form of key-value pairs attached to nodes and edges. Edges

[1] `https://github.com/thinkaurelius/titan`

in the graph are always *directed*, and *parallel* edges are allowed. The Manta Flow repository defines the specific structure of nodes and edges, explained in this section. Because of the ongoing process of migrating to the Neo4j database, it is possible that in the near future, the logical model may change a bit to profit from all features that the new GDB offers.

2.3.1 Graph

No matter what system is extracted and processed, in the end metadata are always stored in a graph in the form of nodes and edges. Although the Manta Flow can analyze various resources, every system consists of data sources and targets, such as tables or files, and data transformations, such as ETL jobs, procedures, scripts, or macros [24]. All those are entities represented as nodes. Both nodes and edges have unique database IDs and allow an arbitrary number of properties attached to them, which are used, e.g., for metadata versioning.

2.3.2 Nodes

The nodes in the graph are of 9 types, which are described below. Because TitanDB does not allow node labeling, one node property named *vertexType* was reserved for defining the node *type*. It is very likely that after the transition to Neo4j, node types will be intuitively stored as the node *labels*.

- *SUPER_ROOT* – The node serves as an artificial root of all nodes of type *NODE*, and has edges to all nodes of *RESOURCE* type. There is only one *SUPER_ROOT* node in the database.
- *RESOURCE* – Representation of source systems technologies analyzed by Manta Flow, such as Oracle, PostgreSQL, Cognos, or just a Filesystem.
- *LAYER* – Representation of metadata model layers. The purpose of the layer is to distinguish between different model reality views. Therefore, customers may view their stored metadata from different perspectives based on the level of required technical details [26]. During initial analysis, the objects are stored in a physical layer. Later on, it is possible to enrich the metadata and create more abstract layers of modeled reality. An example may be a business layer representing the flows between the corresponding entities from a physical layer, but more understandable for business people.
- *NODE* – Nodes of type *NODE* represents real objects of source systems. An example may be a particular database, table column, script, folder, ETL job, or any other user-defined object in the data flow graph.
- *ATTRIBUTE* – Representation of supplementary information of nodes of *NODE* type. This information may be data type of column, description of database objects, etc.
- *SOURCE_NODE* – Representation of a source code file. The example is a database script performing transformations on data flows.
- *SOURCE_ROOT* – The node serves as an artificial root of all nodes having *SOURCE_NODE* type. There is only one *SOURCE_ROOT* node in the database.
- *REVISION_NODE* – Representation of revisions, which are used for metadata versioning.
- *REVISION_ROOT* – The node serves as an artificial root of all nodes of *REVISION_NODE* type. There is only one *REVISION_ROOT* node in the database.

2.3.3 Edges

The nodes are connected by edges of 10 different labels. In contrast to nodes, edges in TitanDB may have at most one *label*. If there is a need to have an edge with more labels between specific nodes, it is possible to create two distinct edges with different labels. In Neo4j, the label is called edge *type*. There are 8 labels representing the hierarchical structure of stored metadata listed below:

- *HAS_PARENT* – Edge between two nodes of *NODE* type creating basic hierarchical structure. Every node has at most one predecessor. An example *HAS_PARENT* edge starts in a particular table and ends in the database object.
- *HAS_RESOURCE* – Edge connecting *NODE* to the *RESOURCE* node, defining that given *NODE* belongs to a specific resource. Besides, this edge label is used to connect all *RESOURCE* nodes to the *SUPER_ROOT*. Every *NODE* has at most one *HAS_RESOURCE* edge.
- *IN_LAYER* – Edge connecting *RESOURCE* node to a *LAYER* node, which specifies that the particular resource belongs to a specific metadata layer.
- *HAS_ATTRIBUTE* – Edge starting in *NODE* with a target in *ATTRIBUTE* node assigning attribute values to a given *NODE*. Typically a *NODE* has many *HAS_ATTRIBUTE* edges.
- *HAS_SOURCE* – Edge connecting *SOURCE_NODE* node to the *SOURCE_ROOT*. The purpose is to keep *SOURCE_NODE* nodes accessible from the *SOURCE_ROOT*.
- *HAS_REVISION* – Edge connecting *REVISION* node to the *REVISION_ROOT*. The purpose is to keep *REVISION_NODE* nodes accessible from the *REVISION_ROOT*.
- *MAPS_TO* – Edge representing the connection of nodes of *NODE* type from different layers. An example *MAPS_TO* edge starts in a column named “NAME_FIRST” (in the physical layer) and ends in the attribute “First name” (in the business layer).
- *PERSPECTIVE* – Edge starting in *NODE* from the physical layer with a target in a *NODE* from a different perspective layer, representing the alternative parent of the source for an aggregate lineage. Aggregate lineage term is used to describe the capability of building simplified data lineage to display in Manta UI.

Furthermore, two edge labels are representing real flows in a data lineage. Those edges always connect nodes of *NODE* type on the lowest hierarchy level. In some cases, the flow edges may create directed cycles. The existing flow edge starting at column A with the target in column B means that column A’s value somehow affects the value in column B.

- *DIRECT_FLOW* – Edge representing a direct data flow from the source node to the target node.
- *FILTER_FLOW* – Edge representing an indirect data flow from the source node to the target node. Unlike the *DIRECT_FLOW* edge, the *FILTER_FLOW* edge affects target of a different *DIRECT_FLOW* edge.

Figure 2.3 shows a simplified example of the main graph hierarchy. The hierarchy edges are displayed by dashed lines, while solid lines indicate data flows. There is one *DIRECT_FLOW* edge from *Party.LastName* to the *Customer.Name* based on a filtering condition in the *Party.DIC* represented by the *FILTER_FLOW* edge.

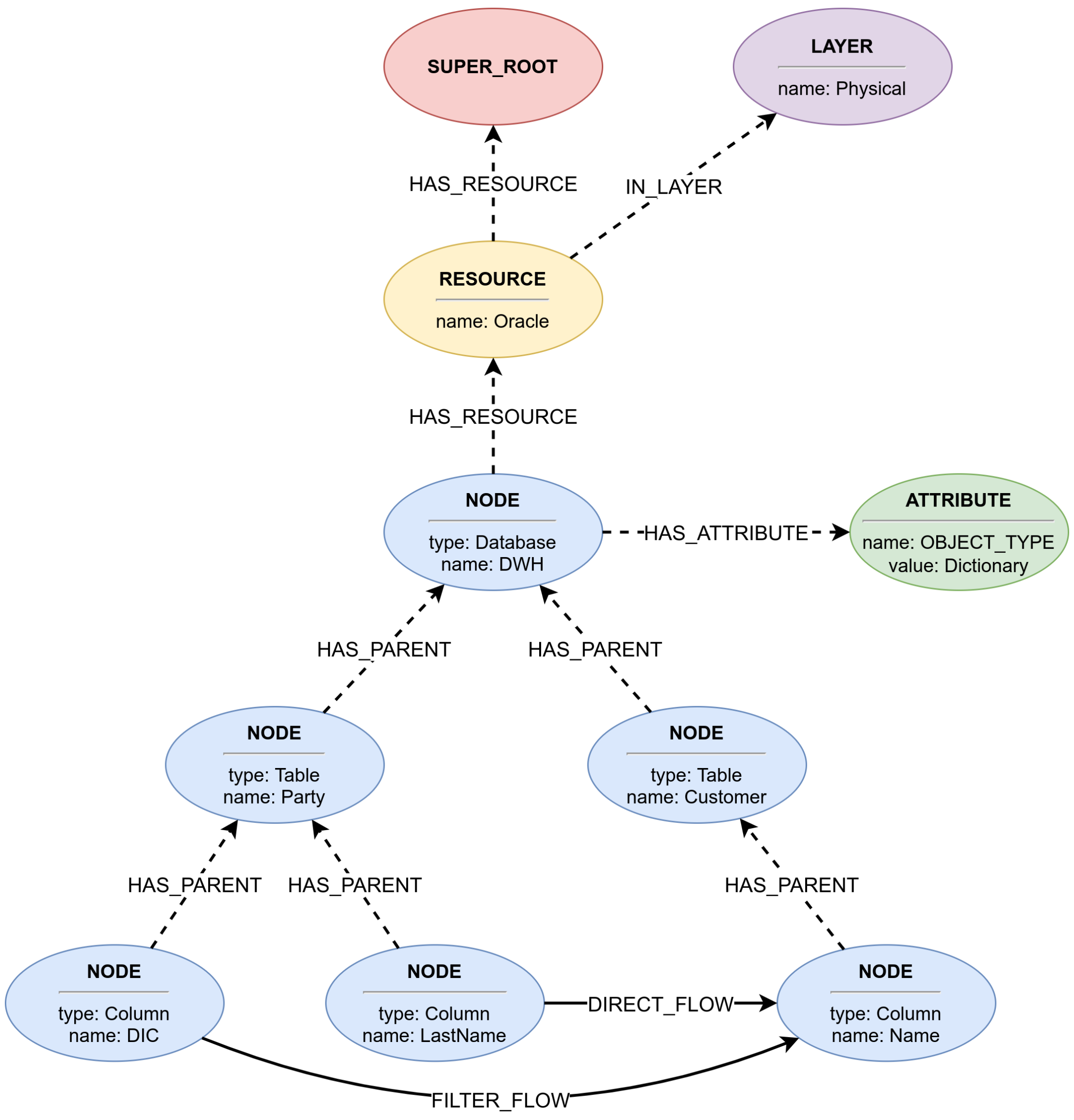


Figure 2.3. Simplified example of the main graph hierarchy.

2.3.4 Indexes

To allow effective querying of the graph, Manta storage uses various types of indexes. These are discussed very briefly in this section because it is very likely that they will be replaced to suit the needs of the new Neo4j database during the proceeding migration process. The types of currently used indexes are:

- Root indexes – Indexes pointing to the particular root nodes of the model, which are *SUPER_ROOT*, *SOURCE_ROOT*, and *REVISION_ROOT*.
- Node *childName* indexes – Indexes allowing faster access to a *NODE* child with a given *nodeName* property.
- Edge property indexes – Indexes allowing faster traversal of edges based on their label and properties, such as revision versions.
- Fulltext *nodeName* index – Indexes allowing a search of *NODE* given by the *node-Name* property.

Chapter 3
Requirements

The previous chapter focused on the data lineage and how the metadata is represented in a graph database. Yet, no explanation of how to consume stored metadata from the repository was given. The present-day approach is to inspect lineage *manually*, specifically, in the GUI. On the contrary, it would be helpful to provide some additional tool, called the **rules engine**, that would *automatically* examine data with almost no effort by the end-user. This chapter aims to gather and analyze business, user, functional and non-functional requirements about the new rules engine. The requirements will form the basis for the process of design a core component of the implemented prototype.

First of all, the general motivation behind the rules engine and the potential impact on customers is discussed. Also, the retrieval methods of data lineage information in the current Manta environment are summarized. The next section collects both functional and non-functional requirements gathered during the initial project phase. After that, the general objectives of this project are explained. The final and most extensive section of the requirements stage presents use cases of specific business rules with detailed descriptions.

3.1 Motivation and ways of lineage interpretation

When the metadata is loaded to the Manta Flow Server into GDB according to the storage structure outlined in section 2.3, the end-user can inspect the constructed data lineage map. One way, which is the most common, is to depict the lineage in the Manta UI visualization tool, as shown in figure 3.1. Although graphical visualization may dramatically simplify dataflow inspection compared to the traditional approach without data lineage tool, for customers with many systems, even this task may turn into a very time-consuming job with no room for mistakes. Moreover, the end-user may not be confident about the patterns, data flows, or structures he is searching for because many corporate departments may have various demands, and every division may seek different lineage information.

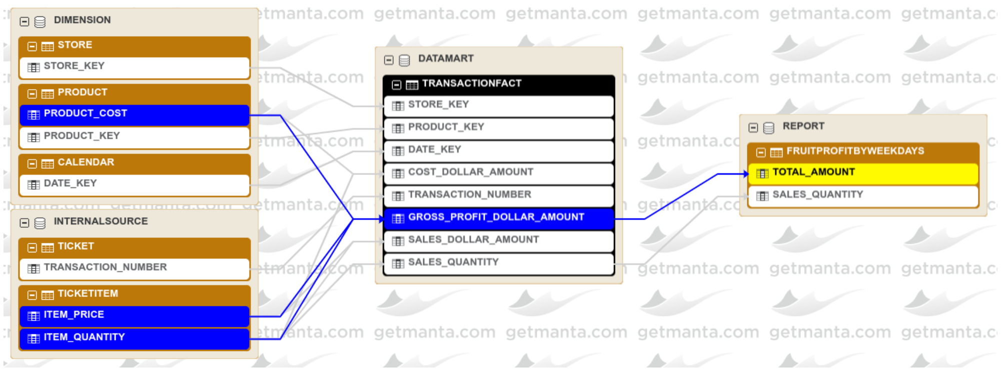


Figure 3.1. A simple data lineage visualization in the Manta UI [27]

Currently, the Manta ecosystem provides three general methods of data lineage ingestion from the database. Metadata can be graphically represented in the Manta UI, integrated with third-party applications, or requested by Public API, and those methods are described below.

3.1.1 Manta UI

As mentioned in the previous section, the fundamental approach is a visualization of the map of data flows in the Manta UI, where the end-user can display chosen parts of the lineage, focus on particular data resources, filter flow types, node types, etc. Moreover, he can intuitively extract or contract selected objects to examine the flows in more detail or see a global overview. However, this process still requires a lot of manual effort because if the user wants to perform a profound data inspection of the whole resource, he has to go through each lineage object separately.

3.1.2 Third-party tools

Besides, it is possible to integrate third-party applications and add new information to those applications' storage. The integrated tool, e.g., Alation Data Catalog[1], automatically connects to the Manta repository and enriches its data with the knowledge of data flows. Then the particular tool provides combined data from both sources in one system, which may bring a new added value to the customers [28]. When writing this thesis, Manta supports integration with eight various tools, and more direct integrations are expected to come.

3.1.3 Public API

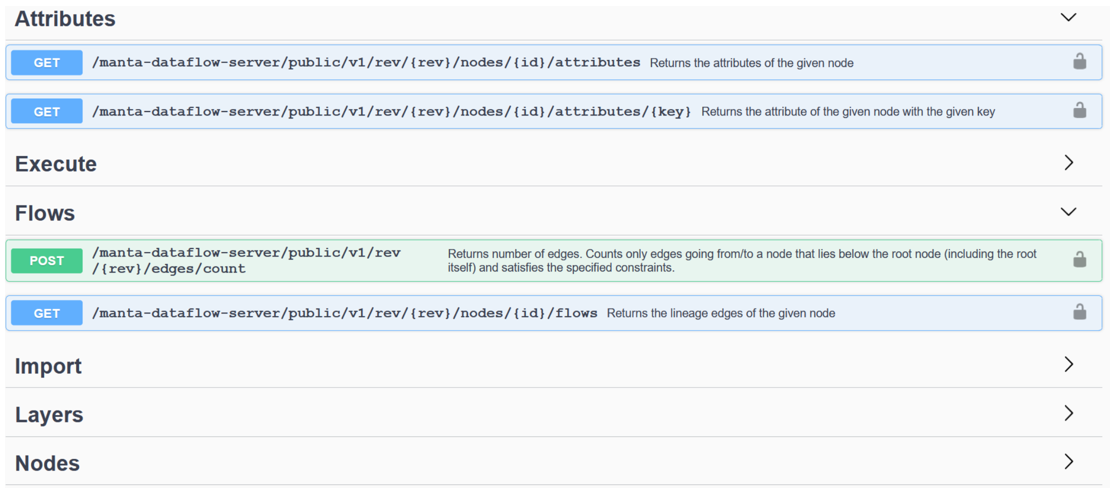
A third way to consume metadata from the Manta repository is by the Public API server component, which provides a public REST API. Therefore the customer can use the information provided by Manta from any custom tool he is already familiar with, which communicates with the Manta server by HTTP/HTTPS requests. Consequently, it can be utilized to optimize the customer's environment to enhance processes, resulting in higher efficiency [25, 29].

To provide documentation of API and graphical interface for interactive creation of HTTP requests, Manta's customers may use the Swagger[2] framework that follows OpenAPI specification, as shown in the image 3.2 [30]. Moreover, advanced users may also call Public API methods from Groovy-based DSL scripts.

The API provides a set of functions executed by sending a request to the Manta server. Those functions may be parameterized, e.g., specify starting nodes, the revision number, or define specific node attribute constraints. Below there are a few examples of operations with a brief description:

- *Find nodes by the name* – Finds nodes matching the given name.
- *Find node by the path* – Finds the node with the given path.
- *Get node attributes* – Get the attributes of the given node.
- *Get all resources* – Get all the resources in the repository.
- *Import CSV* – Imports data in CSV format into the database.
- *Get nodes with the most or fewest edge* – Finds nodes with the most or fewest edges.
- *Traverse* – Traverses the graph and returns information about the nodes and edges visited.

[1] `https://www.alation.com/`
[2] `https://swagger.io/`



Attributes

GET `/manta-dataflow-server/public/v1/rev/{rev}/nodes/{id}/attributes` Returns the attributes of the given node

GET `/manta-dataflow-server/public/v1/rev/{rev}/nodes/{id}/attributes/{key}` Returns the attribute of the given node with the given key

Execute

Flows

POST `/manta-dataflow-server/public/v1/rev/{rev}/edges/count` Returns number of edges. Counts only edges going from/to a node that lies below the root node (including the root itself) and satisfies the specified constraints.

GET `/manta-dataflow-server/public/v1/rev/{rev}/nodes/{id}/flows` Returns the lineage edges of the given node

Import

Layers

Nodes

Figure 3.2. The Swagger documentation of Manta's REST API interface

3.1.4 Rules engine

Although the three discussed approaches offer different ways of consuming repository metadata, they may not suffice for all business cases. Table 3.1 compares those three methods in terms of intuitiveness of user interface and interpretation of results (so that the end-user intuitively understands the output), ability to customize parameters, and automation running with no user's involvement. As can be seen, no approach offers sufficient parameters flexibility allowing full customization. Moreover, all methods can still be considered passive systems due to the incapability to be fully automatic.

	Manta UI	Third-party tool	Public API
Intuitiveness of UI	High	Medium	Small
Simplicity of output	High	Medium	Small
Parameters flexibility	Small – Medium	Small – Medium	Medium
Automation	Manual	Semi-automatic	Semi-automatic

Table 3.1. Comparison of current ways of metadata extraction from Manta's repository

On the grounds of the presented reasons, Manta's ecosystem demands a new way of metadata extraction and interpretation to solve discussed problems. The entirely new tool, called rules engine, will serve these purposes and provide a mechanism allowing the customers to automatically scan their data lineage and search for particular patterns customized by preconfigured conditions.

What should primarily differentiate the rules engine from the Public API is the usage simplicity. In the Public API, the customer has to create his own program if he desires complex scenarios benefiting from requests chaining in a way that the result of one operation is input into another one. On the contrary, the rules engine will provide a set of predefined rule templates targeting the common use cases. The user will not have to code anything by himself but will get a full solution completely prepared by Manta with convenient access. Examples of detected lineage patterns may be data leaks from one source system to another, duplicated objects, redundant tables or whole databases, too complex objects in the data lineage, or assurance of a safe shutdown of a given system. The reader may find more use case examples in section 3.4.

3.2 Requirements analysis

This section describes the requirements of the planned rules engine application. Although it is not assumed that the implemented prototype will offer all required functionalities, it is crucial to consider long-vision requirements when designing the software architecture allowing sufficient flexibility and space for scalability for future development.

3.2.1 General

The rules engine's general purpose is to allow customers to discover patterns in their data lineage and take specified actions with the minimum involvement from the end-user. The engine should ease the development and maintenance by separating business logic from the source code. Every provided rule should focus on a single type of pattern in the metadata storage.

The rules engine should allow users to select the particular rules, configure them to their own needs, and immediately run the rules or save them for repeated usage. When the rules with specific patterns are evaluated, the engine tool will create a report or alert the user to take action or focus on the detected parts of his system and quickly inspect the flow in the UI. With this new feature, it is much easier for customers to make important decisions, and the whole process becomes streamlined [25].

Software system requirements may be categorized into a few groups based on various metrics. At this place, the author divided the requirements into functional and non-functional requirements. The first category, functional requirements, describes the system's real functions and capabilities, simply put – *what* the system should do. The other class, non-functional requirements, specifies *how* the system should behave, under which condition, the system's limit, etc. Besides, non-functional requirements may also put constraints on system performance, expected maintainability, flexibility, security, or documentation quality [31–32]. In section 7.1.1 the author will evaluate how the implementation managed to fulfill initial requirements.

3.2.2 Functional requirements

The following list shows numbered functional requirements with their description that will be used for future reference.

(i) **The server repository will use Neo4j** – Due to parallel activities regarding the complete migration of all Manta repositories from TitanDB to Neo4j, the rules engine project will also be integrated with the Neo4j database. More arguments are supporting this decision:

- Unlike the TitanDB with TinkerPop[1] framework, which is in Manta in unmaintained version 2.6 and queried without a query language, Neo4j offers very different ways of building graph queries, especially by using the Cypher language. More will be discussed in the Neo4j section 4.1. Hence it will not be necessary to completely rewrite the rules engine project after the migration process is fully completed.
- It brings the opportunity to get a more hands-on experience with Neo4j, such as exploring Neo4j Java APIs, testing libraries, using stored procedures from

[1] `https://tinkerpop.apache.org/`

remote libraries, or sharing custom stored procedures across more concurrent projects.

(ii) **Flexibility and simplicity when parametrizing rules** – The rules should be implemented to allow flexible configuration by optional input parameters. On the one hand, the rules should provide default configuration options allowing the user initial running of a rule with almost no starting effort. On the other hand, when the user gets more familiar with a particular rule, he must have options to configure it to his specific needs by replacing the default parameters with custom ones. Moreover, the manta storage model may change in the future, so the engine should be easily expandable.

(iii) **Well-tested code** – Likely, rules will not be fully locked up once implemented because the queries may be tuned up in the future or new rule parameters may be added. Therefore, it is necessary to provide enough unit and integration tests to verify the function of rules and the whole rules engine when making implementation changes of rules and allow future refactoring of the application.

(iv) **Integration into the current Manta ecosystem** – Where possible, try to reuse the present Manta component and features and avoid reinventing the wheel. For example, if the rules engine will provide its REST API, integrate the new one into the current Public API component.

(v) **Exception parameters for every rule** – For various reasons, every rule should have optional parameters allowing to skip evaluation for a specific resource, server, folder, etc. With some paths defined as ignored, the performance may be improved (only a smaller part of the graph is traversed), and also the result may be more readable. Moreover, the exception definitions may be defined by the regular expression patterns, such as, *"don't evaluate this rule for all tables having prefix 'TEMP_'."*

(vi) **Provide input validation mechanism** – It is expected that the rule engine should be error-prone as much as possible and have instant evaluation functionality. Applying or modification of an existing rule or creating a new rule should be intuitive and should not generate confusing errors that would force the user to contact Manta support to help him. Suppose there is no way to avoid creating an invalid rule in the given customer context (missing coma, invalid query, etc.). In that case, the error messages should be intuitive enough so that the user can correct the input parameter syntax on his own.

(vii) **Triggering** – Although the primary function is to run the engine on demand, the end-user should also be able to customize the way of rules triggering:

- *On-demand* – The main functionality.
- *Time-scheduled* – The user would schedule the time or periodic intervals.
- *Action-triggered*: – Triggered by a specific action, such as after each scan, committing major or minor metadata revision, or when a scan of a selected resource is finished.

(viii) **Results reporting** – When the rule is executed, a few suggested ways of report form are suggested. Primarily, there should be provided a simple report file with a standardized structure (e.g., JSON file) containing important information, such as permalinks or node paths to the affected objects. Later on, action message notifications or alerting reports may be implemented in various configurable ways:

- *Write to the document* – Simple approach stored persistently.
- *REST API* – Requests into customer's existing endpoints.

- *Invocation of a shell script* – E.g., when the rule report has a non-empty result, run the predefined script enriched with information from the report.
- *Kafka message* – Send an instant message into the Apache Kafka[1] customer's client.
- *Alerts into the current Manta UI* – Alert immediately or after the next login.
- *Email message* – Send a message to the selected group of affected people.
- *Slack message* – Send an instant message into the customer's Slack[2] channel.
- *Creation of Jira ticket* – Create a ticket in the Jira Atlassian[3].
- *Update of nodes attributes* – Update a custom node attribute in the repository.

3.2.3 Non-functional requirements

The following list shows numbered non-functional requirements with their description that will be used for future reference.

(i) **Diverse technical skills of end-users** – It is expected that end-users will have various degrees of technical skills; therefore, it is important to design the system intuitively, that all target users are capable of running the rules engine and interpret the results with the provided documentation.

Business users or data analysts should be able to run simple rules with predefined parameters and appropriately interpret the output, data scientists with little coding skills may modify attached templates or create simple rule workflows, and the most advanced developers may adjust the rule parameters in a very detail by an API or configuration file with standardized format (e.g., JSON or YAML).

(ii) **Scaling flexibility** – The rules and particular components should be implemented in a generally independent way so the engine may be scaled up in the future – it should be possible to add new features, such as new options of launching the engine or reporting technology, combine the rules to create more complex ones or change the ways of scheduling. This should be achieved by a well-designed architecture, where the components communicate through suitable interfaces.

(iii) **Documentation** – The entire rules engine should be well-documented. The user's documentation should be detailed and informative, and every rule should be provided with at least five demonstrative templates of parameter settings. The customer may simply open the template, copy and paste the content, and run the rule with 1–2 parameter values changed to his needs, enabling him to adapt quickly. In addition, it is crucial to provide explanatory code documentation in the form of Javadoc, allowing easier maintenance, bug fixing, and code refactoring in the future. Finally, it's necessary to include a descriptive README file.

(iv) **Performance** – Rules should make use of well-tuned database queries. The majority of rules should be finished promptly on average database size so that the end-user may see the result immediately after launching the rule. If that is not possible for various reasons (e.g., too complex query, enormous customer's database), the end-user should be warned about the delay of results.

(v) **Localization** – The code, documentation, and templates should be provided in English.

[1] `https://kafka.apache.org/`
[2] `https://slack.com/`
[3] `https://www.atlassian.com/software/jira`

3.3 Objectives

As mentioned in the previous section, the list of functional requirements and long-term visions is extensive, and complete implementation with all functionalities is out of this thesis's scope. Nevertheless, the author aims to achieve the following goals in this project:

- In collaboration with Manta colleagues from various departments, especially from presales, products, and marketing sections, precisely define at least five business rules for the rules engine project. Explain how the customers may benefit from particular rules and how they may encourage existing and future partners to develop their added-value solutions based on the Manta platform.
- Analyze the popular business rules engine tools and discuss whether they may be used in this project.
- Get familiar with tools integrating Neo4j – with the database itself, ways of constructing queries (e.g., Cypher language or traversal frameworks), provided developers tools (e.g., Neo4j Browser or profiler), Java libraries, and ways of automated testing.
- Design and implement the core part of the rules engine – the component responsible for connecting to the database and evaluating database operations.
- Implement five selected rules with configurable parameters allowing basic customization to the user needs. Besides, it is required to implement an interface with basic graph operations. The rules queries should be implemented mainly in the Cypher, but more ways of querying, such as stored procedures, may be also provided.
- Validate developed rules on testing datasets provided by Manta. Furthermore, check the outputs, verify whether the results are faithful, and inspect potential false positives.
- Analyze the implemented algorithms' effectiveness, discuss and propose methods for more efficient detection of patterns in graphs for further research.
- Discuss encountered difficulties when implementing the particular rules.
- Summarize the complete process of creating a rule and discuss the eventual possibility of the end-user creating an entirely new rule from scratch by himself with no Manta employees' involvement.

3.4 Use cases of business rules

This section describes particular rules discovering lineage patterns and how those rules could improve customers' data storage environment and simplify their internal processes. The regular Manta's client is a corporation storing data in a large DWH consisting of many technologies. The typical problems are that some of those resources are unmaintained, data flows are not documented, some ETL procedures are redundant, etc. Users also use Manta to monitor data flow in their source codes or want assistance with an increase in particular workflow tasks' effectiveness. To sum up, the users seek to simplify their storage systems, and for privacy reasons, they also need to be assured that their data does not leak out of secured systems.

As discussed in the previous section, it is expected that the rules engine will be implemented with preconfigured rules. Therefore it is necessary to find several universal use cases to target the majority of customers. The concept and reason for specific rules usually come from customer's business divisions; thus, the following list of rules resulted from fruitful discussions of the thesis author with the presales, products, and marketing Manta representatives who assess the user's needs on a daily basis.

The intended end-user of the rules engine may not be only a software developer but also a less technical user with just basic knowledge of the Manta ecosystem. To enable customers with various technical skills to understand the use cases and be able to configure parameters, it is essential to choose an expression language that all kinds of users understand.

This section contains a list of five rules that were collected during the requirements stage. Note that those are not the only possible uses of the rules engine, and the users may find another by customizing the provided ones by parameters. What is more, by chaining particular rules so that the output of one rule is an input into another, the end-user can create a pipeline resulting in entirely new business information. Nevertheless, chaining is not the only way to combine more rules into complex ones. It may also be possible to run one rule with different arguments, e.g., two different revisions, and report the difference of both outputs. The customer may immediately see what has changed between the two revisions and whether his data lineage modification did not break any protected lineage entities.

Below there is the list of rules in arbitrary order. Every rule is explained with a description of business value. Most of the rules may offer a few modes of operation suitable for different uses. Apart from rule-specific parameters, three input parameters are mandatory for all rules:

- *Revision intervals* – Historical version of metadata.
- *Edge labels* – Edge labels with information on whether the flow traverses only direct edges or filter edges as well.
- *Ignored paths* – List of ignored resources or parts of the lineage graph because the user may want to skip some sources from rule evaluation for various reasons.

3.4.1 Centroids rule

Centroids objects are important parts of data flow with the most flow edges in the *incoming*, *outgoing*, or *both* directions. These objects may be critical points of data lineage, such as database tables containing a mapping from username to customer's name, from which many other procedures read. Therefore, the owner or administrator should guarantee that this table contains verified data. Otherwise, the incorrect values would leak into other lineage parts. Also, the user may find out which database procedures are used the most and carefully inspect them to guarantee their accuracy with no bugs.

The information about centroids objects can also be used in Manta UI. For example, during the initial lineage visualization, the centroids may be displayed in detail with unfolded columns, while non-centroids may be visualized contracted, given by end-user filters. With this information, the user can detect unused columns, which should be removed to reduce the table's overall complexity.

Also, for performance reasons, the administrator may duplicate the centroid table to allow load balancing and achieve better efficiency of reading database transactions. Load balancing is the process of redistributing the workload among more sources to improve resource utilization and job response time while also avoiding a situation where some sources are heavily loaded while others are idle or doing little work [33].

Last but not least, the output of this rule in the form of centroids objects might be a valuable input to other rules, allowing a more complex rules pipeline creating brand new use cases.

3.4.2 Restricted flows rule

The restricted flows rule detects lineage flows that are outside of a set of restrictions. A user provides a list of restrictions by himself. Each restriction entry is created by the definition of sources (e.g., a path to the particular database or the entire selected resource) and parts of the lineage that are permitted or forbidden. Afterward, if there exists any data flow from source objects outside of permitted or to the forbidden objects, the conflicting edge from allowed nodes to the forbidden objects is reported.

An example restriction input may be the following: "The data flows from the table named *empl_wages* are only processed by the *Informatica* resource and nothing else. In addition, the data from this table must not appear in the database called *public*, also stored in the *Informatica*." This case is depicted in figure 3.3, and there can be seen two conflicting edges (red colored) that the rule will report. Note that the graph is only illustrative because, in the real repository, the flow edges are not directly connecting particular table nodes but their children columns, which are not depicted in the diagram for simplicity.

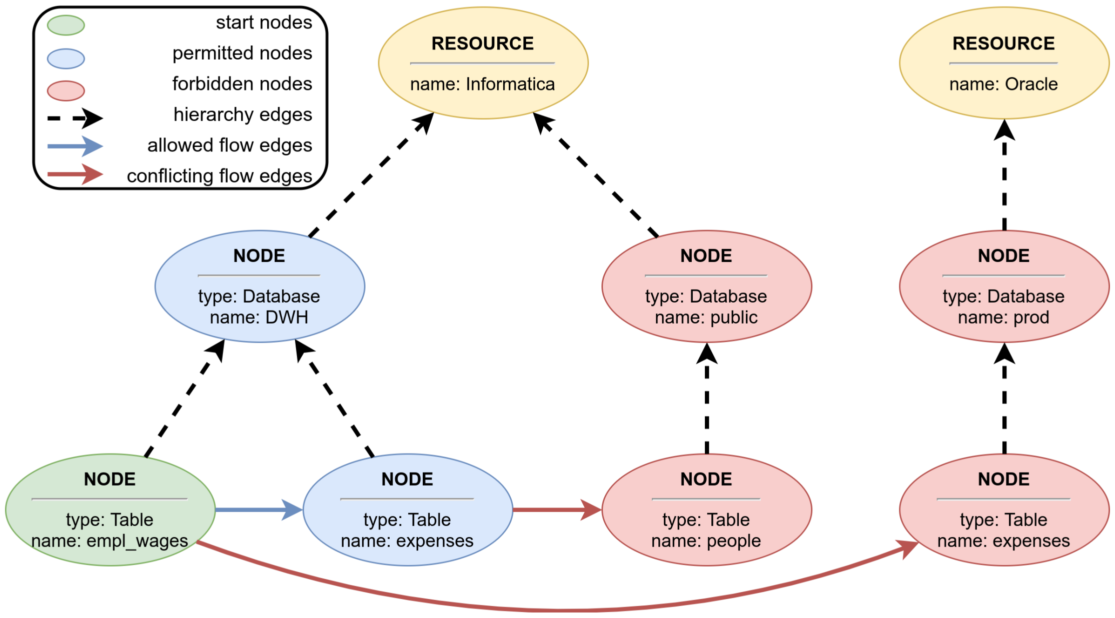


Figure 3.3. An example of the restricted flows rule entry [27]

This rule is expected to be crucial for data governance since it is easy to once configure some watched parts of the lineage and their corresponding forbidden or permitted paths. Then, the rules engine can run this rule in scheduled intervals guaranteeing up-to-date meeting of all regulatory compliances, such as GDPR.

Besides, the rule may be used to reduce dependencies between different data systems. If the resources in a large DWH are excessively interconnected, in case of fail in one system, the whole DWH might crash, resulting in a demanding recovery of all resources. Thus the user can declare, e.g., "Any data starting in the *Oracle* system must not leak outside of *Oracle* or *Oracle scripts*." If this condition is violated, the particular flow edge from *Oracle* to a different resource is reported.

The restricted flows rule should also benefit from multi-revision scenarios. The rule may run on two different revisions. The outputs are compared, and the engine detects new objects that didn't exist in the flow before but were added in a particular revision, e.g., a new file with customer information.

3.4.3 Isolated components rule

Throughout development time, it is often the case that an environment has objects in their data environment that are just not used. For instance, for various reasons, the data are only within legacy systems that are not utilized anymore. Or it may be the case that the development of a particular application was scratched, but its remnants are still within the environment.

The isolated components rule detects objects having no outside flow edges in the following directions:

- *Ingoing* direction – A component with no ingoing flow edges is expected to be an input system. However, sometimes the object without external ingoing edges is not an input system, therefore the data in this component are static. Although this object should be externally updated, it is not. Hence it should be investigated further to find the cause.
- *Outgoing* direction – A component with no outgoing flow edges may be a reporting system or a *dead* object. The term *dead* is in this context used for a component that has no outgoing flow edges and is not reporting a system. Data from *dead* objects are not populated and, hence, should not be used anymore.
- *Both* directions – A fully isolated object is completely independent, which may signal either a good storage architecture or the uselessness of a given object. If it is redundant, the user knows that he can safely remove the whole part of the system without affecting the other systems.

Let's take the graph in figure 3.3 as an example. The end-user might want to analyze isolated components of two given types, *Resource* and *Database*. The following table 3.2 depicts the expected rule result with the names of reported objects. The data lineage does not contain any completely isolated *Resource* or *Database* objects. Input resource in incoming direction is *Informatica* with the particular database called *DWH*, and the reporting resource in outgoing direction is *Oracle* with the database named *prod*. Moreover, there is also another reporting database *Informatica/public*.

	Resource type	*Database* type
Ingoing	Informatica	Informatica/DWH
Outgoing	Oracle	Informatica/public, Oracle/prod
Both	–	–

Table 3.2. The detected isolated objects from graph 3.3 with various settings of directions and types.

After running this rule, the user looks over the report and inspects the results. This rule will typically provide a list of isolated components, and the user should verify if the results are expected. If the report contains isolated objects, but the user knows that they are not isolated in reality, no matter in which direction, he knows that some part of the lineage is missing. This may happen if he uses some technology that Manta does not support, he may not have a proper license, or during the first extraction phase of the Manta scanner, some parts have not been scanned and parsed correctly. Then the user can manually add all missing relationships and data flows by himself in the custom metadata import module, which can be used to ingest any technology that is not formally supported yet. What is more, this process should be simplified in the future by a new Manta tool Custom Metadata UI, which is currently under development.

To sum up, this rule should primarily help the administrators to decide which systems can be safely shut down or simplified without affecting the others. This leads to the increase of performance of the entire environment to reduce the company expenses. What is more, the end-user may use particular isolated components found in this rule as an input into other rules and create more complex patterns such as "Detect all input database tables and files, which contains data flows into the *Oracle* resource." The multi-revision scenario may be used to list isolated objects that were not isolated in the previous revisions, which signals that the data lineage had probably been simplified. Therefore the newly isolated object is also a candidate for deletion.

3.4.4 The longest chains rule

The longest chain rule should detect the longest data flows, i.e., sequences of flow edges, in the lineage. The information about the longest chains is helpful for a few reasons. Primarily, it indicates that the flow may be too complicated, and the administrator should consider the environment simplification into smaller pieces. Typically a long chain of edges traverses many resources, and if any system fails, the entire flow is hard to fix and restart. Moreover, long flow often goes through particular nodes repeatedly, bringing attention to the entire environment's overall complicated architecture.

Note that the storage model of the Manta repository allows cycles in data flows. Because the task of finding the longest paths in a directed *cyclic* graph is an *NP-hard* problem, this rule is expected to be a problematic one. On that account, the implementation of the main rule query might require some approximation techniques for running on large datasets [34].

3.4.5 Independent flows rule

While the previous rule detects the longest flows, this rule aims to find the dependencies between particular groups of data flows. If the user understands the dataflows in his environment, he can plan logical schedules and parallelize some parts of the DWH load, achieving higher effectiveness.

Because it does not make sense to compare all possible data flows of the lineage (there are many), it is important to define what flows are supposed to be analyzed. The specification of one flow could be a single ETL job, also called workflow in this context. One job is typically a set of ordered commands and transformations, which takes data from one component (e.g., table in database) to another component and performs some data manipulation on the way. With the growing environment's size, more ETL jobs are defined. Besides, an increasing amount of data causes a more extensive execution time of every workflow. It is often the case that the jobs are unmaintained and insufficiently documented. Thus everybody is afraid to perform any workflow modification or rescheduling of those processes, even if the loading times are long and data are not coming into the target component on time [25].

Having this rule, the user may safely reschedule the independent ETL workflows to run in parallel. However, some ETL jobs may have mutual dependencies. The figure 3.4 shows two separate workflows named *ETL-job-A* and *ETL-job-B*, that both write to the same target table */db/products*. The concurrent write into one table is often not possible in many databases, e.g., because the database system locks the writing tables during transactions.

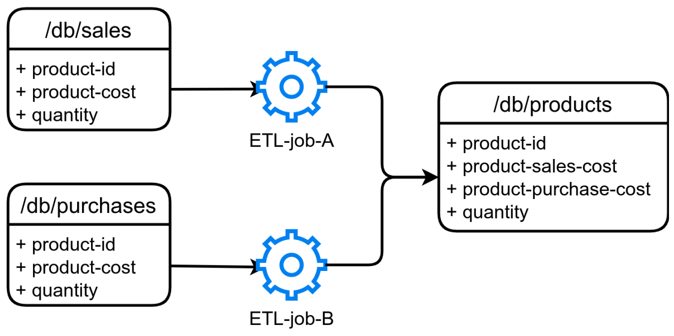


Figure 3.4. An example of two dependent workflows with the same target dependency.

Another example is a process dependency depicted in figure 3.5, where *ETL-job-A* writes to the specific table */db/products*, while *ETL-job-B* reads from the same table. Therefore, *ETL-job-B* cannot run before *ETL-job-A* is finished [25]. Knowing these dependencies, the user may safely sequentially run dependent flows while running all independent flows in parallel. Last but not least, on the independent flows, it is possible to do parallel development.

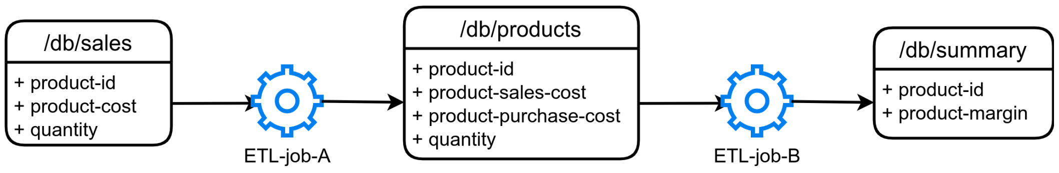


Figure 3.5. An example of dependent workflows with the process dependency.

3.4.6 Use cases summary

In this section, the author described particular rules together with their business use cases. As can be seen, there are many opportunities to use them and profit from the added value. Beyond those uses, it is believed that customers themselves could come up with ideas of new rules, which may be implemented later as the plugins into the rules engine. Besides, with the functions of rules chaining and the comparison of outputs launched with different revisions, the rules engine component's possibilities are enormous.

Chapter 4
Analysis and design

Before moving to the implementation part of the software process, it is crucial to analyze the problem with the inputs from the requirements stage. This chapter aims to explore the used technologies and development environment. At first, the selected graph database, Neo4j, is examined in detail, as it is essential to fully understand how the particular database works to profit from its features. In the following section, the author explores the currently used rules engine tools and discusses their usefulness for this project. The next part focuses on Manta's environment and the current state of the migration process to Neo4j. Finally, the author describes the architecture design of the application implemented in the practical part.

4.1 Neo4j

This rules engine project will use the Neo4j engine because it was the primary functional requirement. The first version of Neo4j was released in 2007, and over time, it has significantly grown up and became the leader in the graph databases segment [35–36]. In this project, the author uses the recent version 4.1. This section aims to provide a description of this technology and discuss the ways of querying.

4.1.1 Overview

Neo4j uses the property graph for data storage, as required by Manta's architecture discussed in section 2.3. It is an open-source tool, written in Java[1]. Nowadays, the project is already very mature, with detailed documentation and effective support. Neo4j offers two basic licenses – *Community* and *Enterprise* Editions. The Community Edition is completely free and fully functional, suitable for single-instance deployments. The Enterprise Edition extends Community Edition's functionality and includes additional features, such as a clustering architecture, online backup functionality, advanced monitoring, or more security settings [37].

Nowadays, most databases run as a server that is accessed through a client library. Therefore Neo4j can be run in a *server mode*, but also it supports the *embedded mode*. No matter what mode is used, based on general architecture choice, the way of querying and working with the database is the same.

Neo4j Server can be deployed as a standalone server with meeting all ACID properties or across multiple machines in a scalable fault-tolerant cluster for production environments [37]. Server mode compared to embedded one is easier to monitor and more robust because potential crashes in the client, e.g., unexpected garbage collector behavior, do not affect the server [38]. The communication between server and client is through an exposed REST API, thus it is independent on a particular used platform.

On the contrary, the embedded mode runs in the same process as the client application storing data. Because there is no network overhead, the latency is minimal.

[1] `https://github.com/neo4j`

Using the Core Java API, the transactional life cycle can be completely controlled, and there can be executed an arbitrary complex sequence of commands in a single transaction. However, that means that the application is fully responsible for the safe starting and closing of the database lifecycle and must deal with garbage collector actions. Note that Neo4j is not an in-memory database, although in-memory databases may also be embedded databases [2].

In addition, Neo4j contributors provide and maintains official Docker[1] images for both Community and Enterprise editions. With Docker containers, the user can create independent throw-away Neo4j instances of many versions and configurations for user-friendly testing and running of applications [39].

To enable a convenient way to approach the database, Neo4j contains drivers for the most popular programming languages owing to the Neo4j contributor community. Even though some of the drivers use the HTTP API under the hood, drivers make them available in a more convenient way than by building the traditional REST API requests. Although language drivers are intended to be used mainly by developers, with the provided demonstrative example projects on the Github repositories[2], even people with little programming skills, such as data scientists or analysts, can create simple applications on top of the Neo4j engine.

4.1.2 Querying the database

So far, we've discussed how Neo4j works with different operation modes. However, the crucial feature of databases is data querying. The following paragraphs contain a description of the ways of graph traversal in Neo4j. Neo4j has historically provided a few methods of data querying and retrieval.

Initially, Neo4j was queried by Java Traversal API. In the traversal framework, the user must create a *traversal* object that performs the traversal operation in a callback-based, lazily-executed way. This object specifies how to query a graph, where to start, and how to behave when visiting the nodes and relationships found during the traversal. This procedural approach is very powerful, but a slight configuration change requires rethinking the traversal and rebuilding the whole project. Besides, the code might quickly become unreadable, especially for non-developers [38]. Due to these reasons, since version 4.0, the traversal framework became deprecated, thus unmaintained.

On the contrary, in 2011, Neo4j introduced declarative language Cypher, which has become a preferred way to query Neo4j graphs [40]. Moreover, operations written in Cypher may be further extended by the user-defined procedures. The following code shows the same query implemented in both the traversal framework and Cypher. Without a doubt, the Cypher statement is more straightforward and better readable. Both queries should *"Find names of all actors that played in the movie with title pulp-fiction."*

```
// Cypher language

MATCH (m:MOVIE)<-[:PLAY_IN]-(a)
WHERE m.title = "pulp-fiction"
RETURN a.name
```

[1] https://hub.docker.com/
[2] https://github.com/neo4j-examples?query=movies

```
// Traversal API

TraversalDescription td = db.traversalDescription()
  .relationships(Types.PLAY_IN, Direction.INGOING)
  .evaluator(Evaluators.atDepth(1));

Node s = db.findNode(Label.label("MOVIE"), "title", "pulp-fiction");
Traverser t = td.traverse(s);

for (Path p : t) {
  Node n = p.endNode();
  System.out.println(n.getProperty("name"));
}
```

4.1.3 Cypher

Cypher is an expressive graph database query language, which is used in more database technologies apart from Neo4j. Cypher is designed to be readable and understandable not only by developers but also by less technical people, e.g., business stakeholders or analysts. Its simplicity comes from the fact that it resembles the way people intuitively describe graphs using diagrams [2]. Cypher uses *declarative* syntax that is convenient not only for essential graph operations, such as *"get the age of a person having the name Vincent Vega,"* but also for finding complicated paths or whole subgraphs. An example of a rich pattern is *"find all nodes and relationships creating triangle pattern in the graph."* The term *declarative* means that the language itself focuses on the result's aspects rather than methods or ways to get the result [38]. Owing to universality and expressiveness, it became one of the most popular graph querying languages.

4.1.4 Stored procedures

Typically there comes a time, where the developer realizes that for his specific task there is not possible to create a Cypher statement, or the query is just too complicated and highly algorithmically inefficient. Then the user may use stored procedures, also called user-defined procedures, which are invoked directly from Cypher language and provide additional functionalities. Instead of constructing long chaotic Cypher queries with too many lines, the stored procedures take arguments, perform operations on the database, and return results [39]. Stored procedures may allow a user to create entirely imperative queries if he needs to control the traversal's complete process. What is more, they are often very fast and can be further tuned to achieve a massive performance [41].

A few stored functions are bundled with the initial installation of Neo4j, but often the end-user wants to use more complex ones. On the one hand, he can create a brand new procedure by himself, programmed in Java. This is an excellent option for companies already having Java developers; therefore, they may quickly implement custom functions in their familiar environment [42]. For this, Neo4j offers a simple API. On the other hand, the user might use utility libraries providing convenient user-defined functions which aren't implemented in Neo4j yet. In both approaches, the end-user just copies the file in .jar format with functions and procedures into the Neo4j install directory, restarts the server, and quickly benefits from the new features.

One of the largest and most widely used extension libraries for Neo4j is APOC, which stands for Awesome Procedures on Cypher, and has open-source code[1]. This

[1] https://github.com/neo4j-contrib/neo4j-apoc-procedures

library provides functionalities for utilities, conversions, graph updates, natural language processing, and many more [39]. Apart from that, APOC contains functions for advanced graph querying, however, they are still not fully flexible in terms of configuration. To allow complex query traversals, APOC offers path-expand procedures, particularly popular *apoc.path.expandConfig()*. Although this traverser allows detailed configuration settings, in the implementation of the current version, the traversal of relationships based on the property value is not supported [43]. However, this is crucial functionality for Manta because most of the relationships own property values representing the revision numbers necessary for proper graph querying. Hence this powerful traverser is not an option for querying the Manta repository.

4.1.5 Platform

Neo4j aims to become a tool that people from different technical backgrounds quickly adopt. Therefore it allows diverse ways of interaction with stored data. Figure 4.1 presents the architecture of the whole Neo4j ecosystem. As can be seen, every component is designed to suit different kinds of people with various job positions [44].

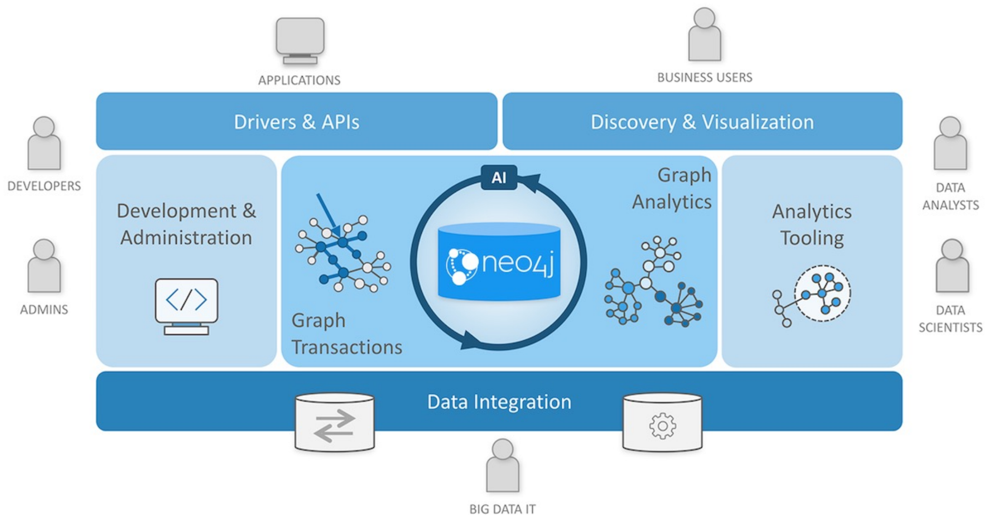


Figure 4.1. Components of the Neo4j platform [44]

Neo4j itself provides many applications or tools built on top of the Neo4 engine, and independent contributors create many more innovative integration tools, libraries, or connectors. In the following list, the author of the thesis would like to briefly discuss some of those tools that might be useful for the practical part and future development [44–45].

- **Neo4j Browser** – It is a web-based client allowing real-time interaction with the Neo4j Server database without configuring or programming anything other than Cypher. It provides a simple interface to query and view the database's data, which is useful especially for prototyping and debugging. The Cypher queries may be written directly to the shell prompt, and they are executed in real-time. The results are typically rendered either as a visual graph, a table format, or JSON, and the user may frequently switch between those representations to his needs [38]. Besides, the Browser offers a profiler tool enabling a convenient way to monitor a particular Cypher query's performance.

- **Neo4j Desktop** – Application managing local instances of Neo4j in the GUI. The user can create many independent projects and databases, configure them, extend with particular plugins or libraries, etc. Therefore for many basic tasks, there is no need to use Java or the command line.
- **Java APIs** – To work with Neo4j from Java applications, there are a few convenient APIs available on the Maven Central Repository[1]:
    - **neo4j-java-driver** – The API with the main Java driver interacting with Neo4j servers. Typically the application constructs a *driver* object on startup, which handles connection and transactions with the database, and is destroyed on application shutdown. Besides, the driver is responsible for proper authentication and security configuration [46].
    - **neo4j** – The API for Community Edition of Neo4j provides an interface for embedded servers. It is also used for the creation of custom stored procedures and functions.
    - **neo4j-enterprise** – The API for Enterprise Edition of Neo4j extending the previous *neo4j* API with additional functionalities, as discussed in section 4.1.1 [47].
    - **neo4j-harness** – The library provides an interface for testing Neo4j. It is a special variant of an embedded Neo4j server instance with the capability of adding custom procedures and extensions [48].
    - **spring-data-neo4j** – Spring Data Neo4j[2] provides easy configuration and access to Neo4j from Spring application; therefore, it uses familiar Spring concepts and annotation-based programming model [49]. Spring Data Neo4j equivalently uses template classes with the Spring Data project, allowing convenient data access for NoSQL databases [50].
- **Graph Data Science library** – The library[3] offers Cypher procedures for even more complex and customizable tasks than the APOC library. It efficiently implements parallelized versions of standard graph algorithms in areas such as pathfinding, centrality, clustering, link prediction, or similarity.

4.1.6 Storage

Firstly, the Neo4j storage model follows a few principles, which are summarized in table 4.1. Although node labels are optional, every relationship must have exactly one node type. Every node and relationship have also automatically generated IDs.

Graph entity	Compulsory	Optional
Node	• ID (automatically generated)	• Labels • Properties
Relationship	• Direction • Type (exactly one) • ID (automatically generated)	• Properties

Table 4.1. Summary of entity types and their attributes in Neo4j

[1] https://mvnrepository.com/repos/central
[2] https://spring.io/projects/spring-data-neo4j
[3] https://github.com/neo4j/graph-data-science

Graph data in Neo4j are kept in its store files, each of which serves a different purpose. Therefore there are separate files for nodes, relationships, relationship types, labels, and properties, and the data from different files are mutually referenced by ids. This storage method is also called *native* graph storage and is very efficient for graph operations [51].

Indexes help optimize the process of finding specific nodes. However, index-free adjacency, which Neo4j provides, means that each node directly references its adjacent nodes and relationships. It ensures fast graph traversal without reliance on user-defined indexes during query processing [52]. Native graph queries, such as finding the node properties or neighbors, perform at a constant rate regardless of the size of the stored data. On the contrary, non-native graph processing requires the user to create those indexes by themselves, and querying through a large number of indexes may slow down the whole execution [51].

Graph data stored on a disk is all double-linked lists. Properties are stored as a linked list of property records, each holding a key and value and pointing to the next property. Each node and relationship points to its first property record. The nodes also reference the first relationship in its relationship chain. Each relationship has pointers to its start and end nodes [9, 45].

In Neo4j, the node ids and labels are indexed automatically. However, sometimes the user wants to start the traversal in a specific node given by a pattern composed of one or more properties. This can be optimized by custom indexes defined on a particular node type and property. Neo4j allows the creation of indexes per any label and property combinations. Moreover, the user can also specify constraints ensuring property values' uniqueness, such as "assure that all nodes of *Person* label have a unique property value *national identification number*." Then, suppose the user wants to create another node satisfying the same combination of label and given property. In that case, the write operation is aborted with a warning message [2].

4.2 Rules engine tools

This section introduces the term *rules engine* and general ideas behind this concept. After getting familiar with rules engine theory, there are a few representative tools discussed. In the last part, the author considers the possibility of using any current rules engine application for this project.

4.2.1 General

A rules engine is a system used in applications to manage some of the business logic. An engine should be used in applications where the business logic changes frequently or it is required to allow end-users to create and modify the business rules parameters. Those changes should be done quickly in a runtime production environment with no source code changes so that business logic remains separated from the code [53].

Every rule consists mainly of two parts, condition and action. An action can be, e.g., sending an email to a responsible person or changing the data in a database. Firstly, the user creates a set of rules. Then the engine runs through all the rules, picks the ones for which the condition is satisfied, and then evaluates the corresponding actions. The engine can also be responsible for the rule scheduling. Then the user does not need to constantly monitor all operations in order to react to an event. Instead, a rule specified by the user is monitored and executed by the active rule engine system [54]. Article [55] proposes a few typical examples of business rules listed below:

```
IF car.owner.hasCellPhone THEN premium += 100;
IF car.model.theftRating > 4 THEN premium += 200;
IF car.owner.livesInDodgyArea AND car.model.theftRating > 2 THEN premium += 300;
```

Although the rules must be flexible enough to allow modifications and the creation of various cases, they should also be simple enough to be created and maintained without involving programmers. This also requires some convenient system for rules management. Examples of such interfaces can be pre-configured excel spreadsheet, GUI, or simple DSL. Storing rules together in one place can also serve as up-to-date documentation. Then the business analysts or non-technical stakeholders can easily read and verify a set of rules because these are the typical people responsible for defining the rules [56].

Most of the rules engine Java libraries implement the ancient JSR 94 standard published in 2005 [57]. This standard, also called Java Rule Engine API, should have solved the previous lack of standards. JSR 94 attempted to standardize the rule engine implementations, as it provides Java API for rules register and unregister, parsing, execution, result retrieval, and filtering [53]. However, some of the rules engines do not allow the creation of business logic outside the Java code, which is entirely against the initial concept of separating logic from the code.

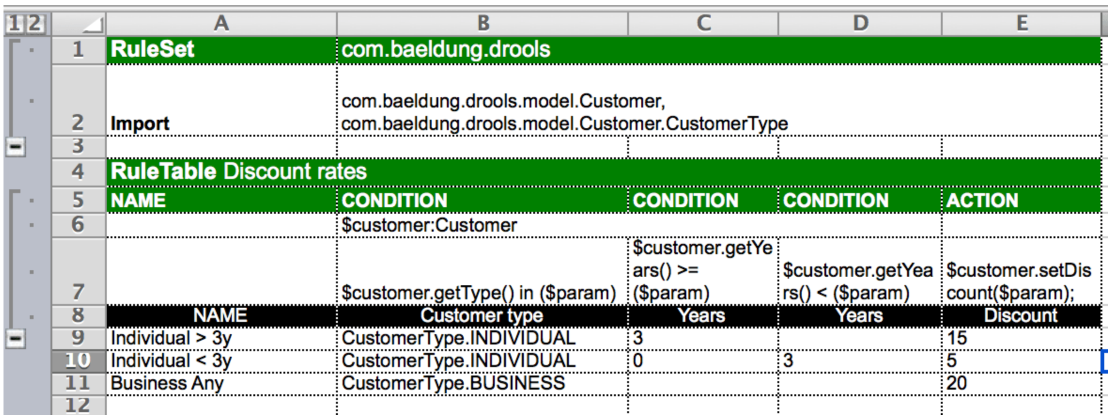
4.2.2 Representatives

This section shows a few examples of currently the most widely used open-source rules engine tools.

Drools[1] is a business rule management system, which provides a core engine and a rules management application [58]. The Drools solution's main aim is to centralize the business logic to make changes fast and cheap without the need for rule coding. The preferred way of creating rules is in their DSL .drl format. Moreover, using Excel decision trees is supported as well, which should be the simplest way of rule definition by the least technical people [59]. Figure 4.2 shows the example of an Excel spreadsheet with three implemented rules in lines 9–11. Although this approach may look like a convenient one, it isn't easy to maintain many excel spreadsheets. It is also necessary to provide a sophisticated mechanism for rule validation. For elementary rules, this can be sufficient. Still, for rules with more complex conditions, as is expected in this project, it may be almost impossible to ensure that the input data are validated and behave as desired. The solution could be to provide a locked spreadsheet with only a few writable cells so that the developer can control what parameters of rules can be edited without exposing the rules directly. However, this would cause a significant decrease in rules flexibility.

Another actively maintained rules engine is **Easy Rules**[2]. This library provides a lightweight API containing valuable abstractions to define business rules and apply them easily with Java. This may be a drawback for some cases because the business logic has to be coded in Java language [57]. However, the user can define rules in four ways – declaratively using annotations, with a fluent API, using an expression language, or loading the YAML file with expression language [60]. Although this project is maintained mainly by a single developer, it provides a smart design, documentation, and convenient API. Hence the library may serve as an inspiration for the practical part of this thesis.

[1] https://github.com/kiegroup/drools
[2] https://github.com/j-easy/easy-rules



	A	B	C	D	E
1	**RuleSet**	com.baeldung.drools			
2	**Import**	com.baeldung.drools.model.Customer, com.baeldung.drools.model.Customer.CustomerType			
3					
4	**RuleTable** Discount rates				
5	**NAME**	**CONDITION**	**CONDITION**	**CONDITION**	**ACTION**
6		$customer:Customer			
7		$customer.getType() in ($param)	$customer.getYears() >= ($param)	$customer.getYears() < ($param)	$customer.setDiscount($param);
8	NAME	Customer type	Years	Years	Discount
9	Individual > 3y	CustomerType.INDIVIDUAL	3		15
10	Individual < 3y	CustomerType.INDIVIDUAL	0	3	5
11	Business Any	CustomerType.BUSINESS			20
12					

Figure 4.2. Example of Drools rules in excel [61]

OpenL Tablets[1] is a rules engine that targets the gap between business users and development because the rules are defined in Excel files containing decision tables. However, this may be a limiting factor because rule definitions in excel files decrease particular rules' overall configuration flexibility. OpenL Tablets also provide a web interface, called WebStudio, which should enable rules management in a GUI [62]. Nevertheless, the GUI offers almost similar functionality as traditional excel files. Moreover, the entire project excessively focuses on examining tabular data, which does not provide much use for this thesis.

4.2.3 Summary

Current rules engine tools are great for using a lot of simple rules that have only a few conditions, differ in the parameters, and operate over simple tables in relational databases. They are also suitable when the rule parameters sometimes change so that people can update them easily without any code modification. Although the concept of rules engines has existed for more than 15 years, no tool has become a widely-used and became a general leader in this area. What is more, there is no standard language for writing the rules themselves. It seems that companies prefer the development of their custom tools because their requirements are too specific, and one rules engine tool cannot satisfy all their different use cases.

As ideas behind rules engines sound promising, it isn't easy to implement them in practice, as discussed at various internet sites [54–55, 63–65]. The user creating the rules should be experienced at the domain, grasp at least general knowledge about the data model, consider all edge cases, etc. Moreover, he is forced to learn a new syntax for defining particular rules.

For the reasons listed above, the author and Manta colleagues decided not to use any current rules engine tool for this project. Specific business use cases collected during the requirements stage (discussed in section 3.4) have shown to be too complex to be entirely defined by the end-user himself. To implement the requested behavior of various rules, the person creating rules must have a deep knowledge about the graph storage structure in Manta, how the traversals work, etc. Besides, he would have to permanently maintain the functionality and promptly change the business logic if the storage model changes or a new conflicting Manta functionality is added. Creating a new rules engine for this project with predefined rules should be advantageous and thus is a preferred way of implementation.

[1] `https://github.com/openl-tablets/openl-tablets`

4.3 Design

The purpose of a software design is to transform functional requirements into a form implementable using a programming language. This includes both low-level and high-level component overviews [66]. During the design process, the developer must consider many aspects, such as reusability, extensibility, robustness, security, and last but not least, security. The target is to divide software into multiple parts, which should be decoupled as much as possible to achieve *loose coupling*. This term is used for components that are mutually integrated in an almost independent manner, which results in easy testing, consistency, documentation, scalability, maintainability, and many more. One of the primary methods for describing and visualizing software architecture and design is Unified Modelling Language (UML).

4.3.1 Overview

As discovered in the previous section, no present-day rules engine tool provides sufficient functionalities for this project. Thus, it was decided to develop a custom rules engine application from scratch, suiting the project needs. Having collected enough use cases during the requirements stage justifies investing a considerable amount of time into the brand new implementation.

The rules engine application will be implemented as a new module called *rules-engine*. Beyond the necessary usage of classes declaring the current storage Manta model, the module will have no other Manta dependencies. Therefore anyone can unzip the provided *rules-engine.zip* file, load the dependencies defined in *pom.xml*, and instantly run the test scenarios. Because the ongoing migration process from Titan to Neo4j is not finished yet, the current Manta implementation, heavily depending on the Titan database, is unusable for this project.

The general business cases from requirement section 3.4 will be represented as rules configured by various input parameters. At this stage of the project, it was decided to implement five distinct rules that should target different types of users. After the rules are developed, they can be trialed with the most eager customers and verified whether the rules increase the business value.

A relevant note is that the metadata from various Manta storage resources is not fully standardized up to date. Unfortunately, this means that the semantics of various objects in different input systems are not entirely unified. For example, although many systems may have lineage systems representing one *ETL job*, this information about being an ETL job is not standardized over various systems. Although this may pose minor problems for rules creation, mainly it might confuse the end-user, who wants to list all ETL jobs but does not want to deal with a different notation of ETL jobs in all distinct systems in the user's DWH.

4.3.2 Module design

One of the main objectives of this thesis is to implement a rules engine prototype. It is not expected that the application will be production-ready, but it might demonstrate the possible benefits of the tool. Moreover, it should be developed in a way allowing quick time to a market, future scaling, and future use in production environments.

For these reasons, the major focus is put on the module core part. Beyond the small component responsible for saving data from the Manta dump to the Neo4j, the rules engine's core part is the only component interacting with the database and performing graph operations. This component contains all the business logic of the specific rules.

To achieve loose coupling and cohesion, there are only two places in the entire module from which the database is accessed and stores the logic of executed queries. These are two interfaces called *Rule* and *GraphOperationRepository*.

Two following figures describe the design of rule classes and their dependencies. To ilustrate, the *CentroidsRule* class was selected, but other rules classes are implemented in the same manner. Figure 4.3 contains the implementation of a *CentroidsRule* class colored in blue, which implements the main method *evaluate()* containing the major rule query.

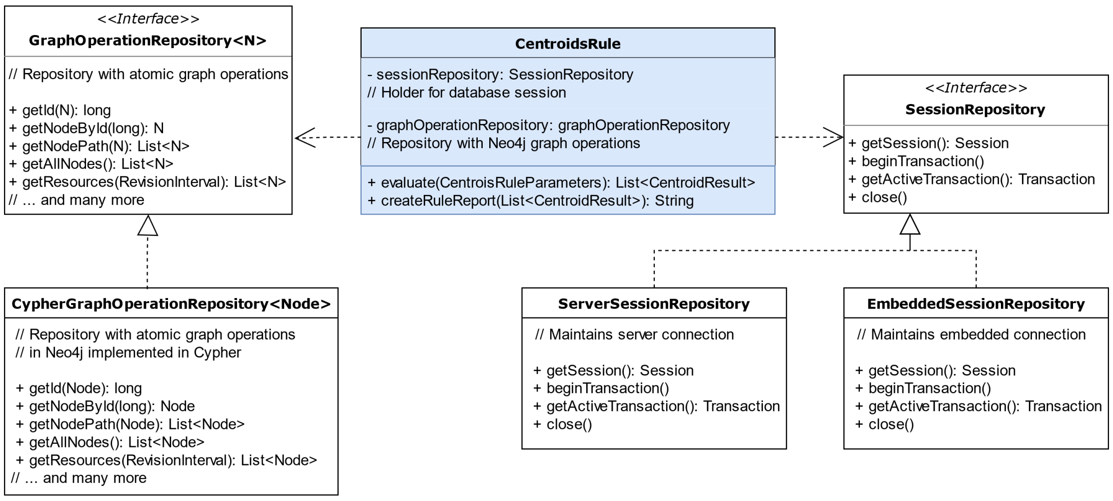

<<Interface>>
GraphOperationRepository<N>
// Repository with atomic graph operations
+ getId(N): long
+ getNodeById(long): N
+ getNodePath(N): List<N>
+ getAllNodes(): List<N>
+ getResources(RevisionInterval): List<N>
// ... and many more
CentroidsRule
- sessionRepository: SessionRepository
// Holder for database session
- graphOperationRepository: graphOperationRepository
// Repository with Neo4j graph operations
+ evaluate(CentroisRuleParameters): List<CentroidResult>
+ createRuleReport(List<CentroidResult>): String
<<Interface>>
SessionRepository
+ getSession(): Session
+ beginTransaction()
+ getActiveTransaction(): Transaction
+ close()
CypherGraphOperationRepository<Node>
// Repository with atomic graph operations
// in Neo4j implemented in Cypher
+ getId(Node): long
+ getNodeById(long): Node
+ getNodePath(Node): List<Node>
+ getAllNodes(): List<Node>
+ getResources(RevisionInterval): List<Node>
// ... and many more
ServerSessionRepository
// Maintains server connection
+ getSession(): Session
+ beginTransaction()
+ getActiveTransaction(): Transaction
+ close()
EmbeddedSessionRepository
// Maintains embedded connection
+ getSession(): Session
+ beginTransaction()
+ getActiveTransaction(): Transaction
+ close()

Figure 4.3. Simplified UML diagram of Centroid rule with the dependent repository interfaces and implementations

As all rules require some preprocessing of input parameters, the *CentroidsRule* class also uses a few methods with atomic graph operations from *GraphOperationRepository*. As can be seen, this interface is implemented by *CypherGraphOperationRepository*, which performs these operations declaratively in Cypher and returns the Neo4j node objects. Likely in the future, another *GraphOperationRepository* can be easily added, e.g., a class implementing the graph operations by stored procedures only. There can be seen dependency of the *CentroidsRule* class on the *SessionRepository* interface on the right side of the diagram. This interface is responsible for creation and handling of connection with the database. As discussed in chapter 4.1.1, the Neo4j database can be of server mode or embedded mode, implemented by *ServerSessionRepository* and *EmbeddedSessionRepository*, respectively. The *EmbeddedSessionRepository* is primarily used for unit and integration testing because it does not require a running Neo4j instance, as will be explained in-depth in chapter 6.1.2 on page 55. Note that the rules engine core performs read-only transactions because the rules do not write any data into the database. Therefore, the rules' order is arbitrary, and the engine can run them in parallel if required.

Figure 4.4 shows other dependencies of the *CentroidsRule* class. Firstly there is a Rule interface ensuring that all implemented rules follow the same structure and generally simplify testing rules. Beyond the *CentroidsRule* with the main business logic, the rule also requires classes *CentroidRuleParameters* and *CentroidResult*. The first one represents an object containing all input parameters, their default values, and the validator of inserted parameter values. The *CentroidResult* object represents one result item of the Centroid rule.

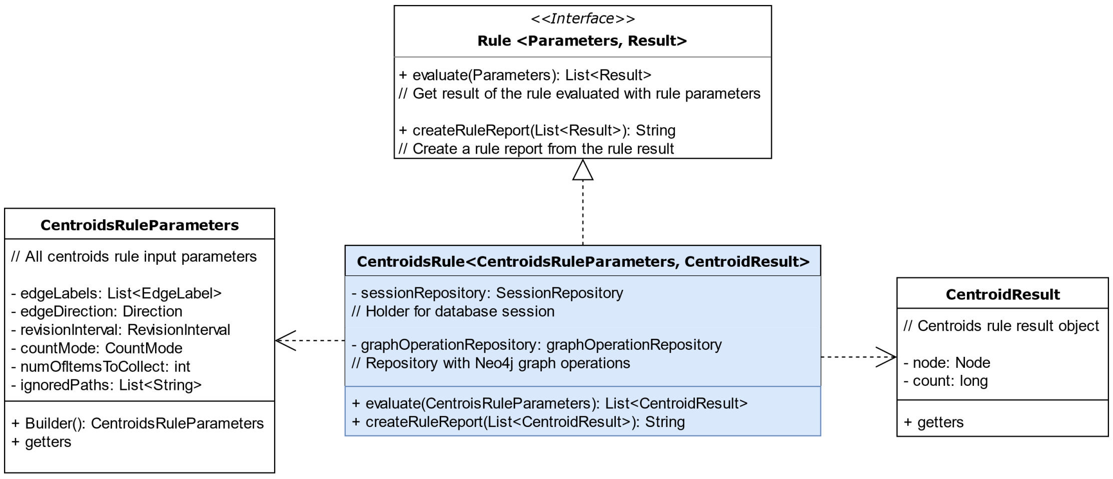

<<Interface>>
Rule <Parameters, Result>
+ evaluate(Parameters): List<Result>
// Get result of the rule evaluated with rule parameters
+ createRuleReport(List<Result>): String
// Create a rule report from the rule result
CentroidsRuleParameters
// All centroids rule input parameters
- edgeLabels: List<EdgeLabel>
- edgeDirection: Direction
- revisionInterval: RevisionInterval
- countMode: CountMode
- numOfItemsToCollect: int
- ignoredPaths: List<String>
+ Builder(): CentroidsRuleParameters
+ getters
CentroidsRule<CentroidsRuleParameters, CentroidResult>
- sessionRepository: SessionRepository
// Holder for database session
- graphOperationRepository: graphOperationRepository
// Repository with Neo4j graph operations
+ evaluate(CentroisRuleParameters): List<CentroidResult>
+ createRuleReport(List<CentroidResult>): String
CentroidResult
// Centroids rule result object
- node: Node
- count: long
+ getters

Figure 4.4. Simplified UML diagram of Centroid rule with the interface and dependent parameters and result classes

For now, it was decided not to focus on reporting methods, which were discussed as a functional requirement (viii) in section 3.2.2. Therefore, the engine's current implementation simply writes output in the form of a textual report into the logger. Then the end-user can inspect the report in the terminal or decide to automatically write the output into a file by editing the logger configuration file *log4j2.xml*. It is expected that an interface for the convenient registration of new output ways of notification and alerting will be implemented in the future.

Chapter 5
Implementation

This chapter discusses the rules engine's implementation stage based on requirements, analysis, and design information from previous chapters. At first, the development process of the entire application is described. The subsequent section focuses on implementing particular rules chosen during the requirements stage, their description, flexibility of configuration, and interpretation of rule reports.

5.1 Process

This section describes the development process of the prototype of the rules engine application. This task involves selecting tools used for implementation and testing, model preparation, connecting to the database, and selecting APIs and libraries.

The author also discusses the issues he has to deal with, mainly because the migration process of the whole Manta ecosystem to the new Neo4j has not been finished yet at the time of rules engine prototype development.

Then the overall process of creating a single rule is explained with all necessary steps leading to the successful integration of a new rule into the rules engine application.

5.1.1 General

The rules engine module is developed in Java, uses Apache Maven[1] for dependency management, and is versioned by Git[2], needed for the functional requirement (iv). The author uses Neo4j Server in Enterprise edition, in particular 4.1.6 version. However, the Community edition would also be sufficient for the prototype because no specific functionality from the Enterprise edition was required during the development process.

The author had to collaborate with other developers and participate in the concurrent migration process from Titan to Neo4j, which involved a few tasks.

Primarily, it was necessary to implement a layer maintaining connection between the application and the Neo4j instance. The application typically connects to the local server, but embedded database mode was also required for testing. To appropriately simulate behavior on data from production environments, it was necessary to implement an importer that would take a database dump from a current Manta instance and import it into Neo4j.

It was also needed to make minor adjustments regarding the storage model for indexing of selected properties. Data loaded into Neo4j are further inspected by the Neo4j Browser, which also serves as the primary runner of database queries for easy prototyping. The graph is traversed purely by Cypher queries with the occasional help of stored procedures from the APOC package.

[1] `https://maven.apache.org/`
[2] `https://git-scm.com/`

5.1.2 Graph operations

The Connector module, which is part of Manta Flow Server, provides a set of basic graph operations assessing databases, with example functions discussed in section 2.2.2, which other components may internally use from the Manta ecosystem. The Connector also offers crucial data flow graph traversal and manipulation methods with metadata storage by a combination of visitors and traversals. The traversal object declares how to traverse the whole graph, e.g., where to start (typically root node) and method to use (e.g., BFS or DFS). Subsequently, the visitor object, inspired by the visitor design pattern [67] declares what operation to execute when approaching a specified node [24]. This is an entirely different way of traversal than recommended for Neo4j. All these graph operation methods are currently working merely with the Titan database. On top of that, in an imperative way, thus the author has to implement many of those in Cypher and benefit from its declarative approach.

5.1.3 Libraries and APIs

The following list contains public libraries and APIs with their versions used in this project. A more detailed description of the libraries may be found in section 4.1.5. Note that it is crucial to use the same version of the Neo4j server and libraries, otherwise the compatibility is not guaranteed. Thus all Neo4j libraries are of versions 4.1x.

- **neo4j** (4.1) – Although the server uses Enterprise edition, this Community Edition of the library was sufficient for the purpose of this work.
- **neo4j-java-driver** (4.1)
- **neo4j-harness** (4.1) – The library has only a test scope.
- **apoc** (4.1) – APOC package contains stored functions and procedures.
- **junit**[1] (4.13) – Framework used for unit and integration testing.
- **hamcrest**[2] (2.2) – Testing framework extending junit functionalities.

5.1.4 Rules integration

The end-users will not be allowed to create entirely new rules, but they will be encouraged to use the existing ones with detailed configuration suiting their needs. However, the current set of provided implemented rules is, without doubt, not final. With the feedback from customers, the developers in Manta will implement new rules enabling other use cases. This section describes this integration process.

To create a new rule from the ground up, the developer has to:

1. Implement the **evaluation function** containing the main business logic of the rule. The main database query uses Cypher, which may also call stored procedures. Before the query runs, there are always a few steps of input preprocessing required, e.g., to find starting nodes given by full path in the database. Presently, the aim is to implement most of the logic into one or two main Cypher queries to benefit from the Neo4j potential of effective query planning. On the contrary, the previous Titan database approach was to perform a large number of small database transactions.
2. Come up with meaningful and easily adoptable **input parameters** with reasonable default values. However, one minor adjustment of a single parameter may result in a very different Cypher query.

[1] `https://junit.org/junit4/`
[2] `http://hamcrest.org/JavaHamcrest/`

3. Create an appropriate model and comprehensive textual representation of the **rule result**. End-users will typically seek different output information in various rules.

5.2 Rules

This chapter focuses on each rule's implementation details and discusses potential user adaptability, as this is the primary aspect of this rules engine project. In some cases, the rule instructions gathered in the requirements stage seemed simple and straightforward at first. However, during the implementation stage, many new problems and ambiguities emerged.

Initially, the implementation summary of each rule is described. This is followed by the tables of input and output parameters. The first table contains a name, type, and description of input parameters with allowed values to demonstrate a specific rule's flexibility. Also, output rule parameters of every detected object are shown and explained. Note that the most detailed description may be found directly in the Java documentation of the implemented code, needed for the non-functional requirement (iii). The next part presents an example of a report in the textual form. To provide a simple readability for the user, the found nodes are represented by paths and ids. After that, the author explains the rule report results and detected objects are also visualized in Manta UI. During the implementation, a lot of ideas in terms of user-friendliness and other configurations have shown up. Therefore ways of further improvements are proposed in the last enhancement part of every rule.

Every rule is implemented in a way that specification of all parameters is required. Still, the user usually might just use the default input parameter values to allow the initial running of a rule with almost no starting effort, as this is part of the non-functional requirement (i). The main goal is to have a few input parameters allowing flexibility when running the rules, but not to make them too complex, needed for the functional requirement (ii). The following table shows three input parameters that are common for all rules.

Field	Type	Description
Edge Labels	List<String>	Labels of traversed edges, typically *directFlow* or *filterFlow*
Revision Interval	<double, double>	Revision interval of traversing
Ignored Paths	List<String>	Paths to the ignored objects or groups of objects, such as tables, resources, etc. Nodes and their hierarchical subtrees will not be evaluated in the result. This parameter is needed for the functional requirement (v).

Table 5.1. Common input parameters of all rules

5.2.1 Centroids rule

The centroids rule detects objects with the most flow edges from their children in the selected direction (or in both directions) and sorts them from the largest. The objects are evaluated on the second-lowest hierarchy level, e.g., tables, files, or database procedures.

Input and output parameters

Field	Type	Description
Edge Direction	Direction	The direction of counted flow edges: • *BOTH* • *OUTGOING* • *INCOMING*
Count Mode	CountMode	For every centroid candidate, the count value is computed. This parameter specifies the conditions: • *CHILDREN_COLUMN* – For all children nodes, count the number of columns (or other types on a similar hierarchy level) the flow goes to. • *CHILDREN_TABLE* – For all children nodes, count the number of unique tables (or other types on a similar hierarchy level) the flow goes to.
Number of items to collect	integer	The number of largest centroids to return

Table 5.2. Input parameters of Centroids rule

Field	Type	Description
Count	long	Count value (number of edges to the other objects), which is used for sorting
Node	Node	Centroid node, typically table

Table 5.3. Output parameters of Centroids rule

Sample report

```
Count: 10 -- /Oracle/ORCL/DWH/PARTY (id: 5290)
Count:  9 -- /Oracle/ORCL/DWH/IMPORT_LOAN (id: 5145)
Count:  9 -- /Oracle DDL/IMPORT_BODY//<13,5>MERGE/ (id: 5165)
```

The result shows three objects that are important components of the flow based on the large number of flow edges connecting their children to other tables. Two nodes are a *table* type, and one is a type *script*. Therefore the user should ensure that data in those critical objects are always accurate and adequately maintained. In the center of figure 5.1, there can be seen the largest centroid table called *PARTY* with ten connections to other components in both directions.

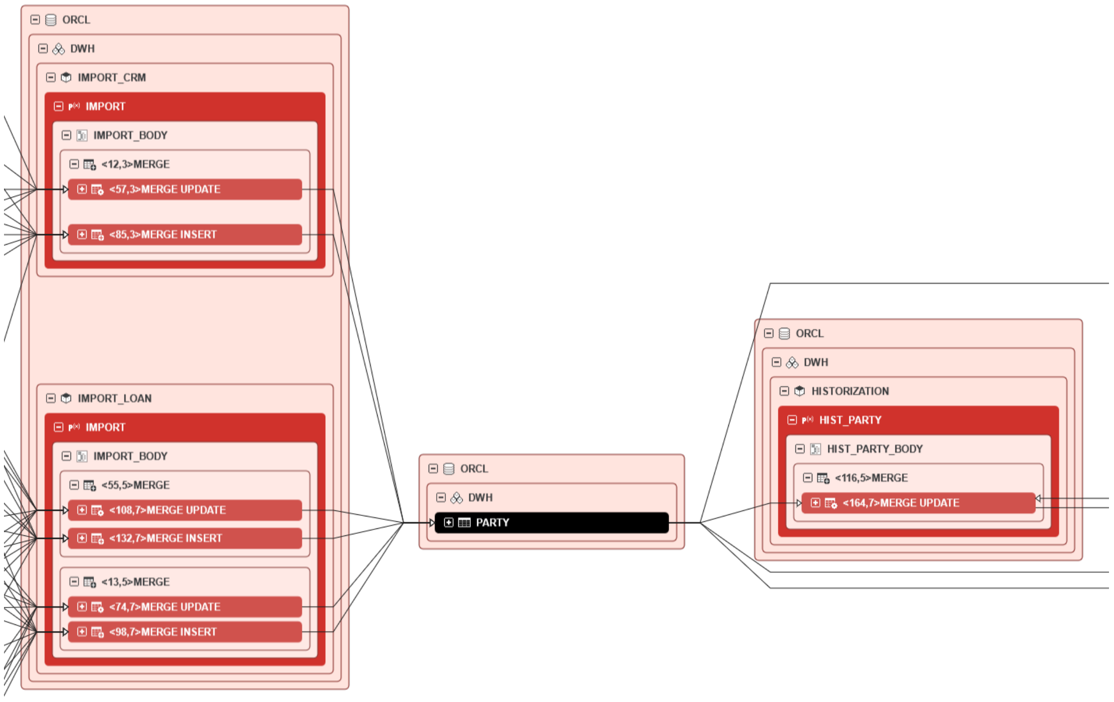


Figure 5.1. Centroids rule results in Manta UI

The validation and proposed methods for further enhancements:

- *Simplification of input field Count Mode* – Maybe there is no need to have two Count Modes. Both behave a little differently, but the end-user may not understand and notice the difference between them, so one of the modes could be removed in the future.
- *Creating a new field Minimal Value* – Instead of defining the exact number of centroids to return, the user could specify the minimal count value threshold. It could be convenient for the end-user to set up the threshold once and report if some table counts exceed the selected threshold value. Then, the user could define the rule such as *"Show me at most 15 largest centroids having more than ten edges."*
- *Addition of filtering condition functionality* – Every result could provide information about its most used children, e.g., column, with most of the flow edges, to let the user know which columns contain the most requested information.

5.2.2 Restricted flows rule

The rule detects all flows beginning in start nodes and traversing to any of rejected nodes or outside of permitted nodes. When the conflicting/rejected flow edge is detected, all start nodes that are affected are reported as well. Any *RestrInput* must have defined *startPaths*, and at least one of *rejectedPaths* and *permittedPaths* must not be empty. When the restricted edge is detected, the user can easily find the node in the Manta UI and inspect the dataflow in more detail.

Input and output parameters

Field	Type	Description
Restricted Inputs	List <RestrInput>	One RestrInput object consists of: • *startPaths* – Paths of watched objects • *rejectedPaths* – Paths to objects to where the flow shouldn't go • *permittedPaths* – Paths to the only allowed objects
Report Mode	ReportMode	• *REPORT_COLUMN* – Restricted data flows are reported on the lowest hierarchy level (e.g., columns) • *REPORT_TABLE* – Restricted data flows are reported and grouped on a second-lowest hierarchy level (e.g., tables or files)
Maximum flow depth	integer	The flow is evaluated to the maximum number of relationships from start nodes.

Table 5.4. Input parameters of Restricted flows rule

Field	Type	Description
Start Nodes	List<Node>	List of affected nodes where the restricted flow starts in.
Rejected Flow	RejectedFlow	Rejected flow object has two fields: • *startNode* – The start node of the rejected flow edge. Also, the last node of the flow that is permitted. • *endNode* – The end node of the rejected flow edge. Also, the first node of the flow that is rejected.
Description	String	Additional description

Table 5.5. Output parameters of Restricted flows rule

Sample report

```
Restricted flow:
|- From: /Hive/demo/dwh/party/gender_key (id: 48520)
|- To:   /Hive HiveQL/demo/dwh/HISTORIZATION/<1,13>INSERT/13 gender_key (id: 48611)
|- Affected start nodes:
|-- /Hive/demo/dwh/cd_gender/gender_key (id: 48468)
Restricted flow:
|- From: /Teradata/PROD_DB2/CONTRCT_SALDO_DAY_INC (id: 6335)
|- To:   /IFPC/INFA_REP/SQ_CONTRCT_SALDO_DAY_INC (id: 4142)
|- Affected start nodes:
|-- /Teradata/PROD_DB2/CONTRCT_SALDO_DAY_INC (id: 6335)
|-- /Teradata/PROD_DB3/SYSTEM1_Balances (id: 6341)
|-- /Teradata/PROD_DB3/SYSTEM1_msisdn (id: 6350)
|-- /Teradata/PROD_DB3/SYSTEM2_Contracts (id: 6359)
|-- /Teradata/PROD_DB4/OPER_CONTRCT_KEY (id: 6372)
```

The report shows two rejected flow edges that were found. The first case is depicted in figure 5.2. The user configured to find any flow starting in the table */Hive/demo/dwh/cd_gender* and going to any transformation from */Hive HiveQL/demo/dwh/HISTORIZATION/*. Therefore, the rule detected the flow edge between columns */Hive/demo/dwh/party/gender_key* and */Hive HiveQL/demo/dwh/HISTORIZATION/<1,13>INSERT/13 gender_key*, colored by blue and red colors, respectively. Also, the rule notified the user that this rejected relationship influenced the start node in */Hive/demo/dwh/party/gender_key*, highlighted by the green color.

The second detected flow edge is between *Teradata* and *IFPC* resources, with the complete list of affected starting tables.

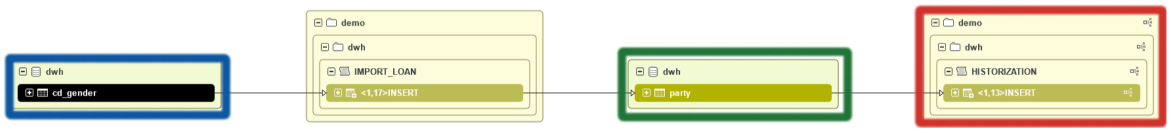


Figure 5.2. Restricted flows rule results in Manta UI

With this information, the end-user may quickly find the conflicting data flows and either fix them or make them ignored for further rule evaluation rounds. However, he is not forced to visually inspect the entire data lineage, trying to find any undesired breaches from one system to another. Once he sets up his set of rules, he can be notified about any lineage changes resulting in unintentional data leaks.

The proposed method for further enhancements:

- *Simplification of input field Restricted Input* – Before initial running the rule, the user must provide restrictive conditions as the parameter Restricted Input, which may not be easy for the average user. It would be great to give the users some mechanism that would interactively help with the configuration; however, the prototype will not implement this.

5.2.3 Isolated components rule

Isolated components are objects at various levels of hierarchy, e.g., tables, databases, or directories. Isolated object's subtree nodes may have data flows to other nodes in the current subtree but no ingoing or outgoing flow edges (with direction specified by the *Isolated Mode* parameter) to or from nodes outside the currently isolated subtree. So, e.g., an isolated object of table type is such a table that may have flow between its own columns but does not have any flows from or to columns in other tables.

Input and output parameters

Field	Type	Description
Isolated Mode	isolatedMode	The direction of isolation configuration: • *NO_INGOING* – Object is isolated if there does not exist any incoming flow to nodes in its subtree. • *NO_OUTGOING* – Object is isolated if there does not exist any outgoing flow from nodes in its subtree. • *COMPLETE* – Object is isolated if there does not exist any incoming or outgoing flow to and from nodes in its subtree.
Node Types	List<String>	List of node types that are isolated object candidates. Example values are *table*, *database*, or *server*.

Table 5.6. Input parameters of Isolated components rule

Field	Type	Description
Node	Node	The isolated node
Type	Node Type	The node type of isolated node

Table 5.7. Output parameters of Isolated components rule

Sample report

```
Number of isolated objects: 3
-- /Teradata/DBC (type: Database, id: 6161)
-- /Netezza/xnz/DEMO (type: Database, id: 4934)
-- /Netezza/xnz (type: Server, id: 49478)
```

The rule was evaluated with the *NO_INGOING* mode option, so the report shows two databases and one server, for which there doesn't exist any incoming flow from other sources. That means that those all are input systems. Figure 5.3 shows the database named *DBC* highlighted by black color. This database consists only of three tables called *ROLEMEMBERSV*, *ALLROLERIGHTSV*, and *TABLESV*. Although these tables have many outgoing flows to the other database (called *Manta* in this case), there are no ingoing flow edges to those tables from other databases; thus, the database *DBC* is reported as isolated in the ingoing direction.

When visually inspecting the second database called *DEMO*, the user can see that no flow edges are going in or out of this database, as illustrated in figure 5.4. Therefore, this database is reported as completely isolated, thus the user may consider shutting down this database if it is really not used.

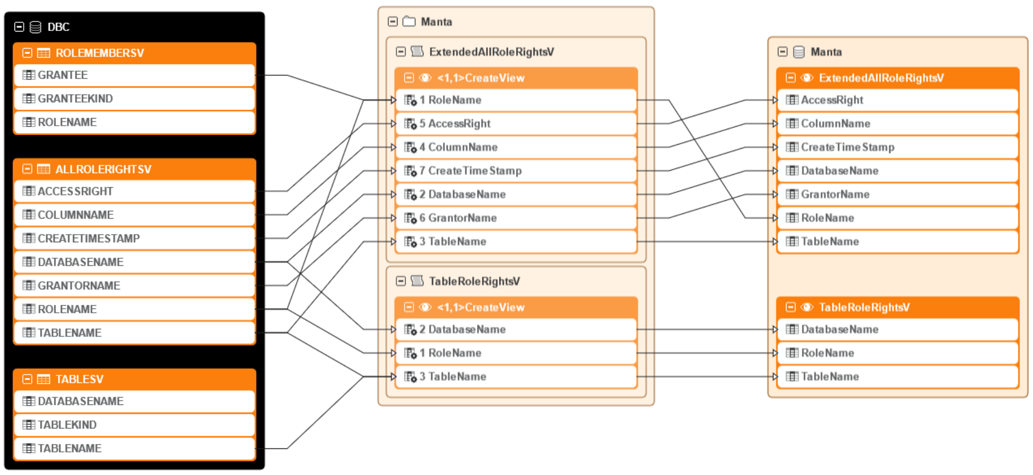


Figure 5.3. Isolated components rule results in Manta UI with *Isolated Mode = NO_INGOING configuration*

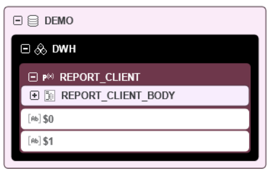


Figure 5.4. Isolated components rule results in Manta UI with *Isolated Mode = COMPLETE configuration*

The proposed method for further enhancements:

- *Create a functionality of groups of isolated objects* – Instead of reporting only particular isolated components, the rule could report entire groups of isolated objects. The input parameter would be the maximal size of the group. Then the rule could detect all islands of objects with a size smaller than the given threshold.

5.2.4 The longest chains rule

Report longest chains, i.e., sequences of relationships in the dataflow, in the graph and sort them by a length in descending order. Skip chains that are subchains, i.e., parts of longer ones, because it is useless to report the longest chain and it's parts separately. The user can define the number of longest chains to report and their minimal length, condition such as *"do not report the chains having length lower than eight."*

Input and output parameters

Field	Type	Description
Resource Mode	ResourceMode	Finding of longest chains in the entire graph or limited by boundaries of resources. • *MULTI_RESOURCE* – Detect longest chains in the entire graph. The detected chains may traverse multiple resources. • *SINGLE_RESOURCE* – All nodes of the detected chain are located in the same resource, although the flow may further continue to (or start in) other resources.
Cycle Mode	CycleMode	When detecting a cycle in the graph, this parameter affects if the cycle should be traversed or avoided. • *CYCLE_TRAVERSE* – Traverse the found cycle and add the cycle flow edges to the total flow length. That means that every node in the path may be visited more than once. However, every edge is traversed just once. • *CYCLE_AVOID* – Avoid the found cycle such that the resulting flow traverses every node on the path only once.
Minimal length	integer	Minimal length of the detected chains, a minimal number of flow edges during traversal.
Number of items to collect	integer	The number of longest chains to report.

Table 5.8. Input parameters of The longest chains rule

Field	Type	Description
Length	integer	Length of the found chain, number of traversed flow relationships
Traversed nodes	List<Node>	List of chain nodes in the traversed order

Table 5.9. Output parameters of The longest chains rule

Sample report

```
Length: 16
Nodes in the chain:
|-- /StreamSets/test/test0345/Dev Data Generator 1//address (id: 20483)
|-- /StreamSets/test/test0345/Schema Generator 1//address (id: 20476)
|-- /StreamSets/test/test0345/Field Splitter 1//address (id: 20469)
|-- /StreamSets/test/test0345/Field Splitter 1//address1 (id: 20467)
|-- /StreamSets/test/test0345/Field Replacer 1//address1 (id: 20456)
|-- /StreamSets/test/test0345/Field Masker 1//address1 (id: 20415)
|-- /StreamSets/test/test0345/Field Hasher 1//address1 (id: 20403)
|-- /StreamSets/test/test0345/Field Hasher 1//hashed_address (id: 20408)
|-- /StreamSets/test/test0345/Field Hasher 1//newtesthashentirefield (id: 20405)
|-- /StreamSets/test/test0345/Eval 1/INPUT//newtesthashentirefield (id: 20393)
|-- /StreamSets/test/test0345/Eval 1/EXPRES 1//newtesthashentirefield (id: 20371)
|-- /StreamSets/test/test0345/Eval 2/INPUT//newtesthashentirefield (id: 20348)
|-- /StreamSets/test/test0345/Eval 2/EXPRES 1//newtesthashentirefield (id: 20325)
|-- /StreamSets/test/test0345/Renamer 2/INPUT//newtesthashentirefield (id: 20254)
|-- /StreamSets/test/test0345/Renamer 2/OUTPUT //newtesthashentirefield (id: 20231)
|-- /StreamSets/test/test0345/Type Converter 1//newtesthashentirefield (id: 20207)
|-- /StreamSets/test/test0345/Trash 1//newtesthashentirefield (id: 20184)
```

The rule reported the single longest chain in the graph. Figure 5.5 shows the starting part of the whole workflow with a total length of 16 flow edges. The starting stage called *Dev Data Generator 1* can be seen on the left side, and the flow further continues through many transforming filters into the final stage called *newtesthashentirefield*.

With this information, the user can consider simplifying the whole workflow as it looks too complex with many redundant steps.

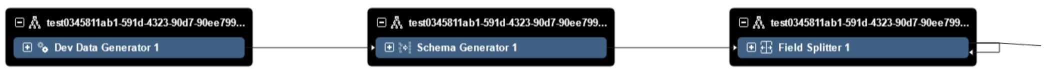

test0345811ab1-591d-4323-90d7-90ee799...
Dev Data Generator 1
test0345811ab1-591d-4323-90d7-90ee799...
Schema Generator 1
test0345811ab1-591d-4323-90d7-90ee799...
Field Splitter 1

Figure 5.5. Starting part of the longest chains rule result in Manta UI

The validation and proposed methods for further enhancements:

- *Removal of the field Resource Mode* – The parameter *Resource Mode = SINGLE_RESOURCE* is used to optionally limit the chains by the boundaries of the resources, with the idea of increasing overall performance. That means that the flow edges between various resources are considered non-existing. However, it didn't bring a significant improvement in query performance but only created new implementation problems.
- *Removal of the field Cycle Mode* – Currently, the input parameter *Cycle Mode* determines whether the cycles should be traversed or avoided. If the cycles may be traversed (*CYCLE_TRAVERSE* mode), the report contains many almost identical chains traversing the same nodes but just in a different order, so the report becomes confusing.

  Moreover, the *Cycle Mode = CYCLE_AVOID* uses the APOC package's stored function that ensures avoidance of cycles. With rewriting this rule completely into custom procedure, this dependency could also be removed.

5.2.5 Independent flows rule

The independent flows rule detects ETL jobs with required dependencies. The rule can find pairs of jobs mutually dependent by certain conditions, specified by particular use cases, but also it can list all jobs that are entirely independent to others.

Input and output parameters

Field	Type	Description
Dependency Mode	Dependency Mode	Mode of dependency configuration between particular ETL jobs: • *WRITE_DEPENDENCY* – The jobs are reported if they write to the same target object. • *READ_DEPENDENCY* – The jobs are reported if they read from the same source object. • *PROCESS_DEPENDENCY* – The jobs are reported if one writes to the target object from which the other reads. • *READ_DEPENDENCY* – Evaluate all three previous dependencies and combines results. • *PROCESS_DEPENDENCY* – The jobs are reported if they are entirely independent of any others.
Job Node Types	List<String>	List of node types representing a single ETL job object. Example values are *IFPC_WORKFLOW*, *TALEND_JOB*, or *SSIS_DATA_FLOW_TASK*.
Dependency hierarchy level	integer	Defines at what hierarchy level the dependence is evaluated on. Typical values are: • *0* – column, etc. • *1* – table, file, etc. • *2* – database, schema, directory, etc.

Table 5.10. Input parameters of Independent flows rule

Field	Type	Description
Dependency Mode	Dependency Mode	Mode of dependency between particular ETL jobs
Start job node	Node	The first node of a given dependency
Dependent job nodes	List<Node>	List of dependent ETL jobs influenced by the *start job node*. It may be empty in case of independence of ETL job.

Table 5.11. Output parameters of Independent flows rule

Sample report

```
NO_DEPENDENCY
|- Job: /Talend/talend/Daily/LoadingWF_1 (id: 68302)
NO_DEPENDENCY
|- Job: /Talend/talend/Daily/LoadingWF_2 (id: 68343)
PROCESS_DEPENDENCY
|- Job: /SSIS/Party/DWH_Load/DWH_Load/Import LOAN (id: 50297)
|-- Dependent job: /SSIS/Party/DWH_Load/DWH_Load/Stage to Core (id: 50297)
PROCESS_DEPENDENCY
|- Job: /SSIS/Party/DWH_Load/DWH_Load/Import CRM (id: 50262)
|-- Dependent job: /SSIS/Party/DWH_Load/DWH_Load/Stage to Core (id: 50262)
```

This report presents four dependency records of ETL jobs. The first two jobs called *LoadingWF_1* and *LoadingWF_2* are entirely independent of any other jobs, thus they can be scheduled in parallel.

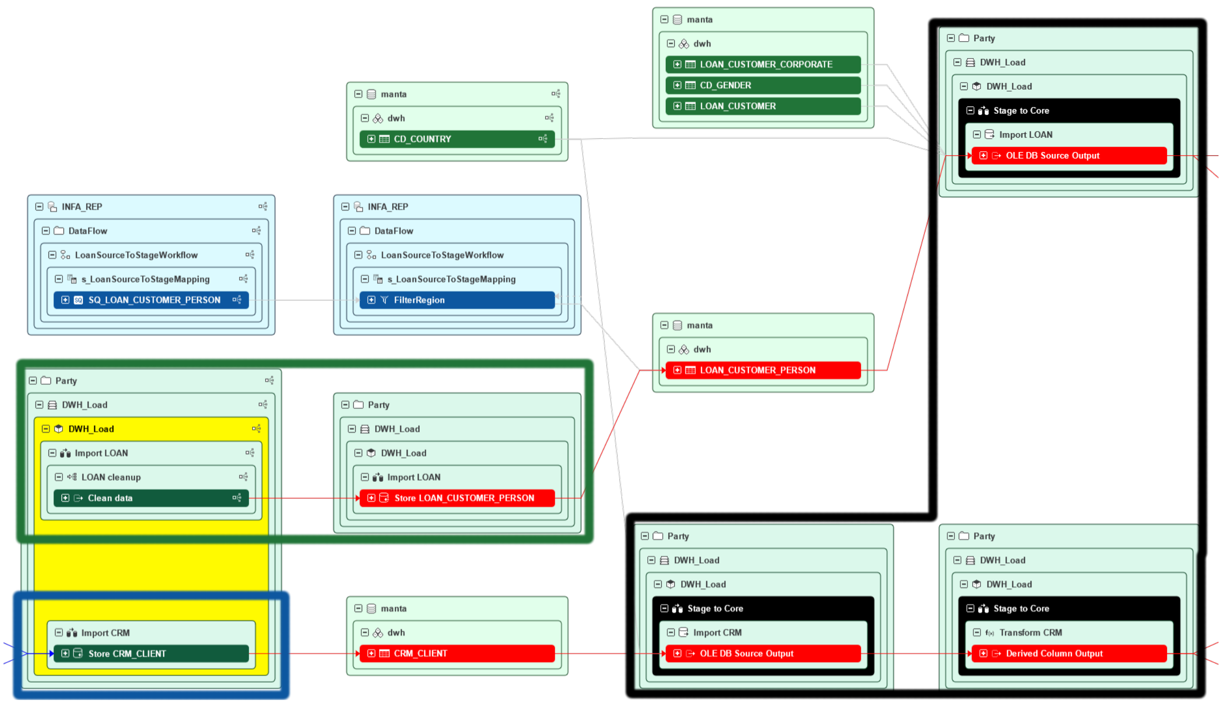


Figure 5.6. Independents flows rule results in Manta UI

Figure 5.6 illustrates two pairs of detected ETL jobs with mutual process dependencies in a small part of an extensive data lineage. The green and blue rectangles highlight the final tasks of start jobs, the directly affected ETL process is in the black-shaped area, and red arrows indicate data flows between those jobs.

It can be seen that the last task of the first green workflow (called *Import LOAN*) writes into the target table *LOAN_CUSTOMER_PERSON* in the center of the diagram. From there, the following job called *Stage to Core* reads the data and processes them further. Therefore, this pair of two ETL jobs is reported as a process dependency, so the *Stage to Core* job must be run after the previous *Import LOAN* workflow is already finished.

The same goes for the second source job called *Import CRM*, which is highlighted in blue color. In the last step, data are written into the *CRM_CLIENT* table at the bottom, from where the *Stage to Core* job reads. So this pair of workflows is reported as well, and the user is informed that before running the *Stage to Core* job, he has to schedule workflows *Import LOAN* and *Import CRM* first.

This example demonstrates that even with Manta UI, it may be difficult to interpret the dependencies between all ETL jobs correctly, especially for complex lineages with lots of workflows.

The proposed method for further enhancements:

- *Creation of metadata standardization* – Currently, the user has two options to define a single ETL job. Either he manually specifies node type (or list of node types) representing a single job, or he uses the default set of node types. However, every technology is different, and in the current storage structure, it is not possible to easily define *"list all ETL jobs from any technology"* because the information about being an ETL job is not unified through different technologies. The metadata standardization process would allow the setting of common property values for various node types from diverse resources with the same semantics functions.

5.3 Summary

This chapter has described the entire development process of the rules engine. In the first part, the author presented tools and libraries which have been used for implementation. All steps leading to the creation and integration of a new rule were discussed as well.

It was shown that the results of particular rules might provide a lot of added value for users. Having those rules, the user can automatically scan his lineage by the implemented rules for desired patterns and then inspect those in the Manta UI, which should decrease the total time of manual lineage inspection and therefore dramatically step up the efficiency of people involved.

The rules were implemented in a way that customers should run them with almost no effort from the start. However, every rule provides a few configurable options, which may be helpful for more advanced customers. By selecting various combinations of input parameters, the user may create more complex rules scenarios suited to their own use cases. This is needed by the non-functional requirement (ii) and should generally increase the whole Manta platform's capability.

Chapter 6
Evaluation

This chapter aims to describe the evaluation process of the project. The first section explains the importance of software testing and focuses on the testing methods used throughout the development. The main emphasis was put on the performance testing, particularly measuring of database hits of database queries. Therefore the following section provides a detailed performance analysis of the specific implemented rules and proposes future improvements of efficiency database queries. Finally, the overall patterns for creating effective Cypher queries and their suitability for the current storage model are discussed.

6.1 Testing

Even though the validation of rule results for customers was evaluated in the previous chapter, it is also important to test the software from the developer's point of view, needed for the functional requirement (iii). Testing is a crucial part of a software process for many reasons. Not only it helps to verify that the application behaves correctly, but it also serves as up-to-date documentation, enhances software design, and simplifies future refactoring. Moreover, having a complete test set also speeds up the onboarding process of new team members. Having enough automatic tests encourages newcomers not to be scared of changing anything in the current implementation because they can be sure that the good test suite will let them know almost immediately about the broken functionality in case of any mistake.

Software testing may be categorized into many types and approaches. The next sections explain the main methods that were used throughout the entire development process.

6.1.1 Unit tests

Unit tests are essential for implementing any software project. A unit test typically tests a single method or functionality of one class. Unit tests help developers implement new functions, quickly test them for positive and negative cases and ensure the desired behavior in potential future code changes.

During this project's development process, the author had employed the *test-driven development* (TDD) approach, primarily for the implementation of single rules. This paradigm is based on the idea of the creation of test cases before the actual implementation of particular functionality. A key benefit is that the developer focuses on essential requirements rather than coding the functions which are actually not needed. The following list shows the sequence of steps of the TDD cycle for implementation of a new feature [68]:

(i) Create one or more tests, typically unit or integration tests.
(ii) Run all tests and let them fail.
(iii) Write elementary code which is required to pass the tests. During this stage, the developer should not write more code than needed for tested functionality.

(iv) Pass the tests and possibly refactor the application if needed while still passing all tests.
(v) Add new tests and repeat all steps, which leads to the accumulation of tests over time.

6.1.2 Integration tests

Integration tests aggregate the individual units and verify the functionality of whole groups. This ensures that all small methods, which were already tested by unit tests, work correctly when connected to each other and run together.

In this work, integration testing was primarily used for the validation of the behavior of particular rules. To achieve that, the author had to create a few testing graphs in Neo4j. The graphs use the simplified structure of the Manta repository and aim to be small enough for readability but also complex to test various configurations of different rules.

The following code shows an example of a simple test method using the embedded server mode provided by the neo4j-harness library, which is a convenient way for testing purposes. By using an in-memory database, there is no need to clear up store files on disk after each test [2].

```
import org.junit.jupiter.api.Test;
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.harness.Neo4j;
import org.neo4j.harness.Neo4jBuilders;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class TestSingle {
  @Test
  public void testEmbeddedServer(){
    // given
    final Neo4j neo4j = Neo4jBuilders.newInProcessBuilder()
          .withDisabledServer()
          .withFixture(
            "CREATE (p1:Person)-[:knows]->(p2:Person)-[:knows]->(p3:Person)")
          .build();
    final Driver driver = GraphDatabase.driver(neo4j.boltURI(), AuthTokens.none());
    // when
    final String query = "MATCH (p:Person) RETURN count(n)";
    final int numOfPersons = driver.session().beginTransaction()
           .run("MATCH (p:Person) RETURN count(p) AS numOfPersons")
           .single()
           .get("numOfPersons")
           .asInt();
    // then
    assertEquals(3, numOfPersons);
  }
}
```

When combining unit and integration tests, there are altogether 72 test methods covering 88 % of lines of the classes with database operations. Because it is likely that other developers will completely redesign the dump importer and session services with different particular functionalities, these services are currently missing any tests.

6.1.3 Performance tests

Unlike unit and integration tests, performance tests are part of non-functional software validation, focusing on how the application behaves under a particular load. Queries that work fast against a small dataset may not have to scale well for large repositories.

Performance tests should give developers the diagnostic information they need to eliminate system bottlenecks [69]. Also, it is important to run those tests frequently during the development because we want to avoid a situation in which a lot of time is invested into adjustments of a particular query, which is too computationally expensive for large graphs, and therefore useless.

For this project's purpose, the author uses a database loaded with an amount of data similar to average customer graphs. This should simulate behavior in expected production environments. For in-depth performance testing, ideally, the application would be tested with different configuration settings of a particular Neo4j server instance, loaded with an exceptionally high number of database elements, modified heap size, etc. However, these tasks are beyond this thesis's scope, as the project's primary aim is to verify rule validity and behavior in typical customer environments. The author explains the performance testing in detail in the following section because it was needed for the non-functional requirement (iv).

6.2 Rules

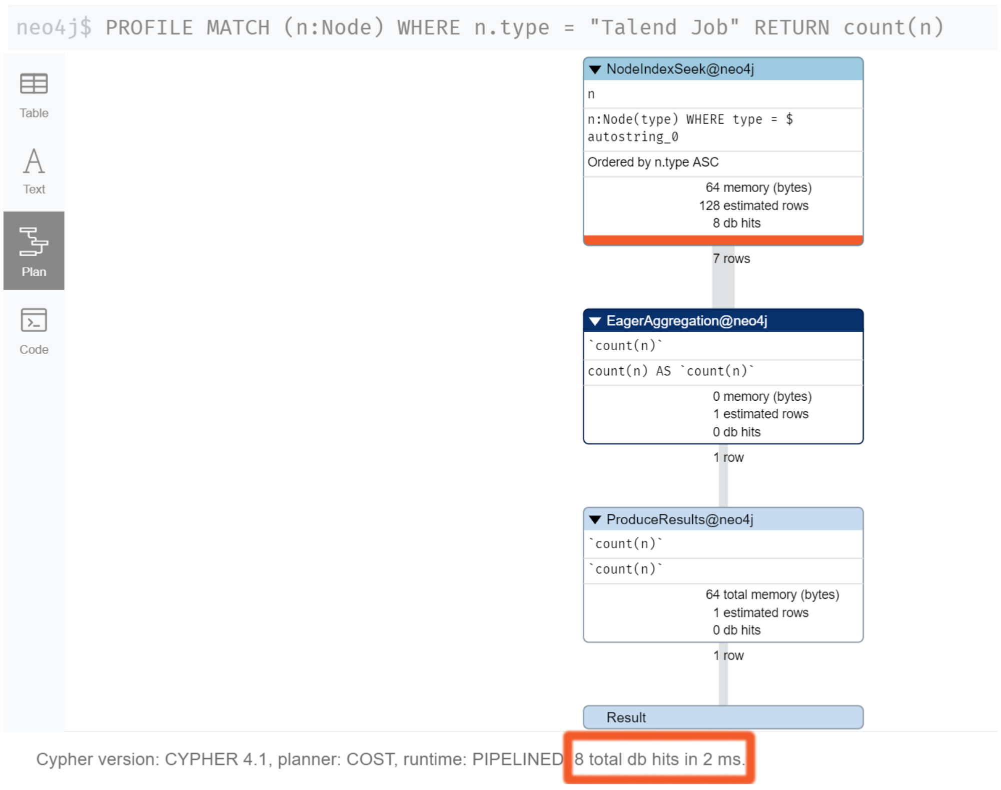
This section aims to present the main Cypher queries of each rule, discuss performance, and propose methods for more efficient evaluation. Firstly the metrics of performance testing are summarized, followed by an explanation of the testing environment. Next, a few frequently occurring patterns are illustrated because they can be observed in many particular database queries. There are also separate sections explaining every rule and proposals for query or model enhancements, leading to the potential increase in evaluation efficiency.

6.2.1 General

In this work, there are two matrices used for the rules' performance testing – time and number of database hits. A database hit, further referenced as *db hit*, is an abstract working unit of the Neo4j engine. Most of the typical database operations, such as finding a node or traversal of neighboring edges, trigger database hits [44]. An exemplary query searching for a node based on a supplied id triggers one hit. However, searching for a node referenced by an unindexed property value requires an equal number of hits with nodes in the database. The reason is that the engine must retrieve all nodes and evaluate their property values one by one.

With the *PROFILE* command provided by Neo4j Browser, the user can observe the total number of query database hits together with the map of executed operations and their computational expensiveness in terms of database hits and used memory. Figure 6.1 shows the profiling interface with a highlighted total number of db hits. This example triggers only eight database hits because the query performs a simple *MATCH* operation given by the indexed property matching eight graph nodes. Although the resulting map may appear confusing for an inexperienced user, it gives the basic idea of the most demanding operations of a given query. Some operations, e.g., aggregating functions such as *COUNT*, *DISTINCT*, or *ORDER BY*, do not generate any hits but may use a significant amount of computer memory.

During the work on the thesis, it was realized that the validation of rules based on the running time should not be the critical evaluation aspect for this project. Time performance is influenced by various conditions, such as the warmup state of the database instance, actual cache volume, server settings regarding cache, memory allocation, query replan intervals, rule configuration, and many more.



```
neo4j$ PROFILE MATCH (n:Node) WHERE n.type = "Talend Job" RETURN count(n)
```

```
NodeIndexSeek@neo4j
n
n:Node(type) WHERE type = $
autostring_0
Ordered by n.type ASC
64 memory (bytes)
128 estimated rows
8 db hits
7 rows

EagerAggregation@neo4j
`count(n)`
count(n) AS `count(n)`
0 memory (bytes)
1 estimated rows
0 db hits
1 row

ProduceResults@neo4j
`count(n)`
`count(n)`
64 total memory (bytes)
1 estimated rows
0 db hits
1 row

Result

Cypher version: CYPHER 4.1, planner: COST, runtime: PIPELINED. 8 total db hits in 2 ms.
```

Figure 6.1. Profiler tool in Neo4j Browser

However, the purpose of this work is not to create exceptionally optimized queries but to provide rule queries that finish in a reasonable time, i.e., units of seconds on an average production graph, and do not grow into an enormous complexity. Therefore, the measured time intervals may be used for a brief comparison of various rules against each other in terms of general algorithmic complexity, but the time information is not much informative by itself under given conditions. Nevertheless, if the rules will be evaluated by end-users and proved valuable, the performance of queries will be inspected in more-depth and further optimized.

Entity	Total number (-)
Node	43471
Edge	78550

Table 6.1. Number of nodes and edges in the testing graph

The performance was evaluated on the testing graph created from the testing database dump from Manta, thus the graph structure is very similar to production environments. Table 6.1 shows the graph size in terms of the number of particular graph entities. Table 6.2 summarizes the average duration of the primary database queries of all rules. Every query was evaluated on a freshly started Neo4j server instance, with no warm-up queries, averaged by ten performed measurements. Although the number of database hits may be dramatically diverse among different rules, the average query times of particular rules do not show such variance. The cause is probably the required internal warm-up time and initial precaching. For more comparable results, the author suggests using larger testing graphs and perform systematic warm-up database queries. Moreover, The longest chains rule has no recorded time because, for unknown reasons,

Rule	Db hits (thousand)	Time (ms)	Conditions
Centroids	1439	928	Count Mode = CHILDREN_TABLE, Direction = BOTH
Centroids	1352	1030	Count Mode = CHILDREN_COLUMN, Direction = BOTH
Restricted flows	0.2	638	Not informative value, because it is completely determined by the selected input restrictions
Isolated components	75	728	Isolated Mode = NO_INGOING, two node types with 23 candidates for isolated components
The longest chains	2669	-	Resource Mode = MULTI_RESOURCE, Cycle Mode = CYCLE_AVOID
Independent flows	11	844	Dependency Mode = WRITE DEPENDENCY, one type with 7 starting nodes
Independent flows	26	945	Dependency Mode = NO DEPENDENCY, one type with 7 starting nodes

Table 6.2. Comparison of database hits and average duration time of the main database queries of every rule.

the query sometimes did not finish at all and had to be run repeatedly to successfully complete.

This section also presents the code of main Cypher queries of single rules. The most detailed description of queries may be found in the Java documentation of implemented code. Nevertheless, a reader should be aware of a few repetitive patterns in these queries to understand the semantics.

Firstly, the operator *$* represents a place in the query for the input parameter inserted into a given place of the query dynamically. This syntax should be heavily used for two reasons. Firstly, it is used to write readable and maintainable code. The second purpose is to boost the performance of queries that are executed many times. The engine builds one execution plan, cache it, and uses the plan repeatedly with different parameters. Therefore, there is no need to create a new execution plan for separate queries [38].

In the following query that returns all nodes in a graph that are not present in the input list of ignored ids, the *$ignoredIds* holder expects an input parameter called *ignoredIds* (list of id numbers) injected from the application into that particular place of a query.

```
MATCH (n)
WHERE NOT id(n) IN $ignoredIds
RETURN n
```

However, only a few basic types of input parameters and composites, in particular lists and maps, are currently supported in Cypher [47]. Moreover, it is not possible to parse input parameters at an arbitrary place of the query string. Thus for some particular cases, the query was customized by a basic Java method *String.format()* by using the operators *%s* and *%d*. This method is also used to select edge directions by injecting symbols < and > next to the relationship operator. The following example returns the start node's neighbors in the *INCOMING*, *OUTGOING*, or *BOTH* directions, implemented in a single query and configured only by the input parameter direction.

```
String.format(
  "MATCH (s)%s-[r]-%s(e)
  WHERE id(s) = $startId
  RETURN e",
  Direction.INCOMING.equals(direction) ? "<" : "",
  Direction.OUTGOING.equals(direction) ? ">" : ""
)
```

Secondly, because in Manta storage, the traversal algorithm unconditionally depends on the relationships properties, the following code block ensures that the path traversal is performed only in a given revision interval and specified relationship types (typically either *DIRECT_FLOW* or both *DIRECT_FLOW* and *FILTER_FLOW*). This sample query will return all nodes achievable from the start nodes parametrized by the *startIds* input argument, which satisfies traversal conditions given by arguments *revisionEnd*, *revisionStart*, and *flowTypes*.

```
MATCH p = (s)-[*]->(e)
WHERE id(s) IN $startIds
AND all (rel IN relationships(p)
  WHERE rel.revisionStart <= $revisionEnd AND rel.revisionEnd >= $revisionStart
  AND type(rel) IN $flowTypes
)
RETURN e
```

6.2.2 Centroids rule

The query for the Centroids rule with *Count Mode* = *CHILDREN_TABLE* settings:

```
MATCH (s)<-[:hasParent]-()%s-[r]-%s()-[:hasParent]->(e)
WHERE NOT id(s) IN $ignoredIds
AND type(r) IN $flowTypes
AND r.revisionStart <= $revisionEnd AND r.revisionEnd >= $revisionStart
AND NOT s = e
RETURN s, COUNT (DISTINCT e) AS num
ORDER BY num DESC
LIMIT $limit
```

The query for the Centroids rule with *Count Mode* = *CHILDREN_COLUMN* settings:

```
MATCH (s)<-[:hasParent]-()%s-[r]-%s(e)-[:hasParent]->(eParent)
WHERE NOT id(s) IN $ignoredIds
AND type(r) IN $flowTypes
AND r.revisionStart <= $revisionEnd AND r.revisionEnd >= $revisionStart
AND NOT s = eParent
RETURN s, COUNT (e) AS num
ORDER BY num DESC
LIMIT $limit
```

Operation	Db hits (thousand)
Total	1439
Find the children neighbors parents	555
Traverse children edges	538
Find centroid candidates	146
Other	200

Table 6.3. Number of database hits of the main Centroid rule query and its decomposition

Table 6.3 shows the number of database hits for the following configuration:

- Count Mode: CHILDREN_TABLE
- Direction: BOTH

The rule finds all centroid candidates, traverses its children, and inspects all their flow relationships. Most of the database hits are caused by the inefficient repeating traversal of flow edges and requests for their revisions property, which cannot be indexed in Neo4j. Operations such as *DISTINCT*, *COUNT*, and *ORDER BY* do not generate any db hits because they are entirely performed in memory.

The proposed method for improvement is to create a custom stored procedure that would traverse all relationships in a given revision interval and flow type and then increment counters for their hierarchical node parents.

6.2.3 Restricted flows rule

The main query for the Restricted flows rule with *Report Mode = REPORT_COLUMN* settings:

```
MATCH allowedPath=(s)-[r*0..%d]->(l)
MATCH (l)-[rejectedRel]->(e)
WHERE id(s) IN $startIds
AND id(l) IN $allowedIds
AND NOT id(e) IN $allowedIds
AND all (n IN NODES(allowedPath) WHERE id(n) IN $allowedIds)
AND type(rejectedRel) IN $flowTypes
AND rejectedRel.revisionStart <= $revisionEnd
AND rejectedRel.revisionEnd >= $revisionStart
AND all (rel IN r
  WHERE type(rel) IN $flowTypes
  AND rel.revisionStart <= $revisionEnd AND rel.revisionEnd >= $revisionStart
)
RETURN l AS startNode, e AS endNode, collect(DISTINCT(s)) AS startNodes
```

The main query for the Restricted flows rule with *Report Mode = REPORT_TABLE* settings:

```
MATCH allowedPath=(s)-[r*0..%d]->(l)
MATCH (l)-[rejectedRel]->(e)
WHERE id(s) IN $startIds
AND id(l) IN $allowedIds
AND NOT id(e) IN $allowedIds
AND all (n IN NODES(allowedPath) WHERE id(n) IN $allowedIds)
AND type(rejectedRel) IN $flowTypes
AND rejectedRel.revisionStart <= $revisionEnd
AND rejectedRel.revisionEnd >= $revisionStart
AND all (rel IN r
  WHERE type(rel) IN $flowTypes
  AND rel.revisionStart <= $revisionEnd AND rel.revisionEnd >= $revisionStart
)
MATCH (s)-[:hasParent]->(sParent)
MATCH (l)-[:hasParent]->(lParent)
MATCH (e)-[:hasParent]->(eParent)
RETURN lParent AS startNode, eParent AS endNode, collect(DISTINCT(sParent))
```

It is useless to evaluate this rule in terms of database hits because it is almost entirely dependent on the complexity and a number of input restrictions. Restrictions such as *"Detect data flows from table A to table B"* are very fast. On the contrary, more complex scenarios such as *"Detect data flows going from resource Oracle to resource Teradata"* may become very computationally expensive, because the engine starts his traversals in all nodes under the Oracle resource (and there can be many), and evaluates all paths from these into the maximum depth repetitively.

Moreover, the *REPORT_TABLE* mode is even more complex, because it has to find hierarchical parents of every detected restricted flows and aggregate those correctly.

6.2.4 Isolated components rule

To avoid keeping too much information in memory, the rule business logic is implemented by two separate queries. The first query finds all candidates for isolated objects, which means all nodes of one of the configured node types.

```
MATCH (p:Node)-[r:hasParent|hasResource]->()
WHERE p.type IN $types
AND r.revisionStart <= $revisionEnd AND r.revisionEnd >= $revisionStart
RETURN id(p)
```

Afterwards the list of ids of those candidates called *pIds* is inserted into the main query and evaluated as follows:

```
UNWIND $pIds AS pId
// (e) are all nodes in the (p) subtree
MATCH (p)<-[:hasParent|hasResource*0..]-(e)
WHERE NOT pId IN $ignoredIds
AND id(p) = pId
WITH p, collect(id(e)) AS eIds
MATCH (p)
WHERE NOT EXISTS {
  (s)%s-[r]-%s(e)
  WHERE type(r) IN $flowTypes
  AND id(e) IN eIds
  AND NOT id(s) IN eIds
  AND r.revisionStart <= $revisionEnd AND r.revisionEnd >= $revisionStart
}
RETURN p AS node, p.type AS nodeType
```

Operation	Db hits (thousand)
Total	75
Find descendants in candidates subtrees	58
Inspect flow edges from those subtrees	15
Other	2

Table 6.4. Number of database hits of the main Isolated components rule query and its decomposition

Table 6.4 shows the number of database hits with the following configuration:

- Isolated Mode: NO_INGOING
- Node Types: Database, Server

In the first query, all nodes given by indexed type property are almost immediately collected. During the main query, the engine firstly finds all descendants in the subtrees of those isolated candidates. This operation triggers most db hits because the engine has to traverse a lot of relationships with non-indexed revision properties. Then all flow edges in the ingoing direction are inspected.

For this rule, there are not proposed further improvements because it seems efficient enough. To improve readability for developers, it would be helpful to decompose the queries into custom stored procedures.

6.2.5 The longest chains rule

The main difficulty with the Longest chains rule is that the problem is *NP-hard*, as discussed in section 3.4.4 on the page 27. Therefore the Neo4j engine does not have a method for efficient evaluation. The query finds all possible paths of flow edges matching given properties, removes the subpaths of longer ones, removes paths containing duplicated nodes (hence cycles), and sorts them from the longest paths.

The main query for the Longest chains rule with configuration *Resource Mode* = *MULTI_RESOURCE* and *Cycle Mode* = *CYCLE_AVOID*:

```
MATCH path=(s)-[*%d..]->(e)
WHERE all (node IN nodes(path)
  WHERE NOT id(node) IN $ignoredIds)
AND all (rel IN relationships(path)
  WHERE type(rel) IN $flowTypes
  AND rel.revisionStart <= $revisionEnd AND rel.revisionEnd >= $revisionStart
)
// Get rid of subchains
AND NOT EXISTS {
  ()-[r]->(s)
  WHERE type(r) IN $flowTypes
  AND r.revisionStart <= $revisionEnd AND r.revisionEnd >= $revisionStart
}
AND NOT EXISTS {
  (e)-[r]->()
  WHERE type(r) IN $flowTypes
  AND r.revisionStart <= $revisionEnd AND r.revisionEnd >= $revisionStart
}
// Avoid flow cycles if required
AND NOT apoc.coll.containsDuplicates(NODES(path))
RETURN length(path) AS pathLength, nodes(path) AS nodes
ORDER BY pathLength DESC
LIMIT $limit
```

Operation	Db hits (thousand)
Total	2669
Traversal of allowed relationships	1708
Detection and removal of shorter sub paths	575
Removal of cycles	325
Finding of all potential start nodes	48
Other	2

Table 6.5. Number of database hits of the main Longest chains rule query and its decomposition

Table 6.5 shows the number of database hits for the following configuration:

- Resource Mode: MULTI_RESOURCE
- Cycle Mode: CYCLE_AVOID
- Minimal length = 12 (the longest chains in this graph has length = 16)

This rule has no recorded time in table 6.2 on the page 58 because the query sometimes had not finished successfully for unknown reasons. Usually, it helped to run the query repeatedly or perform other graph operations. Moreover, the engine occasionally decided to run the same query with different operations orders, resulting in the change of the total number of database hits. However, this rule's non-deterministic behavior is bothersome and probably would require in-depth insights into how the Neo4j internal cache works.

The removal of cycles (configured by parameter *Cycle Mode = CYCLE_AVOID*) was performed by the stored procedure *apoc.coll.containsDuplicates()*. The author also implemented this function in Cypher without dependency on the APOC library, but the performance has degraded significantly. Although it looks like that this configuration caused 325 thousand additional database hits, it improves the overall rule performance because it dramatically reduces the number of traversed relationships by around 50 %.

Another important configuration to adjust when running this rule is the *minimal length* parameter. Currently, the best approach is first to run the rule initially with the larger *minimalLength* value, and if no results are to be found, repeat running the rule with decreasing value of the argument. The engine firstly finds all possible paths over a given length and other properties and then sorts them at the end.

However, suppose the *minimal length* parameter is set to a small value. In that case, the engine has to keep all found paths in the memory, resulting in a very computationally expensive ordering at the end or even complete memory overflow. With the user-defined procedure implementing the entire rule, the evaluator could dynamically drop short unpromising paths during the traversal, so the problem with memory overflow and demanding order would be solved. Nevertheless, this is probably not possible to implement entirely in the Cypher language.

To summarize, this rule is an exemplary candidate for implementation in the form of a custom stored procedure instead of using Cypher query. This should improve functionality, code clarity, but most importantly, the performance and deterministic behavior with complete control of evaluation. However, even after that, the performance may still not be sufficient enough and may require the introduction of some heuristics. The source code of procedure *expandConfig()* from the APOC library might serve as an example for the implementation, although this method is not sufficient because it does not allow traversal based on the relationship properties, as discussed in section 4.1.5 at the page 32.

6.2.6 Independent flows rule

The main query for the Independent flows rule rule with configuration *Dependency Modes = WRITE_DEPENDENCY*, *READ_DEPENDENCY* and *PROCESS_DEPENDENCY*.

```
MATCH (cs)-[:hasParent*%d]->(cp)<-[:hasParent*%d]-(ce)
MATCH p_s_cs = (s)%s-[]-%s(cs)
MATCH p_e_ce = (e)%s-[]-%s(ce)
MATCH p_s_jobS = (s)-[:hasParent*]->(jobS:Node)
MATCH p_e_jobE = (e)-[:hasParent*]->(jobE:Node)
WHERE jobS.type IN $jobTypes
AND jobE.type IN $jobTypes
AND NOT id(jobS) IN $ignoredIds
AND NOT id(jobE) IN $ignoredIds
AND NOT jobS = jobE
AND all (rel IN relationships(p_s_cs)
  WHERE rel.revisionStart <= $revisionEnd AND rel.revisionEnd >= $revisionStart
  AND type(rel) IN $flowTypes
)
AND all (relIN relationships(p_e_ce)
  WHERE rel.revisionStart <= $revisionEnd AND rel.revisionEnd >= $revisionStart
  AND type(rel) IN $flowTypes
)
AND all (rel IN relationships(p_s_jobS)
  WHERE rel.revisionStart <= $revisionEnd AND rel.revisionEnd >= $revisionStart
)
AND all (rel IN relationships(p_e_jobE)
  WHERE rel.revisionStart <= $revisionEnd AND rel.revisionEnd >= $revisionStart
)
RETURN jobS, COLLECT(distinct(jobE)) AS jobsElist
```

When running the rule with *Dependency Mode = ANY_DEPENDENCY*, the previous query is run three times with all three mode settings, and results are combined.

The query for the rule with settings *Dependency Mode = NO_DEPENDENCY* puts into a list all job nodes in the graph and then removes the ones found in the previous query with *ANY_DEPENDENCY* settings.

Operation	Db hits (thousand)
Total	11
Inspect visited nodes	3.7
Traverse the path patterns	3.5
Find descendants of job nodes	3.1
Other	0.6

Table 6.6. Number of database hits of the main Independent flows rule query and its decomposition

Table 6.6 shows the number of database hits for the following configuration:

- Dependency Mode: WRITE_DEPENDENCY
- Node Types: Talend

As expected, the engine starts evaluating the anchored start job nodes and then continues by the specified concatenated paths while evaluating relationship revision properties. At the end of the entire path, the end nodes are evaluated whether they are also of a particular node type.

Same as in the isolated components rule, although the author does not propose any further improvements, it would be helpful to decompose the queries into custom stored procedures to improve overall readability.

6.3 Rules evaluation

The previous section explained particular issues of single rules. At this place, the author discusses common discoveries and problems from analyzing rules performance.

The following list shows the order in which the Cypher queries are generally evaluated. The first items are the most accessible, requiring a little processing, while the last items require the most computing power [45].

1. Anchor node labels and their indexed properties
2. Relationships types
3. Anchor node non-indexed properties
4. Node labels of traversed nodes
5. Properties of traversed nodes and relationships

In general, the engine firstly tries to start the evaluation paths in anchor nodes, which are the nodes that can be quickly found by defined labels or indexed properties. From these nodes, it is easy to follow relationships of a given type, as discussed in section 4.1.6. On the contrary, the most demanding operation is to query by the properties of traversed nodes and relationships. Thus for efficient traversal, these should be reduced as much as possible.

As shown in table 6.2 on the page 58, the rule queries with anchored start nodes have proved the best performance and required the least processing in terms of database hits. Hence the developer creating own rules should aim to design the rules in a way allowing efficient retrieval of start nodes of the traversed paths in a graph.

Unfortunately, traversal in the Manta storage model heavily depends on revision information stored as a relationship property, which currently cannot be indexed in Neo4j. This is one of the main drawbacks of Manta integration with Neo4j.

Chapter 7
Enhancements

The previous chapters described the process of development and testing of a working prototype. It was verified that the rules engine project might become helpful for various types of end-users. In this chapter, the author proposes additional improvements and plans for future development. Since the application is not in a fully production-ready version, the author also discusses the suggested launch plan enabling a short time to market to receive quick feedback from customers.

7.1 Enhancements

This section summarizes the initial functional and non-functional requirements and evaluates how they were fulfilled. The project has allowed examining the general rules engine concept in detail. A few ideas and difficulties have emerged during the process; hence this section also discusses those thoughts with proposed solutions for further enhancements in terms of maintainability and overall performance.

7.1.1 Requirements summary

Two following tables contain descriptions of collected functional and non-functional requirements from sections 3.2.2 and 3.2.3 with their realization in the project.

Number	Description	Realization
i	Diverse technical skills of end-users	The rules provide default parameter values but also offer detailed customization by making use of more complex input parameters.
ii	Scaling flexibility	The engine allows the creation of new rules and new functionalities, with almost no internal dependencies on other Manta modules. This leads to potentially simple application scaling in the future.
iii	Documentation	The project contains a file *readme.md* with the how-to-run description. Implemented classes include Java documentation, and the rules functionality with input and output parameters are described in this thesis in the implementation section 5.2. However, the specific rules are not equipped with sample templates of input parameterization files, which must be completed before a launch in the production environment.
iv	Performance	The rules performance was evaluated in detail in section 6.2 and will be further discussed in section 7.1.3.
v	Localization	As required, the entire project and source code are in English.

Table 7.2. Summary of non-functional requirements and their realization in the project

Number	Description	Realization
i	The server repository will use Neo4j	Neo4j database was examined in depth. The implemented prototype is wholly implemented in Neo4j and contains the dump importer that stores Manta data in the Neo4j server.
ii	Flexibility and simplicity when parametrizing rules	All implemented rules provide enough flexibility of input parameters and can be extended by new parameters allowing even more customization. The rules provide default parameter values (where possible), allowing initial running with almost no starting effort. Presently the engine does not allow rule chaining or simple automatic comparison of results of one rule ran with modified parameters, but this will undoubtedly be the subject of further research as it could bring many new use cases.
iii	Well-tested code	The chapter 6 proved that the code contains a sufficient amount of unit and integration tests. In addition, the project was tested for performance, and when applicable, the author proposed suggestions for increasing efficiency.
iv	Integration into the current Manta ecosystem	The project uses the developer tools from the Manta ecosystem and reuses the current storage model. Although the project is currently separated from the Manta ecosystem, it will be possible to incorporate the project into the platform to integrate with Public API or Manta Admin UI components.
v	Exception parameters for every rule	All implemented rules offer *ignorePaths* input parameter that allow skipping of a graph part for rule evaluation.
vi	Provide input validation mechanism	Currently, the prototype does not offer any sophisticated validation mechanism but only throws an exception at appropriate places with a detailed description. When the application is ready for the user's input by configurable files, a validation functionality must be implemented.
vii	Triggering	It was decided that the core component will not be responsible for the advanced scheduling of the rules engine. In the future, some external mechanism, such as the cron tool in Linux, can be used for rule scheduling [70].
viii	Results reporting	Presently, the only way to report output is to write to the logger console or redirect the output to the external file. However, it is expected that the application will provide an interface for the registration of additional methods for message export or notification in the future.

Table 7.1. Summary of functional requirements and their realization in the project

To summarize, although, for complexity reasons, some of the requirements were not completely fulfilled in the prototype implementation, the project architecture allows future completion of all requirements. On top of that, the rules engine should be scalable enough to allow the addition of new requirements from the business departments and the creation of new use cases by the development of new rules.

7.1.2 Maintenance and simplification

As seen from the source code of particular rule queries in Cypher in the previous chapter, some patterns are frequently repeated. Therefore, it would help separate those fragments into specific user-defined functions, which would be callable from the main Cypher queries. It would lead to more readable, testable, maintainable, and scalable code for future changes in Manta storage architecture. The stored functions could also serve as typical building blocks simplifying the future implementation of new rules. The following list proposes a few example candidates for user-defined functions:

- *"Find all leaves in subtrees of given nodes for a specific revision."*
- *"Find nodes of a given type for a specific revision."*
- *"Traverse the lineage flow from the starting nodes for a specific revision."*

It is expected that in the future, the rules engine will have a validation mechanism that would help users with the configuration of input parameters and instant error handling. This is an important feature as the current parametrization methods may not be intuitive enough for kind of all users.

As discussed on pages 37 and 53, the standardized metadata model for data from various resources is not implemented in Manta up-to-date. The model should allow end-users quicker adaptation of Manta repository structure and fully benefit from collected metadata with less effort when parametrizing rules, e.g., when defining node properties.

7.1.3 Performance

So far, the rules were not perfectly optimized. After the specific rules are proved to be useful for customers, they should be further tuned to increase the efficiency for fast evaluation, even on large datasets. Beyond the suggestions related to particular rules described in the previous chapter, there are a few general approaches to achieving better performance.

Firstly, it would be valuable also to create the rules entirely implemented as stored procedures and compare the time performance. Specific stored procedures could use not only imperative Cypher querying but also the declarative approach offered by *neo4j* API to inspect the efficiency of having one rule implemented in totally different ways. Ideally, the performance would be evaluated on enormous data size exceeding the current largest customer graphs.

With the current approach of using one main rule query, a significant disadvantage has appeared. The complex Cypher queries are hard to debug, as they do not provide any logging mechanism controlling the way and order how the nodes and relationships are visited. The traditional declarative approach usage allows users to log all traversal steps resulting in simpler query debugging.

Neo4j contributors suggest warming up the cache before the complex queries to achieve better performance for the freshly started Neo4j instances [71]. One of the options is to use *apoc.warmup.run()* function from the APOC library, but the optimal configuration settings and real effect on the rules engine's performance were not evaluated yet. However, the warm-up queries should be implemented in a deterministic way. This should avoid the situations in which the same query launched in the same conditions takes a very different time to finish every run.

Last but not least, the rules engine should benefit from the indexing features of Neo4j. A significant decrease of total database hits for the rule queries could be achieved by creating indexes on appropriate particular node properties. So far, only the value-based indexes were discussed. On the contrary, structure-based indexes could also dramatically improve the performance [9, 14]. Nevertheless, this would require extensive research of common patterns in rule queries and appropriate definitions of particular structures to index.

7.2 Deployment plan

The author also proposes a plan for turning the rules engine from the current prototype to the production version. The aim is to distribute the project to the most eager customers to receive feedback as early as possible. Having the reactions from end users provides valuable information for developers to decide which part of the application should receive the most focus. Primarily, it is essential to find out whether the rules output report and input parameters are understandable, documentation is intuitive, if customers are not missing any critical functionality, etc. The author proposes three subsequent approaches of how to distribute the application to the customers.

The initial version can be distributed to selected customers in the form of a single package in the JAR format, which would contain the dump importer into an embedded Neo4j database, implemented rules, and a few configuration templates for each rule. The end-user would provide a path to his Manta dump file and specify the rules to run with input parameters on the command line or by the prepared scripts. The result would be written into a JSON file. Using the embedded database, the user would not need to install Neo4j before; therefore, this evaluation could be done even before completing the migration process from Titan to Neo4j.

The second version of the rules engine would be using REST API, and likely it would be integrated into the current Public API component. This would be the first production deployment, and any customer could run the rules easily from their applications by HTTP requests. Moreover, the REST API provides an intuitive interface for configuration. On top of that, customers could also incorporate the rules engine tasks into their custom schedulers to run the engine in periodic intervals or triggered by specific actions, as discussed as the functional requirement (vii).

The subsequent and most user-oriented distribution of the rules engine would be full integration into Manta UI. The GUI could contain a new section that would become the central place for orchestrating all rules engine features. This involves selecting rules to run, interactive parameters configuration, creation of complex rules workflows, and potential configuration of notifications and messages to various systems.

Although the rules engine's current prototype does not seem close to a mature application with interactive GUI, it is essential to have visions of where the project should aim in the future. Having long-term plans helps design the entire architecture and ensures that the decisions made throughout the process will not stand against the fundamental requirements and future targets. The author believes that the implementation fulfilled the required objective of the prototype.

Conclusion

Data lineage is a map of data flows that provide valuable information for companies storing a large amount of data. Traditionally, a constructed data lineage has been visualized in the graphical user interface to understand data journeys in data warehouses. On the contrary, the *rules engine* tool would automatically detect the lineage patterns without manual monitoring of extensive data flows graphs. It leads to higher efficiency and increased confidence in own data. On this account, this work's main goal was to design and implement a module with an engine consisting of rules detecting the selected graph patterns in a data lineage.

At first, the author describes the theory behind graph databases and data storage methods in those systems. Data lineage concepts with real use-cases are then introduced, with the main focus on the Manta product and its detailed architecture.

In the next chapter, the author explained the entire requirements gathering process. Firstly, the motivation behind rules engine application and the main benefits over the previous situation were discussed. In collaboration with colleagues from presales, products, and marketing departments, there were collected functional and non-functional requirements. On top of that, this chapter summarized the practical objectives to achieve and compiled business rules examples with a detailed description of inputs and intended usage. The usage generally either ensures data privacy in various ways or aims to simplify the structure of complex storage systems. Eventually, it all leads to the companies cost-cutting and an increase in productivity. To sum up, finding enough use-cases confirmed the significance of the rules engine project.

During the analysis process, the selected Neo4j database and its platform were researched. Because the declarative way of database querying is entirely different from the imperative approach in retiring Titan database, there are a few required changes in the Manta data model. With that being said, it was proved that Neo4j brings many advantages and new features. The author also evaluated popular rules engine tools and explained why none of them are used in the practical part.

The author implemented the prototype with the rules engine's core functionality, which is responsible for evaluating rule patterns in the Manta storage. Chapter 5 outlines the development tasks, main implementation obstacles, and procedures to deliver new rules. With the description of the entire process of creating a rule from scratch, it was proved that this task is too complex to be fully developed by end-users themselves. Thus it was decided to provide customers a set of flexible rules that can be configured by various input parameters. The author implemented five rules and explained the roles of their input and output parameters. On top of that, the concrete output reports were demonstrated, and results were visualized in Manta UI. Consequently, it was verified that the business rules might add high value for many types of customers.

The evaluation chapter is devoted to summarizing the test methods used in the project, focusing on performance testing. The efficiency of implemented rules was measured in terms of the number of database hits since it was the primary comparable metric. As the database traversals were written in Cypher language, the queries containing main rules logic were exposed and described. Besides, the Neo4j profiling utility allowed a detailed examination of database queries leading to the location of significant bottlenecks. The final part of this chapter spells out how Cypher queries are executed by the Neo4j engine and suggests ways to optimize graph operations.

In the last chapter, the author summarized the fulfillment of objectives and initial requirements during the practical implementation. This section also contains the application's architecture enhancement proposals to achieve simplification and easy maintenance. The rules include a few repetitive patterns that could be effectively converted into stored procedures. Besides, the author proposed three deployment stages to achieve fast time to market and initial reactions from customers. Finally, when the application will be launched and rules are proved to be helpful, database queries should be further optimized to increase the overall efficiency. Thus a few ways to tune the performance are also suggested. An option is to apply structure-based indexes, but this would require future extensive research of Manta-specific patterns.

To sum up, the application's architecture allows future scaling, enrichment by new features, the addition of other reporting channels, and more. It is believed that end-users themselves will come up with ideas for new use-cases by creating brand-new rules or customized workflows by rules chaining. After all, these steps lead to the increase in the business value of the rules engine application and, in consequence, the entire Manta product.

References

[1] Kasey Panetta. *Gartner Top Data and Analytics Trends for 2021*. 2021.
`https://www.gartner.com/smarterwithgartner/gartner-top-10-data-and-analytics-trends-for-2021/`.

[2] Ian Robinson, JIm Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data 2nd Edition*. O'Reylly, 2015. ISBN 1491930896.

[3] Martin Svoboda. *B0B36DBS, BD6B36DBS: Lecture 1 - Database Systems Conceptual Modeling.* . Czech Technical University.

[4] Guru99.com. *Data Modelling: Conceptual, Logical, Physical Data Model Types.*
`https://www.guru99.com/data-modelling-conceptual-logical.html`.

[5] Visual-paradigm.com. *Conceptual, Logical and Physical Data Model.*
`https://www.visual-paradigm.com/support/documents/vpuserguide/3563/3564/85378_conceptual,l.html`.

[6] Charles Roe. *ACID vs. BASE: The Shifting pH of Database Transaction Processing - DATAVERSITY*. 2012.
`https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/`.

[7] Bryce Merkl Sasaki. *Graph Databases for Beginners: ACID vs. BASE Explained*. 2018.
`https://neo4j.com/blog/acid-vs-base-consistency-models-explained/`.

[8] CSc. Demlová, Marie. *Logic and Graphs Lectures.* . Czech Technical University.

[9] Jaroslav Pokorný, Michal Valenta, and Jaroslav Ramba. *Graph patterns indexes: Their storage and retrieval.* In: *ACM International Conference Proceeding Series.* New York, NY, USA: Association for Computing Machinery, 2018. 221–225. ISBN 9781450364799.
`https://dl.acm.org/doi/10.1145/3282373.3282374`.

[10] Michelle Knight. *What is a Property Graph? - Dataversity*. 2020.
`https://www.dataversity.net/what-is-a-property-graph/`.

[11] Jiří Vyskočil, and Radek Mařík. *PAL: Lecture 1 - Advanced algorithms asymptotic notation, graphs and their representation in computers.* .

[12] Ing Lucie Svitáková. *Query Analysis on a Distributed Graph Database.* . Czech Technical University.

[13] TechTarget. *Efficient indexing for performance*. 2006.
`https://searchsqlserver.techtarget.com/feature/Efficient-indexing-for-performance`.

[14] Jaroslav Pokorný, Michal Valenta, and Martin Troup. *Indexing Patterns in Graph Databases.* In: *DATA 2018 - Proceedings of the 7th International Conference on Data Science, Technology and Applications.* SciTePress, 2018. 313–321. ISBN 9789897583186.

http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006826903130321.

[15] Talend. *What is Data Lineage? (and How to Get Started) - Talend Real-Time Open Source Data Integration Software.* https://www.talend.com/resources/what-is-data-lineage-and-how-to-get-started/.

[16] Irina Ph.D. Steenbeek. *The Basics of Data Lineage - EWSolutions.* https://www.ewsolutions.com/the-basics-of-data-lineage/.

[17] Laura Sebastian-Coleman. *Measuring Data Quality for Ongoing Improvement*. Elsevier Inc., 2013. ISBN 9780123970336.

[18] Olivia Wassén. *What is Data Lineage? — NodeGraph*. 2019. https://www.nodegraph.se/what-is-data-lineage/.

[19] GetManta. *About the MANTA Platform MANTA.* https://getmanta.com/about-the-manta-platform/.

[20] GetManta. *Data Lineage for DataOps — MANTA.* https://getmanta.com/solutions/data-lineage-for-dataops/.

[21] RNDr. Hermann, Lukáš. *Podcast 027 — MANTA: Lukáš Hermann, VP of Engineering*. 2021. https://scriptease.lolo.team/917014/7142050-027-manta-lukas-hermann-vp-of-engineering.

[22] Amnon Drori. *What is Data Lineage? — Octopai*. 2021. https://www.octopai.com/what-is-data-lineage/.

[23] Peter Brejčák. *Datový startup Manta Tomáše Krátkého nabírá téměř 300 milionů korun. Přispělo i české Credo Ventures - CzechCrunch*. 2020. https://www.czechcrunch.cz/2020/10/datovy-startup-manta-tomase-kratkeho-nabira-temer-300-milionu-korun-prispelo-i-ceske-credo-ventures/.

[24] Jakub Ing. Moravec. *Analýza a návrh abstraktní vícevrstvé architektury pro práci s grafovou databází realizující metadatové úložiště pro data lineage.* . Czech Technical University.

[25] Ing. Pikna, Marek. *Webinar: MANTAtalks: Tips for Efficient MANTA API Usage*. 2020. https://getmanta.com/webinars/mantatalks-tips-for-efficient-manta-api-usage/.

[26] Ing. Sýkora, Jan. *Incremental update of data lineage storage in a graph database.* . Czech Technical University.

[27] GetManta. *MANTA Live — MANTA.* https://getmanta.com/manta-live/.

[28] Peter Wang. *Alation and Manta: Automating Advanced Data Lineage — Alation.* https://www.alation.com/blog/alation-manta-automate-advanced-data-lineage/.

[29] Plskova. Katerina. *Manta Goes Public with Its API! — MANTA*. 2017. https://getmanta.com/blog/manta-goes-public-with-its-api/.

[30] Swagger. *OpenAPI Specification - Version 3.0.3 — Swagger*. 2020. https://swagger.io/specification/.

[31] Karl E. Wiegers. When Telepathy Won't Do: Requirements Engineering Key Practices. *Process Impact.* 2000,

[32] Ulf Eriksson. *Functional vs Non-Functional Requirements - Understand the Difference*. 2012.

https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/.

[33] Ali Alakeel. A Guide to Dynamic Load Balancing in Distributed Computer Systems. *IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.6*. 2009,

[34] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction To Algorithms - Thomas H.. Cormen, Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein*. 2001.
https://books.google.cz/books?id=NLngYyWFl_YC&pg.

[35] Jaroslav Bc. Ramba. *Indexování struktur v grafovém DB stroji Neo4j II*. . Czech Technical University.

[36] Peter Neubauer. *Peter Neubauer on Twitter: "sarkkine Neo4j was developed as part of a CMS SaaS 2000-2007, became released OSS 2007 when Neo Technology spun out." / Twitter*. 2010.
https://twitter.com/peterneubauer/status/9248821667.

[37] Neo4j. *Introduction - Operations Manual*.
https://neo4j.com/docs/operations-manual/current/introduction/.

[38] Onofrio Panzario. *Learning Cypher*. Packt, 2014. ISBN 978-1-78328-775-8.

[39] Neo4j. *Neo4j APOC Library - Developer Guides*.
https://neo4j.com/developer/neo4j-apoc/.

[40] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, Andrés Taylor, Al Cypher, Université Paris-Est Alastair Green Neo, Tobias Lindaaker Neo, Stefan Plantikow Neo, Petra Selmer Neo, and Andrés Taylor Neo. Cypher: An Evolving Query Language for Property Graphs. 2018, 14 1433.

[41] Max De Marzi. *Neo4j Stored Procedure Training Part 2*. 2019.
https://www.slideshare.net/maxdemarzi/neo4j-stored-procedure-training-part-2.

[42] Max De Marzi. *Neo4j Stored Procedure Training Part 1*. 2019.
https://www.slideshare.net/maxdemarzi/neo4j-stored-procedure-training-part-1.

[43] Neo4j. *Path Expander Overview - APOC Documentation*.
https://neo4j.com/labs/apoc/4.1/graph-querying/path-expander/.

[44] Neo4j. *Neo4j Graph Platform - Developer Guides*.
https://neo4j.com/developer/graph-platform/.

[45] Neo4j. *Neo4j Developer Guides - Neo4j Graph Database Platform*.
https://neo4j.com/developer/.

[46] Neo4j. *Client applications - Neo4j Driver Manual*.
https://neo4j.com/docs/driver-manual/current/client-applications/.

[47] Neo4j. *Including Neo4j in your project - Neo4j Java Reference*.
https://neo4j.com/docs/java-reference/current/java-embedded/include-neo4j/.

[48] Michael Simon. *Testing your Neo4j-based Java application — Medium*. 2019.
https://medium.com/neo4j/testing-your-neo4j-based-java-application-34bef487cc3c.

[49] Baeldung. *Introduction to Spring Data Neo4j*. 2020.
https://www.baeldung.com/spring-data-neo4j-intro.

[50] Spring. *Spring Data Neo4j*.
https://spring.io/projects/spring-data-neo4j.

[51] Joy Chao. *Graph Databases for Beginners: Native vs. Non-Native Graph Technology*. 2018.
https://neo4j.com/blog/native-vs-non-native-graph-technology/.

[52] Gaurav Sarma. *Neo4j storage internals — Medium*. 2020.
https://medium.com/@gauravsarma1992/neo4j-storage-internals-be8d150028db.

[53] Qusay H. Mahmoud. *Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications*. 2005.
https://www.oracle.com/technical-resources/articles/javase/javarule.html.

[54] Ying Jin, Vadlamannati Lakshmi Venkata Sai Raja Bharath, and Jinaliben Shah. *Active rules in a graph database environment.* In: *EPiC Series in Computing*. EasyChair, 2020. 134–140.

[55] Martin Fowler. *RulesEngine*. 2009.
https://martinfowler.com/bliki/RulesEngine.html.

[56] Michal Bachman. *(Un)common Use Cases for Graph Databases*. 2016.
https://neo4j.com/blog/uncommon-use-cases-graph-databases/.

[57] Baeldung. *List of Rules Engines in Java*. 2019.
https://www.baeldung.com/java-rule-engines.

[58] Drools. *Business Rules Management System (Java™, Open Source)*.
https://www.drools.org/.

[59] Baeldung. *Introduction to Drools*. 2018.
https://www.baeldung.com/drools.

[60] Mahmoud Ben Hassine. *j-easy/easy-rules: The simple, stupid rules engine for Java*.
https://github.com/j-easy/easy-rules.

[61] Sunil Mogadati. *Drools Using Rules from Excel Files — Baeldung*. 2020.
https://www.baeldung.com/drools-excel.

[62] OpenL Tablets. *Easy Business Rules*.
http://openl-tablets.org/.

[63] Mark Proctor, Michael Neale, Peter Lin, Michael Frandsen, and Sam Griffith Jr. *1.2 Why use a Rule Engine? — Jbug.jp*.
http://www.jbug.jp/trans/jboss-rules3.0.2/ja/html/ch01s02.html.

[64] Veselin Pizurica. *What is a rules engine and when do we need one? — Waylay Blog*. 2018.
https://www.waylay.io/articles/what-is-a-rules-engine-and-why-or-when-do-we-need-one.

[65] Ryan Jolly Young. *Why should I use Drools? — Medium*. 2017.
https://medium.com/@ryanjollyyoung/why-should-i-use-drools-ba80be3b5311.

[66] GeeksforGeeks. *Software Engineering — Software Design Process*. 2019.
https://www.geeksforgeeks.org/software-engineering-software-design-process/.

[67] Ionos. *Visitor design pattern: explanations and examples - IONOS*. 2020.
https://www.ionos.com/digitalguide/websites/web-development/visitor-pattern/.

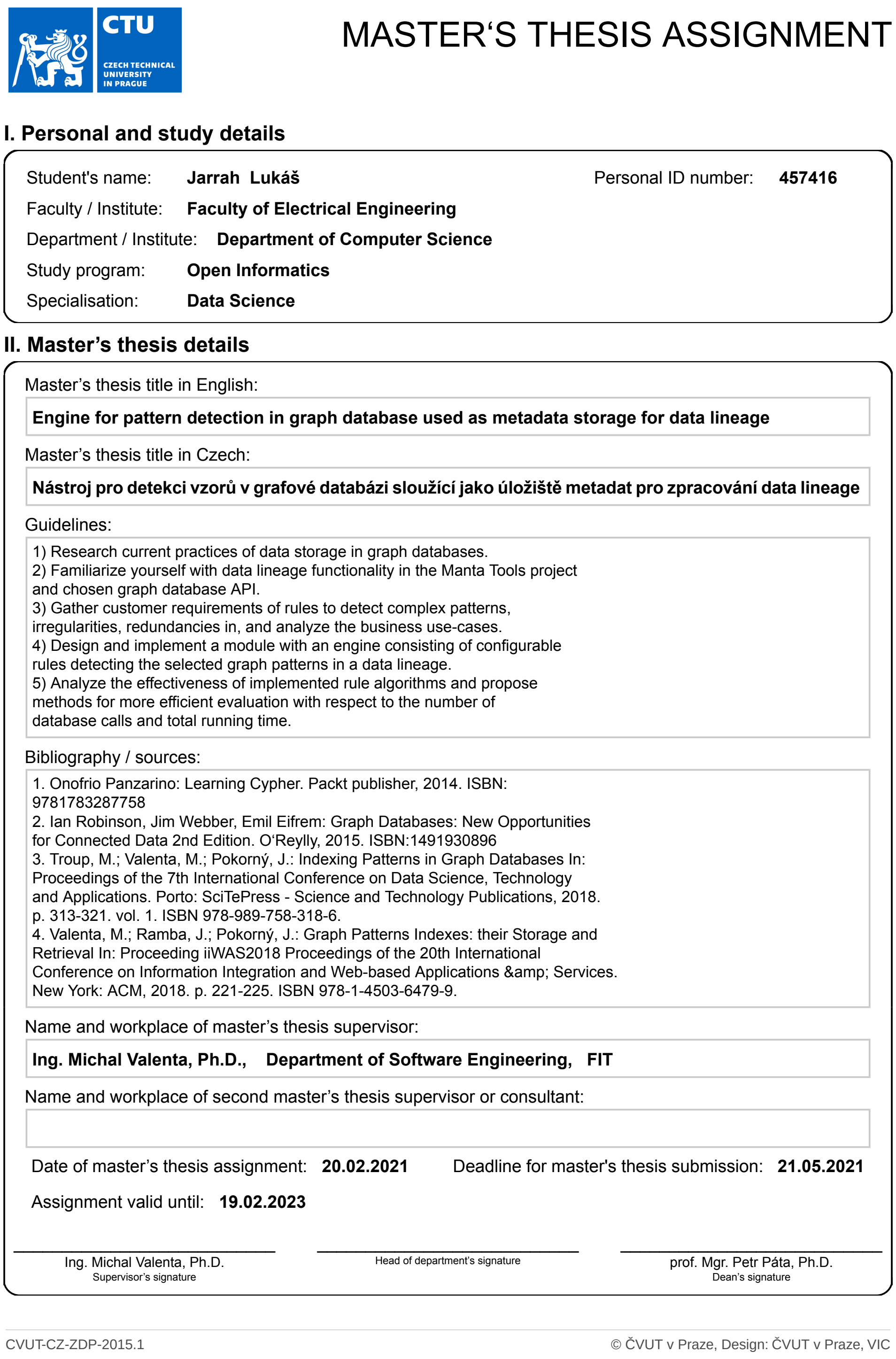
[68] Kent Beck. *Test Driven Development: By Example*. 2000. ISBN 978-0321146533.

[69] Alexandra Altvater. *The Ultimate Guide to Performance Testing and Software Testing: Testing Types, Performance Testing Steps, Best Practices, and More – Stackify*. 2017.
https://stackify.com/ultimate-guide-performance-testing-and-software-testing/.

[70] Suryadi K. *Cron Job: a Comprehensive Guide for Beginners 2021*. 2021. https://www.hostinger.com/tutorials/cron-job.

[71] Dave Gordon. *Warm the cache to improve performance from cold start - Knowledge Base Neo4j*. https://neo4j.com/developer/kb/warm-the-cache-to-improve-performance-from-cold-start/.

Appendix A
Specification



MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jarrah Lukáš** Personal ID number: **457416**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Data Science**

II. Master’s thesis details

Master’s thesis title in English:

Engine for pattern detection in graph database used as metadata storage for data lineage

Master’s thesis title in Czech:

Nástroj pro detekci vzorů v grafové databázi sloužící jako úložiště metadat pro zpracování data lineage

Guidelines:

1) Research current practices of data storage in graph databases.
2) Familiarize yourself with data lineage functionality in the Manta Tools project and chosen graph database API.
3) Gather customer requirements of rules to detect complex patterns, irregularities, redundancies in, and analyze the business use-cases.
4) Design and implement a module with an engine consisting of configurable rules detecting the selected graph patterns in a data lineage.
5) Analyze the effectiveness of implemented rule algorithms and propose methods for more efficient evaluation with respect to the number of database calls and total running time.

Bibliography / sources:

1. Onofrio Panzarino: Learning Cypher. Packt publisher, 2014. ISBN: 9781783287758
2. Ian Robinson, Jim Webber, Emil Eifrem: Graph Databases: New Opportunities for Connected Data 2nd Edition. O‘Reylly, 2015. ISBN:1491930896
3. Troup, M.; Valenta, M.; Pokorný, J.: Indexing Patterns in Graph Databases In: Proceedings of the 7th International Conference on Data Science, Technology and Applications. Porto: SciTePress - Science and Technology Publications, 2018. p. 313-321. vol. 1. ISBN 978-989-758-318-6.
4. Valenta, M.; Ramba, J.; Pokorný, J.: Graph Patterns Indexes: their Storage and Retrieval In: Proceeding iiWAS2018 Proceedings of the 20th International Conference on Information Integration and Web-based Applications &amp; Services. New York: ACM, 2018. p. 221-225. ISBN 978-1-4503-6479-9.

Name and workplace of master’s thesis supervisor:

Ing. Michal Valenta, Ph.D., Department of Software Engineering, FIT

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **20.02.2021** Deadline for master's thesis submission: **21.05.2021**

Assignment valid until: **19.02.2023**

Ing. Michal Valenta, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

CVUT-CZ-ZDP-2015.1 © ČVUT v Praze, Design: ČVUT v Praze, VIC

Appendix B
Glossary

ACID	■	Atomicity, Consistency, Izolacy, Durability
API	■	Application Programming Interface
APOC	■	Awesome Procedures On Cypher
BASE	■	Basically Available, Soft state, Eventual consistency
BFS	■	Breadth-first Search
BI	■	Business Intelligence
CSV	■	Comma-separated values
DB	■	Database
DBMS	■	Database Management System
DFS	■	Depth-first Search
DSL	■	Domain-specific language
DWH	■	Data Warehouse
ETL	■	Extract, Transform, and Load
GDB	■	Graph Database
GDPR	■	General Data Protection Regulation
GUI	■	Graphical User Interface
HTTP	■	HyperText Transfer Protocol
HTTPS	■	Hypertext Transfer Protocol Secure
JAR	■	Java Archive format
NoSQL	■	non-SQL or non-relational database
REST	■	Representational State Transfer
SQL	■	Structured Query Language
TDD	■	Test-driven development
UI	■	User Interface
UML	■	Unified Modeling Language
YAML	■	Ain't Markup Language

Appendix C
Attachments

C.1 List of files

`README.md`	Description of how to run the rules engine by using local Neo4j server instance
`rules-engine.zip`	Source code of the rules engine prototype implementation in Java with dependencies defined in pom.xml file
`thesis-text.pdf`	This document in PDF format
`thesis-text.zip`	Source code of the TEX document