



# **A Case Study on Transitioning from Relational Data models to Graph Data models in an Industrial Context**

**Johanna Ilonen**

Master's Thesis in Information Technology, Computer Engineering  
Åbo Akademi, Faculty of Science and Engineering  
Supervisor: Dragos Truscan

Vaasa 2023

# TABLE OF CONTENTS

ABSTRACT.....	
ABBREVIATIONS.....	
1. INTRODUCTION .....	1
1.1. Research questions.....	4
1.2. Research contributions and limitations .....	4
1.3. Thesis structure.....	8
2. BACKGROUND.....	10
2.1. Databases and database management systems .....	10
2.2. The GDBMS Neo4j.....	14
2.3. Data models.....	17
2.3.1. ER model.....	17
2.3.2. Relational model .....	21
2.3.3. Graph data model .....	23
2.4. Data modeling in the DB design phase.....	26
2.4.1. Understanding the problem .....	28
2.4.2. Conceptual data model.....	28
2.4.3. Logical data model .....	31
2.4.4. Physical data model.....	32
3. LITERATURE REVIEW.....	34
3.1. The relational DB and its weaknesses.....	34
3.2. Graph DB addressing the weakness of relational DB.....	36
3.3. Market leaders using graph DB.....	37
3.4. Use cases gaining success through graph DB.....	38
3.5. Analytical approach for deciding DB type .....	39
3.6. Dynamic data model.....	41
3.7. Conclusion.....	43
4. CASE STUDY .....	47
4.1. Experimental design .....	48
4.1.1. Goal definition.....	49

4.1.2. Hypothesis formulation.....	51
4.1.3. Variables .....	52
4.1.4. Design.....	53
4.1.5. The subject.....	55
4.1.6. The object.....	55
4.1.7. Instrumentation .....	55
4.1.8. Data collection.....	56
4.1.9. Analysis procedure .....	56
4.1.10. Evaluation of validity.....	57
4.2. Data modeling .....	58
4.2.1. Understanding data modeling needs .....	59
4.2.1.1. Summary.....	120
4.2.2. Logical data models.....	121
4.2.2.1. Relational data model.....	122
4.2.2.2. Graph data model .....	130
4.3. Graph data model implementation in Neo4j.....	140
5. ANALYSIS.....	144
5.1. Case study result.....	144
5.2. Competencies and knowledge.....	146
6. CONCLUSION AND FUTURE RESEARCH .....	148
6.1. Answer to research questions .....	148
6.2. Recommendations for future research .....	151
SWEDISH SUMMARY .....	153
REFERENCES.....	159

## ABSTRACT

**Author:** Johanna Ilonen

**Master's Degree Program:** Information Technology, Computer Engineering

**Supervisor:** Dragos Truscan

**Title:** A case study on transitioning from relational data models to graph data models in an industrial context

<b>Date:</b> 24.3.2023	<b>Number of pages:</b> 151	<b>Appendices:</b> -
<p>In this research, the graph data model challenges the well-known relational data model. The relational model, used by the relational data base, uses tables to present data and data relationships. The graph data model, used by the graph data base, explains the data as a connected graph. This fundamental structure makes the relational data model less dynamic and intuitive than the graph data model.</p> <p>In our experimental setup, the graph data model is more dynamic than the relational data model. An interesting finding is that when mapping the data model needs through data analysis, it is easier to build the relational data model than the graph data model. Building a well-functioning graph data model requires understanding on how the business stakeholders describe the problem and what type and questions they want to answer based on the data. To achieve the dynamic capability of a graph data model, the data modeler needs a mindset change from an analytical approach to a social one. The inputs from the business stakeholders are the key to success in graph data modeling.</p> <p>A company considering a change from a relational data base to a graph data base shall not follow hypes. Careful consideration and analysis are needed. The study shall show a clear indication of immediate and remarkable practical benefits in areas like query performance, flexibility, and agility.</p>		
<b>Language:</b> English	<b>Keywords:</b> data modeling, database, ERDPlus, arrows.app, Neo4j	



## ABBREVIATIONS

<b>3NF</b>	Third normal form
<b>ACID</b>	Acronym for <b>a</b> tomic, <b>c</b> onsistent, <b>i</b> solated, <b>d</b> urable
<b>CPU</b>	Central Processing Unit
<b>CRUD</b>	Acronym for <b>c</b> reate, <b>r</b> ead, <b>u</b> pdate, <b>d</b> eleate
<b>CTE</b>	Common Table Expressions
<b>DB</b>	Database
<b>DBMS</b>	Database Management System
<b>DBOP</b>	Delivery Bill of Process
<b>ER</b>	Entity-relationship
<b>GDBMS</b>	Graph Database Management System
<b>IM</b>	Information Management
<b>OSME</b>	Open Smart Manufacturing Ecosystems
<b>RAM</b>	Random Access Memory (The computer's short-term memory where data is stored as the processor needs it.)
<b>RDBMS</b>	RDBMS management system
<b>RQ</b>	Research Question
<b>SQL</b>	Structured Query Language
<b>SSD</b>	Solid State Drive (A server storage device that retains data in flash memory instead of a magnetic-based system like a hard disk drive.)
<b>STH</b>	Sustainability Technology Hub (Wärtsilä' s delivery center in Vaasa Finland.)

## 1. INTRODUCTION

In the latter part of 2021, Wärtsilä and partners started a Business Finland-funded project named Open Smart Manufacturing Ecosystems (OSME) [1]. The goal of the OSME project is to move from traditional linear value streams to a resilient collaborative network based on a digital foundation. This will enable proactive planning of activities, first time right, reduced lead times, traceability and feedback loops, optimized logistics, and better quality with less effort.

**Dynamic data models** and an understanding of data flow have been recognized as one of many focus areas to achieve an efficient and value-creating resilient collaborative network. According to Riihimäki, Director for Delivery Management at Wärtsilä, this network “enables Wärtsilä and the other ecosystem partners to **adapt and innovate to market needs**” [1]. Riihimäki also highlights the purpose of “helping our customers **continuously improve**” [1].

**Dynamic**, defined as continuously changing or developing [2], fits well with the statements from Riihimäki. From his comments, it can be derived that the data and its requirements change due to dynamic market needs, and the data model needs to adapt to this. The purpose of a **data model** is to provide an understanding of data needs and requirements to be addressed when designing a database (DB) to fit the needs of an organization [3]. Data models that structure data into tables have a weak response to change and require expensive configurations to reflect changing business needs [4] [5]. The **relational model**, used by the **relational DB**, is an example where tables present the data and relationships among the data [6]. An alternative is the **graph data model**, used by the **graph DB**, which shows the data structure as a connected graph [7]. The graph data model enables easier data model changes [7].

Another dynamic perspective are data models that allow **processes to be carried out with different approaches** while still managing the situation where the result will be combined and presented [4]. Wärtsilä has recognized that connecting and using data from various processes and systems is challenging. Especially use cases, with complex join queries for fetching data from relational DB, cause unacceptable query execution time [8]. This has made Wärtsilä interested in DBs that efficiently handle relationships between data elements. The graph DB, **explicitly designed to handle and store relationships between**

**data elements** [9], is considered an appealing choice. Of many alternative **graph database management systems (GDBMS)** in the market, **Neo4j** is selected by Wärtsilä due to its technical capabilities of being a native GDBMS, its easy-to-learn **Cypher** query language, and its position of being a GDBMS market leader [10].

Wärtsilä now seeks to extend its success from current implementations of Neo4j to use cases requiring dynamic data models with a particular focus on data relationships. This study focuses on understanding how data modeling for a relational DB differs from data modeling for a graph DB. Aim is to understand if one approach is more suitable for a dynamic environment and if relationships between data elements are easier to build and identify in a graph data model compared to a relational data model.

The result of the study is based on a literature review and a case study where a limited scope of engine manufacturing process data is modeled as a relational data model and a graph data model. The graph data model is implemented in Neo4j to get practical experience and understand what knowledge and competencies are needed for graph data modeling and implementation in Neo4j.

The study of data models and DB types is limited to graph and relational domains. The main reason for this limitation is Wärtsilä's increased interest in Neo4j graph DB technology. The relational DB selection is supported by its popularity. It topped the *DB-Engines ranking* survey list of the database management systems (DBMS) most frequently referenced on websites, job offers, and experience listings in LinkedIn profiles [11]. Figure 1 shows statistics from June 2022, where seven out of the ten most popular DBMSs are relational DBs [11]. Neo4j ranks nineteenth and is the most popular graph DB [11]. Trends in Figure 2 indicate that the relational database management systems (RDBMS) Oracle, MySQL, and Microsoft SQL Server keep a relatively steady trend as the top three DBMSs between 2013 and 2022 [12].

Why would an enterprise move from the popular relational DB to a graph DB? Some arguments can be that:

- The graph DB can be an alternative or an additional option if the relational DB does not manage well with an increased number of attributes, more data, higher speed requirements in business agility or data accessibility, and significantly more connections between data elements [9].

- In the graph, it is easy to add new data elements to adapt to new business requirements [13].
- In contrast to the relational DB, the graph DB is explicitly designed to handle and store relationships between data elements [9].

398 systems in ranking, June 2022

Rank			DBMS	Database Model	Score		
Jun 2022	May 2022	Jun 2021			Jun 2022	May 2022	Jun 2021
1.	1.	1.	Oracle +	Relational, Multi-model	1287.74	+24.92	+16.80
2.	2.	2.	MySQL +	Relational, Multi-model	1189.21	-12.89	-38.65
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model	933.83	-7.37	-57.25
4.	4.	4.	PostgreSQL +	Relational, Multi-model	620.84	+5.55	+52.32
5.	5.	5.	MongoDB +	Document, Multi-model	480.73	+2.49	-7.49
6.	6.	↑7.	Redis +	Key-value, Multi-model	175.31	-3.71	+10.06
7.	7.	↓6.	IBM Db2	Relational, Multi-model	159.19	-1.14	-7.85
8.	8.	8.	Elasticsearch	Search engine, Multi-model	156.00	-1.70	+1.29
9.	9.	↑10.	Microsoft Access	Relational	141.82	-1.62	+26.88
10.	10.	↓9.	SQLite +	Relational	135.44	+0.70	+4.90
11.	11.	11.	Cassandra +	Wide column	115.45	-2.56	+1.34
12.	12.	12.	MariaDB +	Relational, Multi-model	111.58	+0.45	+14.79
13.	↑14.	↑26.	Snowflake +	Relational	96.42	+2.91	+61.67
14.	↓13.	↓13.	Splunk	Search engine	95.56	-0.79	+5.30
15.	15.	15.	Microsoft Azure SQL Database	Relational, Multi-model	86.01	+0.68	+11.22
16.	16.	16.	Amazon DynamoDB +	Multi-model	83.88	-0.58	+10.12
17.	17.	↓14.	Hive +	Relational	81.58	-0.03	+1.89
18.	18.	↓17.	Teradata +	Relational, Multi-model	70.41	+2.02	+1.07
19.	19.	↓18.	Neo4j +	Graph	59.53	-0.61	+3.78

Figure 1. DB-Engines ranking survey list of most popular DBMS in June 2022 (picture source [11])

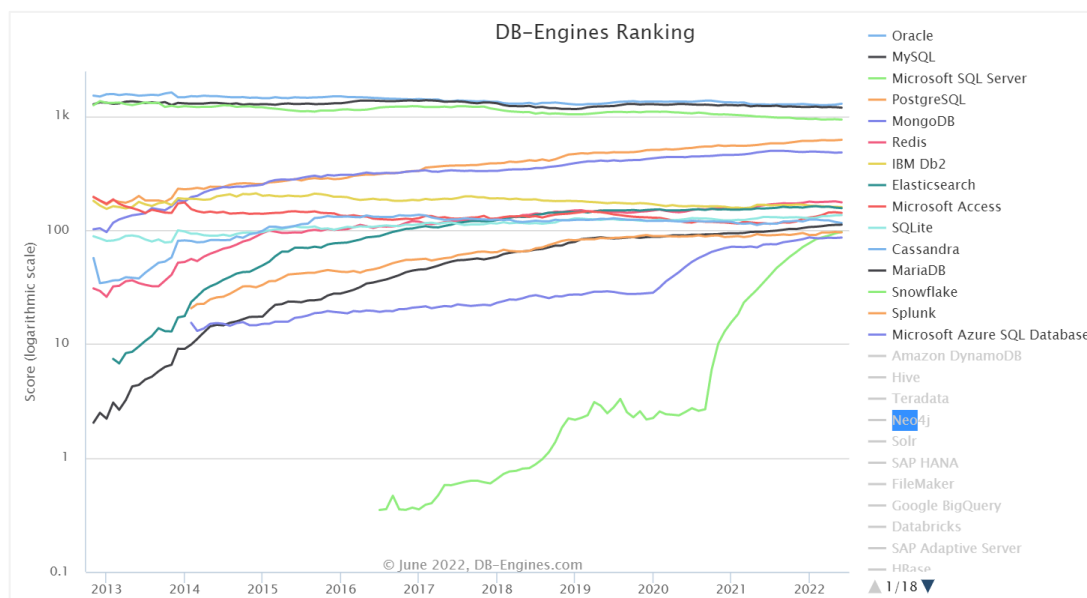


Figure 2. DB-Engines ranking survey trend of most popular DBMS (picture source [12])

This chapter continues by presenting the research questions in **chapter 1.1**. **Chapter 1.2** defines the expected research contributions, approaches, and limitations. **Chapter 1.3** describes the structure of this thesis.

## 1.1. Research questions

At Wärtsilä, the Neo4j GDBMS platform has successfully been used for some specific use cases. There is a desire to understand if this success can be extended to the manufacturing process data and later to value creation in internal and partner networks. This study focuses on finding solutions and answers to the following research questions (RQ):

- RQ1.** How does data modeling for a relational DB differ from data modeling for a graph DB?
- RQ2.** Can an experiment where the manufacturing process is modeled as a relational model versus a graph data model prove that the graph data model is more dynamic than the relational data model?
- RQ3.** How to present the manufacturing process data in Neo4j?
- RQ4.** What knowledge and competencies are needed for graph data modeling and implementation in Neo4j?

## 1.2. Research contributions and limitations

This chapter describes how the study results are produced and which limitations are set. The study is conducted in two sequential steps. A literature review is followed by a case study in an experimental set-up.

The **literature review** aims for basic knowledge in data modeling and DB implementation with a limitation to relational and graph domains. Information was sought from online books, white papers, research articles, tutorials, and case studies to answer **RQ1** and expressly understand:

1. What is a data model, and how does a data model for a relational DB implementation differ from a data model for a graph DB implementation?

2. How to choose between a relational DB and a graph DB implementation? What are the benefits and drawbacks, and what are the current trends?
3. Are graph data models and graph DB implementations dynamic?
4. Is managing and handling relationships among data elements easier in a graph DB than in a relational DB?

When searching for information, the focal point was to find answers to the above questions. Instead of conducting an exhaustive literature review, where all the material returned in the search engines is reviewed and further filtered based on specific criteria, the author's judgement on what knowledge is needed to carry out the empirical study was used in selecting the material. Further, it was verified that the content in the chosen material is trustworthy by investigating the reference material and checking if the material has been cited in other literature and whether the publisher or site provider is well known. The age of the material plays a vital role in a domain evolving rapidly. Therefore, the oldest source information used was twelve years old and fifty percent of the information was less than five years old.

The material was found through *Google*, *Google Scholar*, and *abo.finna.fi* search engines. *Google Scholar* is used for searching research articles, while *Google* is used for the rest. If the research article was not available for free in *Google Scholar*, it was fetched through the university library *abo.finna.fi*. If the article was not found in the university library, a corresponding research article was searched. Keywords utilized in the search are listed below, where each bullet represents an individual search criterion:

- Neo4j
- Data model
- Conceptual data model
- Logical data model
- Physical data model
- Flexible data model
- Dynamic data model
- Relational database
- Graph database
- RDBMS
- GDBMS
- Database

The knowledge gained from the literature review was utilized in a **case study** consisting of the following parts:

1. Relational and graph data modeling experiment design and case study (to answer **RQ2**)
2. Implementing the graph data model in the Neo4j standalone desktop version (to answer **RQ3**).

The data model in the case study covers the data needs for a limited scope of manufacturing process data for an engine produced in the Wärtsilä STH delivery center in Vaasa, Finland. The manufacturing process data in the range of the research is the Delivery Bill of Process (DBOP). The DBOP describes the sequence of assembly steps needed to produce a specific customer engine. The implementation of the graph data model in Neo4j is limited to a standalone Neo4j desktop implementation with no integrations to systems and databases in Wärtsilä. Neither is any data imported to the model.

To understand if the graph data model is more dynamic than the relational model, a small-scale case study with experimental set-up was carried out where the author modelled the identical data set as a relational data model and a graph data model. Initial idea was to run an experiment with ten to twenty Information Technology students as subject. However, due to time constraints of the thesis, the full-scale experiment was not possible. Instead, author performed own subjective case study, still following metrics of the experiment design. The experiment is designed to be possible to use in a later scenario where more subjects are available for the experiment.

As the dynamic capability is an indirect measure, it was determined from the following data:

- The time it took to understand the data needs for the model
- The data modeling time
- The number of data elements in the data model
- The effectiveness and efficiency of the implementation of the data model

In addition, the author made a qualitative and subjective analysis of the difficulty level of building the relational data model versus the graph data model.

Using only one subjective in the analysis causes the value of the evaluation to be statistically irrelevant. For a statistically relevant result the experiment design and analysis are recommended to be carried out with an entire class of Information Technology students. When this number of subjects would be available the experimental design and analysis planned for in in chapter 4.1 would give a statistically relevant research result.

The graph data model from the **RQ2** result, is utilized for **RQ3**. In RQ3, the graph data model is implemented in Neo4j. A qualitative review by an independent domain expert at Wärtsilä was made to understand if the implementation meets the expectations of Wärtsilä. Quantitative measures, such as query execution times, are beyond the scope of this study.

**RQ4** is answered based on the experience gained from the case study and learnings from the literature review.

Table 1 summarizes the methods used for answering the research questions.

*Table 1. Methods used to answer the research questions*

<b>Research question</b>	<b>Method</b>
<b>RQ1</b>	Literature review and case study
<b>RQ2</b>	Case study
<b>RQ3</b>	Case study
<b>RQ4</b>	Own experience gained through the literature review and case study

The high-level research plan is visualized in Figure 3. The thesis worker was responsible for executing the plan, implementation, and results. Guidance was expected from a Wärtsilä Information Management (IM) graph expert and the thesis supervisor. A Wärtsilä business stakeholder was utilized in requirement gathering and verification of the results. The study began with a literature review. The knowledge acquired in the literature review was used in the empirical study. The study was finalized with an evaluation of the results and defining the conclusions. The texts in blue visualize inputs. The black arrows indicate testing needs. For example, when the data model is built, it shall be verified against the requirements specified for the problem domain.



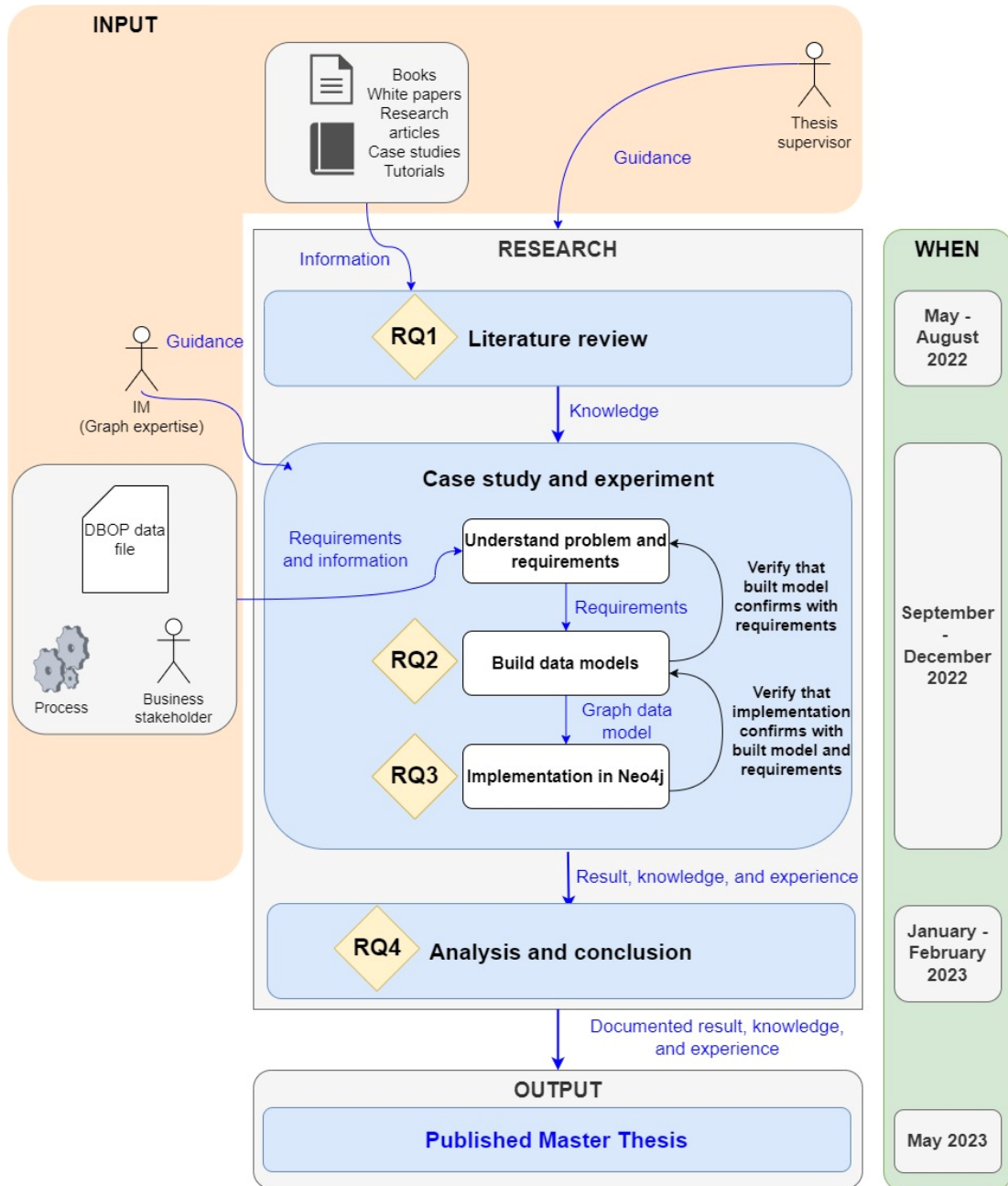


Figure 3. High-level research plan (author's picture)

### 1.3. Thesis structure

The remaining part of this thesis is divided into six main chapters:

- **Chapter 2** presents fundamental concepts in the field of data models and DB implementation. The reader gets a basic understanding of what a data

model is and how a data model for a relational DB implementation differs from a data model for a graph DB implementation.

- **Chapter 3** investigates how to select the suitable DB type for a problem at hand and discusses the literature concerning the dynamic capability of the data model, handling of relationships between data elements, and query performance.
- **Chapter 4** describes the case study, which contains three sequential steps: experiment design, data modeling and the implementation of the graph data model in the Neo4j standalone desktop version.
- **Chapter 5** analyses the learnings and results of the research. A part of the analysis contains a suggestion for competencies and knowledge needed for graph data modeling in Neo4j.
- **Chapter 6** concludes the thesis by revisiting the research questions to understand if the research objective is met and provides suggestions for future study.

## 2. BACKGROUND

This chapter presents fundamental concepts in data models and DB implementation. This knowledge supports the design and implementation decisions in the empirical study. The content of this chapter also serves as a good base for anyone starting a journey from relational DB to graph DB design and implementation. This chapter hence presents the findings from the analyzed literature for **RQ1**, and precisely:

- What is a data model, and how does a data model for a relational DB implementation differ from a data model for a graph DB implementation?

This chapter is divided into four main parts. **Chapter 2.1** introduce DB and DBMS, focusing primarily on the differences between a relational DB and a NoSQL DB. **Chapter 2.2** gives a brief introduction to Neo4j GDBMS. **Chapter 2.3** covers the data model concept and explains the ER, the relational data, and the graph data models. **Chapter 2.4** describes the process for designing data models for a relational DB versus a graph DB.

### 2.1. Databases and database management systems

The word *data* is the plural form of *datum* which means *one piece of information* or *one numerical form* [14]. Data can be stored on paper or in electronic form [15]. A frequently used electronic storage is a **database** (DB), an organized collection of stored data that can be easily accessed and managed [15]. The **database management system** (DBMS) is the software responsible for storing, retrieving, and running queries on the data in a DB [15]. The DBMS provides alternative user interfaces and services, such as data redundancy control, data sharing among multiple users, and data backup and recovery [15]. The DBMS is built for a specific type of DB [15]. The **RDBMS** is the software for the relational DB, and the **GDBMS** is the software for the graph DB.

The logical structure of a database is described by its **data model** [16]. The **relational model**, chapter 2.3.2, is the logical structure of the relational DB, and the **graph data model**, chapter 2.3.3, is the logical structure of the graph DB [16].

Figure 4 presents a principal sketch, in which users and applications use a DB through the DBMS.

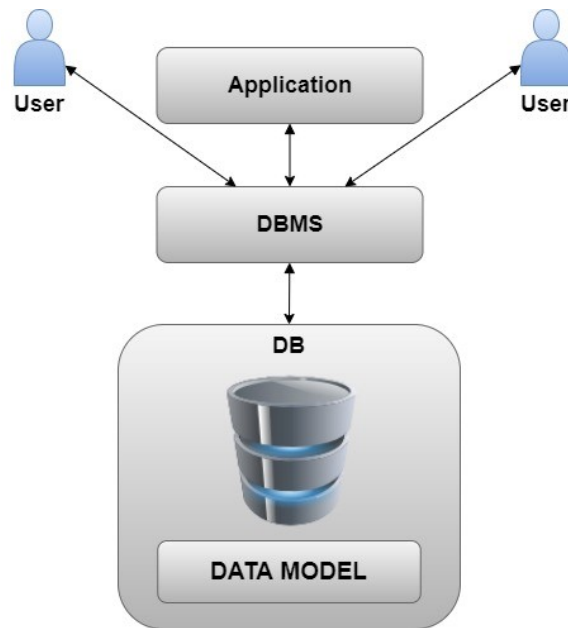


Figure 4. DBMS, DB, and Data Model (author's picture)

Figure 5 visualizes some examples of different DB types. Highlighted are the DB types covered in this research: **relational DB**, **NoSQL DB**, and **graph DB**. We notice that the graph DB is a type of NoSQL DB together with **key-value pair DB**, **column-oriented DB**, and **document-oriented DB** [15]. NoSQL has been developed as an alternative to the relational DB [17]. A concern of the relational DB in the age of big data with accelerating data volumes is that its performance degrades with increased data volume [17]. **Big data** refers to a large volume and wide variety of data captured from different sources with high speed [18].

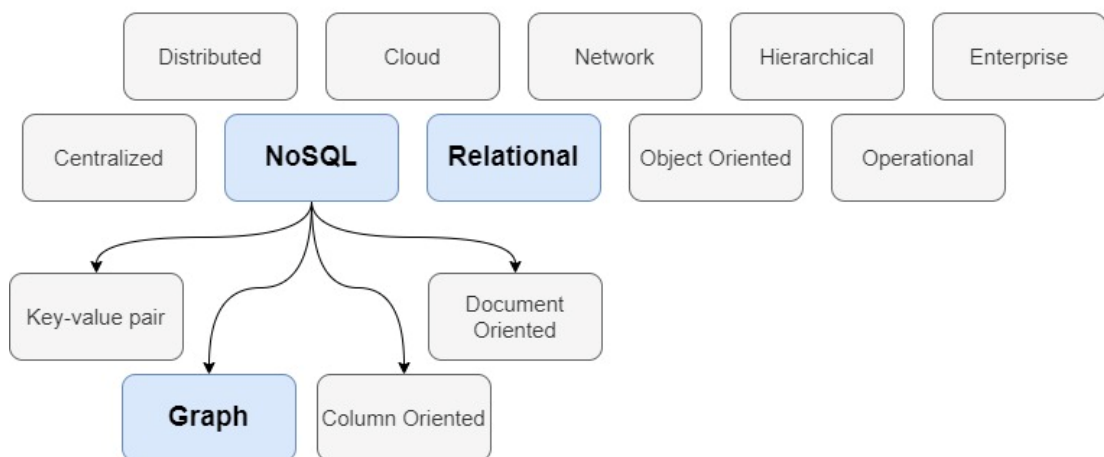


Figure 5. DB types (author's picture, adopted from [19])

Another concern of the relational DB is its rigid and pre-defined DB schemas that are hard to modify [17]. The relational **DB schema** contains the DB's tables, attributes, primary keys, and foreign keys, but no data [6]. The NoSQL DB is developed to handle significant data amounts and has a dynamic DB schema, also called **schemeless** [17], which can be altered without downtime or disruption in the service [18].

The NoSQL DB is not developed to replace the relational DB but rather to coexist with it [17] [18]. Depending on the needs of an application, it is even possible to define a hybrid data layer where the data from the application is stored in multiple DB types [17]. This approach can utilize the strengths of different DB types [17]. Table 2 summarizes a comparison made by Sahatqija et al. in a journal article published in 2018 [17]. It compares relational DB and NoSQL DB in terms of scalability and performance, data consistency, flexibility, query language, security, data management storage and accessibility. This comparison is generic for the DB type and does not consider a specific DBMS provider.

*Table 2. Comparison of features in a relational DB versus NoSQL DB (Created based on text in [17])*

<b>Feature</b>	<b>Relational DB</b>	<b>NoSQL DB</b>
<b>Scalability and Performance</b>	<p><b>Vertical</b> scalability = when data volume grows, the data storage and computing power expand only for existing hardware components like CPU capacity, RAM, and SSD of the DB server.</p> <p>The overall implementation cost increases with data growth.</p>	<p><b>Horizontal</b> scalability = when data volume grows, the system expands by adding more hardware components for data storage and processing power.</p> <p>This is a cheaper alternative than vertical scalability, and by distributing data on different servers, the performance of the DB increases.</p>
<b>Data consistency strategy</b>	<p>The main priority is to fulfill the <b>ACID</b> properties of <b>Atomicity</b>, <b>Consistency</b>, <b>Isolation</b>, and <b>Durability</b>. ACID ensures higher data reliability and integrity than DBs using the BASE principle.</p>	<p>Horizontal scalability makes it challenging to fulfill ACID. The <b>BASE</b> principle is used, and stands for <b>Basically Available</b>, <b>Soft state</b>, and <b>Eventually consistent</b>.</p> <p>The BASE is more flexible than ACID but has less consistency and reliability.</p> <p>Neo4j is entirely ACID compliant. (<b>Chapter 2.2</b>)</p>
<b>Flexibility</b>	<p>The DB schema is <b>static</b> and pre-defined before inserting</p>	<p>The DB schema is <b>dynamic</b> and does not have to be predefined.</p>

	<p>data. Making changes to a DB with data is challenging and can cause server failures and decreased performance.</p> <p><b>Only structured data</b> is supported.</p>	<p>This supports changes in structures and data types. Suitable for agile and scalable environments where continuous development and evolution can be expected.</p> <p><b>Structured, semi-structured, and unstructured data</b> are supported.</p>
<b>Query Language</b>	<p>A <b>standard</b> query language known as SQL is used. This powerful query language handles complex queries through a standardized interface. When knowing the SQL, a developer can write queries in any RDBMS.</p>	<p>No standardized query language. The GDBMS provider can create its query languages. A DB developer faces challenges when getting tasks to understand or write queries in different GDBMS systems.</p> <p>Neo4j has its query language, Cypher. It is intuitive with inspiration from SQL. Neo4j also supports other query languages. (<b>Chapter 2.2</b>)</p>
<b>Security</b>	<p>The structured data and vertical scalability make the security more straightforward to manage than the NoSQL DB.</p>	<p>A large amount of unstructured data distributed between multiple servers cause challenges for the security of the DB. If the NoSQL DBMS provider does not guarantee secure client-server communication, crucial factors like authentication, access control, secure configurations, data encryption, and auditing must be implemented by external methods.</p>
<b>Data Management-Storage and Access</b>	<p>The data is stored in tables, is highly normalized, and is very clean. Data redundancy is avoided through normalization, which slices data into small logical tables.</p> <p>The normalized data model is sometimes denormalized to avoid joins and get better query performance. (<b>Chapter 2.4.4</b>)</p> <p>There are alternative ways of replicating a relational DB between sites in a distributed system:</p>	<p>It can contain data redundancy as data is stored in collections without normalization.</p> <p>Data availability can be improved by replicating the DB between clustered servers. Two different approaches are utilized:</p> <ol style="list-style-type: none"> <li><b>1)</b> master-slave, where the slave can only read data</li> <li><b>2)</b> master-master, which gives the replicated master both read and write access to the data. This can cause inconsistency in the data.</li> </ol>

	<p>1) entire DB is replicated to all sites in the distributed system  2) no replication replicates only a fragment of the DB to one site  3) partial replication replicates some fragments.</p> <p>The replication improves the data availability but consumes a lot of time and storage. Therefore, the DB's performance declines.</p>	
--	---	--

In short, the NoSQL DB focuses on high performance, availability, data replication, and scalability. At the same time, the relational DB shows an advantage in data consistency, powerful query language, structured data storage, and security [17].

## 2.2. The GDBMS Neo4j

The most popular GDBMS provider is Neo4j [11]. This chapter explains what Neo4j is. Some of the central elements and terminology are discussed, providing beneficial resources for learning Neo4j's graph query language, Cypher. This chapter offers valuable information for anyone who will start implementing graphs in Neo4j.

Neo4j's history goes back to 2000, when the three founders of Neo4j encountered performance problems with RDBMS and initiated the first Neo4j prototype [20]. In 2007, the Neo4j company was founded in Sweden, and its first GDBMS was open-sourced [20]. In 2021 its open-source community had millions of downloads and hundreds of thousands of deployments [21]. The open-source version of Neo4j went under the name *Neo4j Community Edition* [22]. There is also an alternative for using the license-based *Neo4j Enterprise Edition* as a closed-source software application [22].

In Neo4j, data are stored as graphs, processed as graphs, and presented as graphs [23]. It is hence a native GDBMS with graph processing and storage [5]. **Native graph processing** utilizes index-free adjacency, which means nodes maintain direct references to nearby nodes [5]. This is a cheaper alternative than using global indexes [5]. With this approach, the query times are independent of the

total graph size and are only affected by the part of the graph searched by the query [5]. **Native graph storage** means the DB is specifically built for storing and managing graphs, having a stack engineered for performance and scalability [5]. In native graph storage, the relationship information is a primary data element [21]. If graph data is stored in a non-native graph storage DB, the relationship information can get lost, disconnected, or neglected, which are symptoms of data corruption [21]. The graph data model Neo4j use is the **labeled property graph**, which consists of the elements described in Figure 13 [5]. This is a variant of a property graph model [5], further described in chapter 2.3.3.

The consistency model which Neo4j uses for data transactions is ACID. Hence Neo4j reaches the same consistency levels as an RDBMS [24]. ACID stands for [25]:

- **Atomic** = All operations in a transaction need to succeed, or every operation is rolled back.
- **Consistent** = The DB is structurally intact when a transaction is completed.
- **Isolated** = Transactions do not compete with one another. The DB controls the continuous data access to make transactions appear to run sequentially for the users.
- **Durable** = The results of a completed transaction are permanent, ensuring data remains in the DB no matter the failures.

The technical details of Neo4j will not be discussed more profoundly here. Instead, the focus turns to **Cypher**, the query language of Neo4j [5]. Query languages like **SPARQL** and **Gremlin** are also supported [5]. Cypher is an open-source textual query language that utilizes ASCII art symbols in its syntax [26]. The inspiration for Cypher comes from the relational DB domain and the Structured Query Language (SQL) [26]. All the standard DB CRUD (**create**, **read**, **update**, and **delete**) operations are supported [26]. Cypher is the most intuitive and effortless graph query language to learn [5]. It can be understood by developers, DB professionals, and even business stakeholders [5]. Its ease of use derives from its close resemblance to graphs, as presented in Figure 6 [26].

Figure 6 shows a *MATCH* clause followed by a *RETURN* clause [26]. The pattern has been anchored to the node labeled *Person*, whose *name* property is *Dan* [26]. Cypher matches the remainder of the pattern to the nodes immediately surrounding this anchor point [26]. Hence, we can expect this *MATCH* pattern to



return various values for the *whom* variable while traversing through the graph [26].

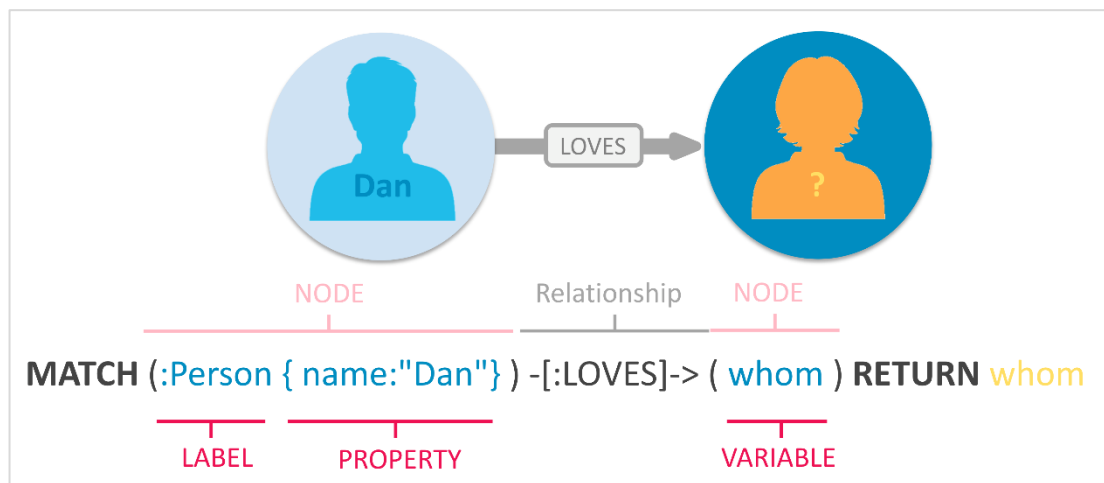


Figure 6. Cypher pattern example (Picture source: [26])

An alternative to anchoring a pattern to a specific node is to move the property lookup from the *MATCH* clause to a *WHERE* clause [5]. With guidelines in [5], the pattern in Figure 6 transforms to:

```
MATCH (a:Person) -[:LOVES]-> (whom)
WHERE a.name = 'Dan'
RETURN whom
```

Anyone familiar with SQL will find the resemblances when looking at the pattern above. Other customary clauses from SQL available in Cypher are *ORDER BY*, *SKIP LIMIT*, *AND*, and comparison possibilities like *p.unitPrice > 10* [26]. For further guidance on Cypher, we recommend investigating the Cypher Developer Guide [26] and the relevant version of the Neo4j Cypher Manual [27]. The current Cypher Manual is 4.4.

When starting the developer journey with Neo4j and Cypher, it is essential to remember to write a code that others quickly understand. This is enabled by following the guidelines in the Cypher style guide [28] and Cypher naming rules and recommendations [29]. An excellent approach is collecting hints from the Cypher query guidelines to ensure the written queries are optimized for execution performance [30].

## 2.3. Data models

The purpose of a **data model** is to provide an understanding of data needs and requirements to be addressed when **designing a DB** to fit the needs of an organization [3]. The data model can hence be considered as the design drawing of the DB, describing the data structure and purpose of the data [31]. Correctly designed, it improves the quality of information used in decision-making in an organization [31]. The quality of information means that it fits the needs, is available when needed and is accurate enough [31]. Tuning a model already providing the expected information quality increases costs and is not worth the effort [31].

The relational DB data model focuses on the objects, while the graph data model focuses on the objects' relationships [7]. Hurlburt et al. suggest a direct dependency between the quality of the data in a graph DB and the quality of the relationships in the data model [32]. This chapter focuses on understanding how the models produced for a relational DB and a graph DB differ. We limit our guide to the **ER** and **relational data model** for the relational DB design. For the graph DB, we restrict our attention to the **graph data model**, further limited to **hypergraphs** and **property graphs**.

### 2.3.1. ER model

The **ER model** is not a data model for a specific DB type, but a high-level data model used to describe the system on a conceptual level [16]. The ER model maps different entities and how they relate [16]. It describes how users experience a real-world situation without technical or system details [33].

The ER model is often used as the conceptual data model (chapter 2.4.2) when designing the relational DB, and after the relational DB is implemented, it is used in troubleshooting [34]. ER models can also be used in software engineering design to identify system elements and their relationships [34]. When moving from relational DB to graph DB, an ER model created for the relational DB is a valuable input when investigating the problem to model in the graph data model [7].

The ER model represents the data requirements of future users and the structure that fulfills these requirements [6]. The ER model can be defined in textual, Figure 7, or graphical form, Figure 8 [6]. The elements presented in Figure 8 are named in Figure 9, and the most relevant ones are further described below:

### Entity

An entity is an object, such as a person, a place, an event, or an item. The entity may be concrete, such as a student or a classroom, or abstract, such as a course or a department [6]. The ER model names entities using singular nouns [34]. An **entity set** is a set of entities that share the same attributes [6]. For example, the *student* entity set contains all the student entities in a university [6].

A **weak entity set** depends on the existence of another entity set [34]. Figure 9 gives two alternative notations for presenting weak entities. Either with the double-lined box or the double-lined diamond shape [6]. In Figure 8, the *section* is an example of a weak entity set that depends on the *course* entity set [6].

### Attribute

An attribute is a property or characteristic of an entity, a relationship, or another attribute [6, 34]. Each attribute is expected to hold a value in the DB implementation [6]. Figure 9 gives two alternative notations of visualizing attributes; either by listing the attributes within the entity set table or with oval shapes. When the attribute name is underlined, it symbolizes the entity's unique identifier, the **primary key** [6].

In Figure 8, the weak entity set *section* depends on the *course* entity set and has only a **partial key** [6]. The partial key for the *section* is  $\{sec\ id, year, semester\}$  and is used to distinguish the *section* entities from a course with the same *course\_id*. The primary key in the *course* is *course\_id*. The primary key for the *section* is a union of the primary key of the *course* and the partial key of the *section*:  $\{course\ id, sec\ id, year, semester\}$ .

A **composite attribute** identifies other attributes [6, 34]. It can group associated attributes and make the ER model cleaner [6]. An example is a composite attribute *address* and its attributes: *street*, *city*, *state*, and *zip\_code* [6].

A **multivalued attribute** is an attribute that can have more than one value [6, 34]. In Figure 8, the entity set *time\_slot* has the multivalued attribute *day* with both *start\_time* and *end\_time* [6].

An attribute based on another attribute is called a **derived attribute** [6, 34]. This is seldom used but could, for example, be the area of a circle derived from the radius of the circle [34].

Attributes can be left out if the ER model is modeled on a very high level [34]. When a relationship has an attribute, this attribute is specified as a **descriptive attribute**. In Figure 8, the *grade* is a descriptive attribute of the *takes* relation between *student* and *section* entity sets. In this example, the *grade* is utilized to specify the grade which the student gets from a specific course during a particular section.

### Relationship

A relationship describes how entities interact [6, 34]. Examples are the *teaches* and *takes* relationships in Figure 8. Combining the relationship with the entities, we understand that the *instructor teaches* the *section*, and the *student takes* the *section*. Verbs are used when naming the relationships in the ER model [34].

A relationship where the same entity participates more than once is called a **recursive relationship** [6, 34]. A recursive relationship named *prereq* for the *course* entity is presented in Figure 8. This example describes which *course*, identified with *prereq\_id*, is a prerequisite for another *course*, specified with *course\_id*.

Mapping the **cardinality** or **ordinality** of the relationship sets a constraint on how many entities another entity can be associated with [6, 34]. Figure 9 illustrates notations for the many-to-many, one-to-one, many-to-one, and one-to-many. One alternative is to give the cardinality in number format, and another is to format the relationship line between the entities [6]. The doubled lines between entity sets in Figure 8 indicate the **total participation** of an entity in a relation. For example, between the entity *instructor* and *inst\_dept*, the doubled line marks that an instructor must be associated with a department. In addition, the directed arrow from *inst\_dept* to the *department* indicates that each instructor can have only one associated department.

**Entity sets and their attributes, with primary keys underlined:**

- **classroom:** with attributes (building, room\_number, capacity).
- **department:** with attributes (dept\_name, building, budget).
- **course:** with attributes (course\_id, title, credits).
- **instructor:** with attributes (ID, name, salary).
- **section:** with attributes (course\_id, sec\_id, semester, year).
- **student:** with attributes (ID, name, tot\_cred).
- **time\_slot:** with attributes (time\_slot\_id, {(day, start\_time, end\_time)}).

**Relationship sets:**

- **inst\_dept:** relating instructors with departments.
- **stud\_dept:** relating students with departments.
- **teaches:** relating instructors with sections.
- **takes:** relating students with sections, with a descriptive attribute *grade*.
- **course\_dept:** relating courses with departments.
- **sec\_course:** relating sections with courses.
- **sec.class:** relating sections with classrooms.
- **sec.time\_slot:** relating sections with time slots.
- **advisor:** relating students with instructors.
- **prereq:** relating courses with prerequisite courses.

Figure 7. University DB ER model in textual format (picture source: [6])

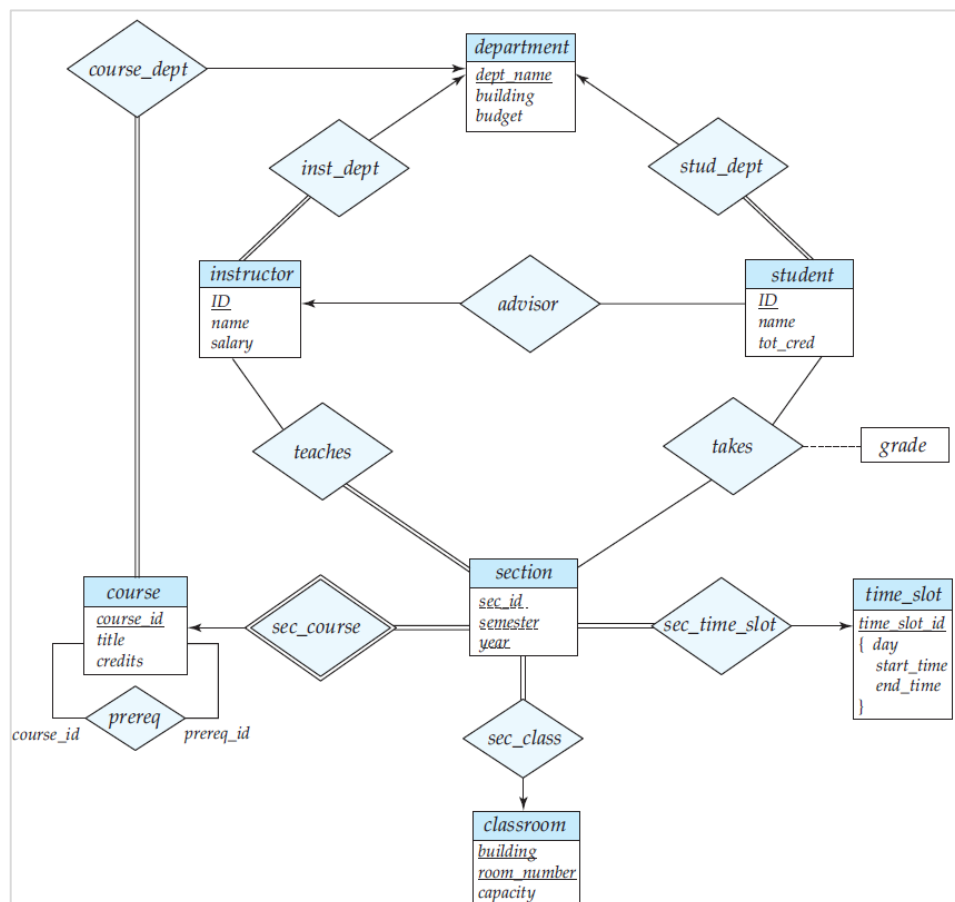


Figure 8. University DB ER model in graphical form (picture source: [6])

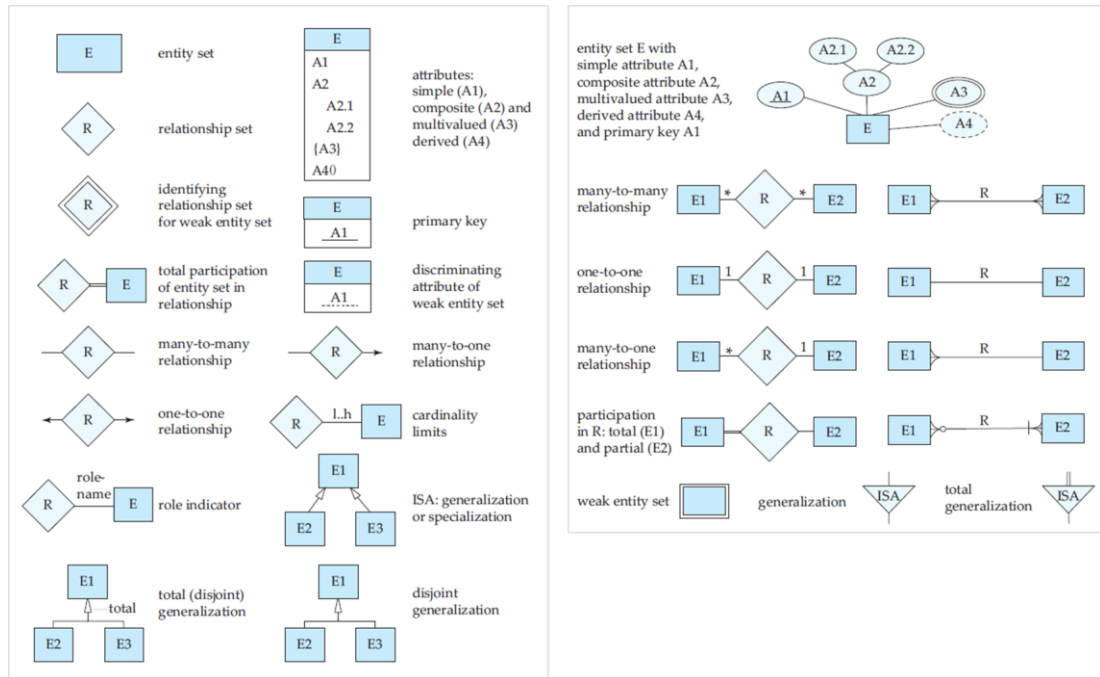


Figure 9. ER notations. Figure 8 uses the left ( $\leftarrow$ ) (picture source: [6])

## 2.3.2. Relational model

The relational DB is based on the **relational model**, which uses tables to present the data and relationships among the data [6]. Each table has a unique name [6] and a predefined set of columns [6]. The relational model structures the DB in fixed-format records of various types and is hence a **record-based model** [6]. Each table holds records of a specific kind, and each column in the table represents an **attribute** of that record type [6]. The table is also called a **relation**, and a row in the table can be called a **tuple** [6]. A constraint on the table is that each row needs to be unique [6].

The uniqueness of a row is realized by identifying one or a set of attributes that contains unique values, the **primary key** [6]. The attribute value is null if a non-primary key attribute on a specific row is unknown or does not exist [6]. References between tables in the DB are linked with **foreign keys** [6]. A table has only one primary key, while it can have several foreign keys [6]. A foreign key can contain null values, and its values do not need to be unique [6]. A specific set of rows in a table is referred to as a **relation instance** [6].

Figure 10 is an example of relation instances for *Instructor* and *Department* in the *university DB* where the following elements are identified:

1. Unique table (relation) name
2. Relation instance
3. Column names (attributes)
4. A row (tuple)
5. Primary key in *Instructor* table
6. Primary key in *Department* table
7. Foreign key in *Instructor* table

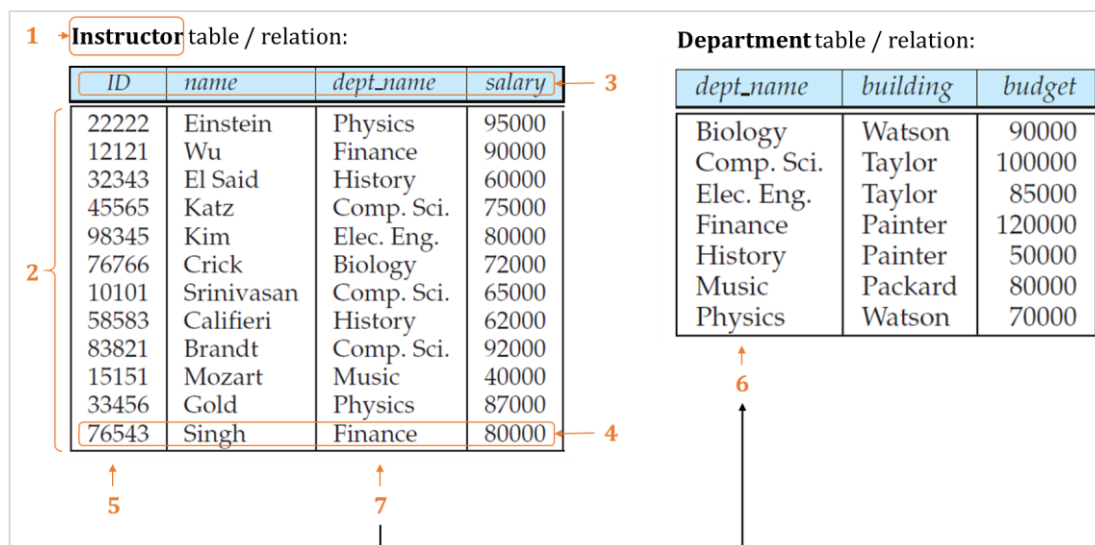


Figure 10. Elements in the relational model (author's picture, adopted from [6])

A **database instance** is all the data in a DB at a particular time [6]. This differs from the **database schema**, Figure 11, which is the logical design of a DB [6]. Each row in Figure 11 is a **schema** of a **specific table** [6]. The database schema can also be described in graphical form, named **schema diagram**, Figure 12 [6]. The database schema or schema diagram contains the tables in a DB, their attributes, primary keys and foreign keys, but no data [6].

```

classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
section(course_id, sec_id, semester, year, building, room_number, time_slot_id)
teaches(ID, course_id, sec_id, semester, year)
student(ID, name, dept_name, tot_cred)
takes(ID, course_id, sec_id, semester, year, grade)
advisor(s_ID, i_ID)
time_slot(time_slot_id, day, start_time, end_time)
prereq(course_id, prereq_id)

```

Figure 11. Database schema of a simple university DB (picture source: [6])

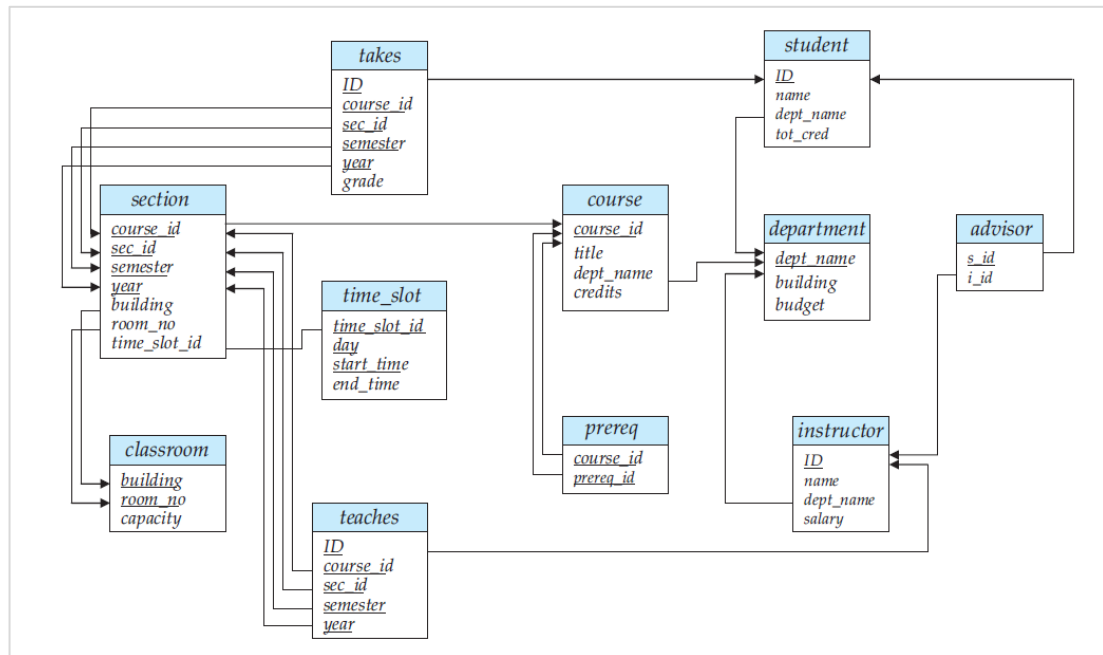


Figure 12. Schema diagram of a simple university DB (picture source: [6])

### 2.3.3. Graph data model

In the past decades, we have witnessed increased data volumes and rapid changes and variations in data structures [5]. Chapter 2.3.2 explained how the relational model captures data in rigid data structures that require high effort to modify once the relational DB implementation is done. To address the challenges faced in a rapidly changing data environment, the number of NoSQL alternatives have risen [5]. The NoSQL data models are argued to be less expressive than the relational model but more flexible and able to handle significant data volumes better [5]. The graph data model is of NoSQL type but still more explicit than the relational model [5].

The graph data model is flexible and handles significant data volumes and rapidly changing demands well [5]. In the graph data model, it is easy to add, modify or delete data elements based on the needs of the business [5]. The graph DB is based on the graph data model [16]. In this chapter, we investigate the graph data model structure to understand better the benefits it brings.



Several different graph data models are available in the graph DB's domain. These can be, for example, property graphs and hypergraphs [5]. The **property graph** is restricted to directed connections, with one start node and one end node [5]. The property graph is the most widely used graph model in GDBMS [5]. The property graph gives a straightforward and efficient modeling technique [5]. The **labeled property graph** is a property graph with the ability to use labels on the nodes for grouping and indicating specific roles of the nodes in a dataset. A **labeled property graph** consists of the elements described in Figure 13.

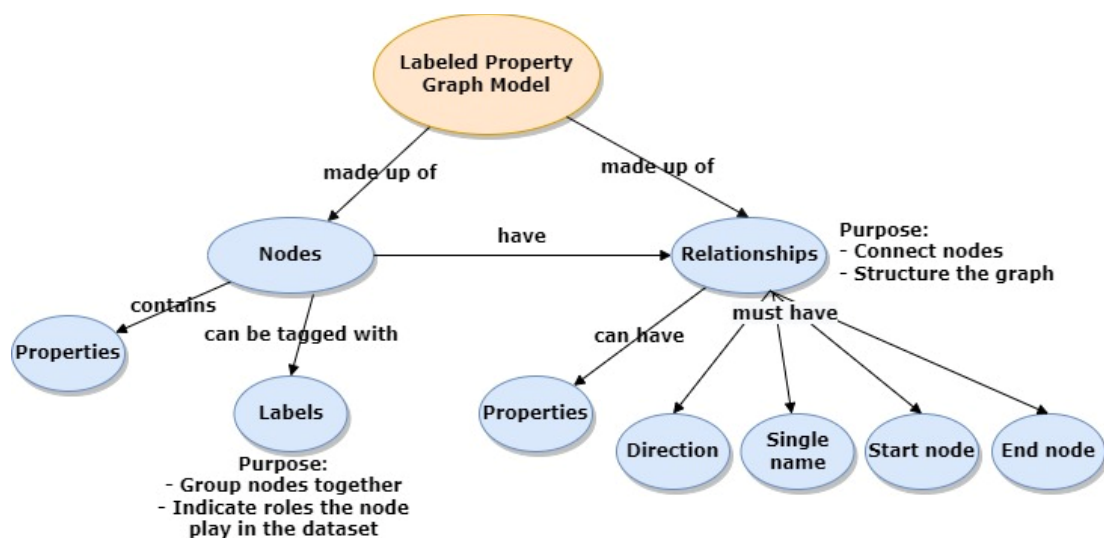


Figure 13. Elements in a labeled property graph model (Author's picture adopted from the text in [5])

The **hypergraph** is an alternative to the property graph that allows any number of start and end nodes for any relationship in the graph [5]. This graph model can be practical for capturing data with many-to-many relationships [5]. Figure 14 shows the difference between a hypergraph versus a property graph in a case where we want to model who owns the cars *Alice* and *Bob* drive [5]. To describe this situation, only one relationship is needed in the hypergraph, while six relationships are required for the property graph [5]. When the multidimensional hypergraph is used, there is a risk of missing essential details [5].

The property graph is more explicit and allows for fine-tuning the model by utilizing properties on the relationships [5]. Adding properties to the relationship in a hypergraph is not permitted [5]. With properties on the relationships, the primary owner of the car in Figure 15 can be identified using a property named *primary* on the *OWNS* relationship [5]. The *primary* property is *true* for the *OWNS* relationship between the car and the car owner [5]. Any use case can be modeled

with the multidimensional hypergraph or the property graph, and the builder of the data model or the type of application determines which is used [5].

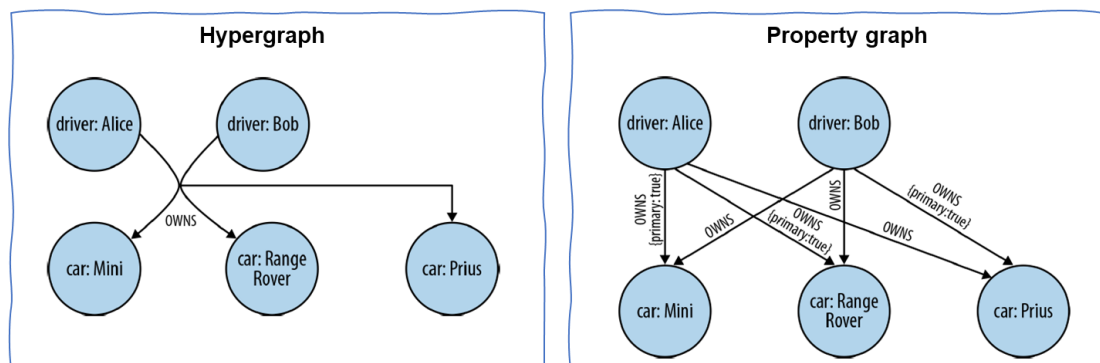


Figure 14. Hypergraph versus property graph (picture adapted from Figure A-7 and Figure A-8 in [5])

This study is limited to the **labeled property graph** as this is the graph data model used by the Neo4j GDBMS. Figure 15 shows a simple example of the labeled property graph model and how the different elements come into action [23].

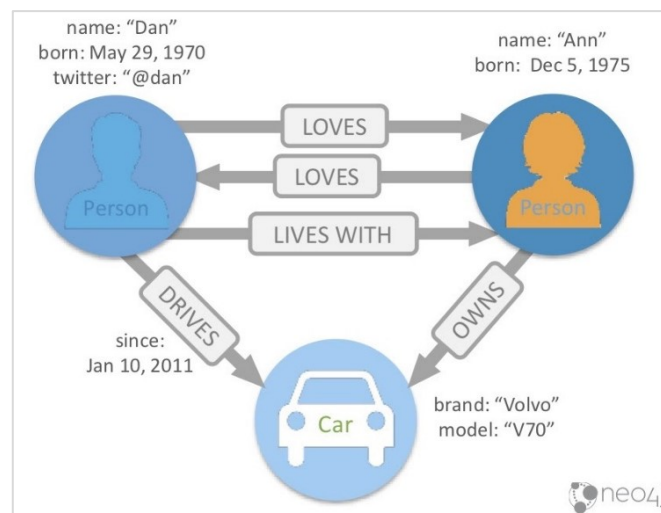


Figure 15. A simple example of a labeled property graph. (Picture source: [23])

Figure 15 contains:

- Three **nodes**
- Two different types of nodes with **labels**: *Person* or *Car*
- Node **properties** as name-value pairs
  - *Person* node: *name*, *born*, *twitter*
    - The *twitter* property does not exist on the node with the *name*: "Ann". If a property is inapplicable for a node, it is left out instead, as in a relational database, set the value to *null*.

- *Car* node: *brand, model*
- Four **Relationships**
  - **Direction** indicated with an arrow
  - **Single name:** *LOVES* (used twice), *LIVES WITH*, *DRIVES*, and *OWNS*
    - The *LOVES* relationship is needed in two directions, as we can have a case where *Ann* loves *Dan*, but *Dan* does not love *Ann*. *LIVES WITH* is required only in one direction because if *Ann* lives with *Dan*, it is explicitly understood that *Dan* also lives with *Ann*.
  - **Start node** where the arrow starts
  - **End node** where the arrow points
- Relationship **property** as name-value pairs: "*since: Jan 10, 2011*".

From this intuitive representation, we can see that Dan drives a Volvo V70 and has done this since Jan 10, 2021. However, Dan is not the owner of this car, as the only owner is Ann.

We note that all the relationships in a graph model are drawn with edges in one direction. The graph data model requires no primary or foreign keys [5] [7]. Compared to the relational model, there is no requirement that all entities in an entity group need to share the same attributes.

## 2.4. Data modeling in the DB design phase

The design phase of a DB aims at translating specific real-world problems, considerations, and questions into technical terms to use as guidelines in the DB implementation [7]. Hence, it is essential to understand the user requirements and data needs first [6] [7].

Reviewing the literature to understand how to perform design for a relational DB reveals a reoccurring pattern of producing a **conceptual data model**, a **logical data model**, and a **physical data model** [6]. We also recognize that the enterprise owning the DB decides on a specific notation and process to follow in data modeling [31]. The enterprise might also specify the modeling tools [31].

There are no rules to follow in data modeling. The general practice is to create an **ER model** as the conceptual data model of a relational DB [6] [16].

When searching for guidelines in the literature on how to perform data modeling for a graph DB, I noticed, as Roy-Hubara et al., that many articles describe the data modeling process for a specific use case but no data modeling rules to follow [35]. One reason is that data modeling for NoSQL DB is still maturing [36]. There is also a tendency to doubt the usefulness of data modeling for the schemeless NoSQL DB [36]. Schema or not, the importance of understanding and describing the data stored in the DB remains [36]. The best way to represent data structures is through data models [36]. It is not likely that there will be strict rules to follow for NoSQL design. It is my believe that, similarly as for the relational DB design, it will be up to the enterprise owning the DB to decide which design approach and tools to utilize [31].

The design process for a graph DB follows mostly the same design process as that of the relational DB. The main difference is the transition from a conceptual to a logical data model. The ER model is translated into tables in relational data modeling, while the graph data model remains a graph [7]. In graph data modeling, the conceptual graph is only enhanced by utilizing the elements for a specific graph data model type [7]. The physical data model used for the relational DB is not created for the graph DB. It is directly linked to a DBMS provider and depends on the DB schema [7]. For the schemeless GDBMS, a physical data model is not needed [7]. Testing the model to ensure that no poor design decisions are made is frequently highlighted as the final step in graph data modeling [5] [7].

### 2.4.1. Understanding the problem

The first step in the DB design phase is to thoroughly understand the user requirements and data needs [6] [7]. These can be derived from user interviews and analysis carried out in the enterprise [6]. If a DB implementation is available, the DB schema, ER, and other models are beneficial for understanding data structures and terminology already used in the problem domain [7].

According to Fernigrini [36], the data structure is essential in the relational DB design. When modeling a NoSQL DB, **the type of queries to be executed on the data is the focal point**. Robinson et al. agree that a graph data model shows how related issues are considered and communicates essential questions in the modeled domain [5]. Bechberger and Perryman confirm this and state that the physical data model is equal to the queries addressed in the problem domain [7].

To reduce data model change needs, the queries should be defined before the modeling [7]. In addition, the questions need to be prioritized [37]. The prioritization is needed because no model will be perfect for everything, and there will always be a need for tradeoffs [37]. Identifying the queries that provide the most significant business value and need the highest performance is a critical step at this point [37].

The output from the initial step is a textual description of how the problem is understood, written in a language understood by business stakeholders [7]. This forms the base for the conceptual design phase [6] [7].

### 2.4.2. Conceptual data model

The conceptual data model describes how the users experience real-world situations without technical or system details for a relational DB and graph DB. Therefore, it is an excellent tool for communicating requirements between business stakeholders and developers [7]. The conceptual data model can be as simple as a graph with entities and relations drawn on a whiteboard in a meeting between business stakeholders and developers [5] [7]. It is important to remember in this phase that this is a description of the understanding of the

system drawn from the business stakeholder's point of view [7]. Hence, developers will not start solving the problem and not think about the actual DB implementation [7].

For the graph DB design, I found guidelines for how to carry out the whiteboard drawing in two sequential steps, Table 3. The entities are first identified and grouped, and then the relationships between the entities are added [7] [37].

Table 3. Guidance for the conceptual data modeling (own table created based on text in [7] [37])

What	How
<b>Identify and group entities.</b>	<ul style="list-style-type: none"> <li>• Identify entities and name them as singular nouns. Focus on understanding the “What” and “Who.”</li> <li>• Identify groups of entities by listening to business stakeholders and identify if some nouns are used interchangeably. For example, the <i>user</i>, <i>employee</i>, and <i>client</i> could form an entity group named <i>Person</i>.</li> </ul>
<b>Identify relationships between the entities.</b>	<ul style="list-style-type: none"> <li>• Identify the relationships by focusing on functional questions and understanding the “How.”</li> <li>• Verify that the model supports forming sentences: <i>entity – relationship – entity</i>. For example, <i>Restaurant – Serves – Cuisine</i>.</li> <li>• Properties for the entities and relationships do not need to be included in this model. If some are identified, it is good to list them separately to review again in the logical design phase.</li> </ul>

For the relational DB, no guidelines for the whiteboard drawing were recognized, but I noticed an often-used approach to transform the whiteboard sketch into an *ER model* [6] (see chapter 2.3.1). The guidelines for the ER model describe the entity as an object such as a person, a place, an event, or an item. The entity may be concrete, such as a student or a classroom, or abstract, such as a course or a department [6]. The ER model names the entities using singular nouns [34]. This description correlates with the graph data model's descriptions in [5] [7].

However, spotting nouns in the domains through speech is a risk, as it can cause situations where all entities are not found [5]. The reason is that many technical and business jargon uses nouns instead of verbs [5]. For example, we say *email one another* instead of *sending an email* or *google* instead of *searching Google* [5]. This way of speaking also causes a risk for the relationships, which are named in the ER model and the graph data model with verbs [7] [34] or verbal phrases [6]. Instead of missing entities, the way we speak can cause falsely identified relationships [5].

I also noted a correlation between the statement for the ER model regarding the **entity set**, which is a group of entities sharing the same attributes [6], and how [5] describes the possibility of grouping entities with labels in a graph data model. A clear difference between the ER model and the graph data model is that all the entities in a graph group do not need to have common attributes, Figure 15.

Figure 16 shows an example output from the conceptual design phase for a graph DB [37]. In a meeting with business stakeholders, a whiteboard drawing is created to map entities and relationships for movies [37]. The whiteboard drawing is digitalized, and the syntax for the relationship labels, expected by Neo4j in the property graph, is introduced [37]. According to Bechberger and Perryman [7], the correct syntax can be left to the logical design phase.

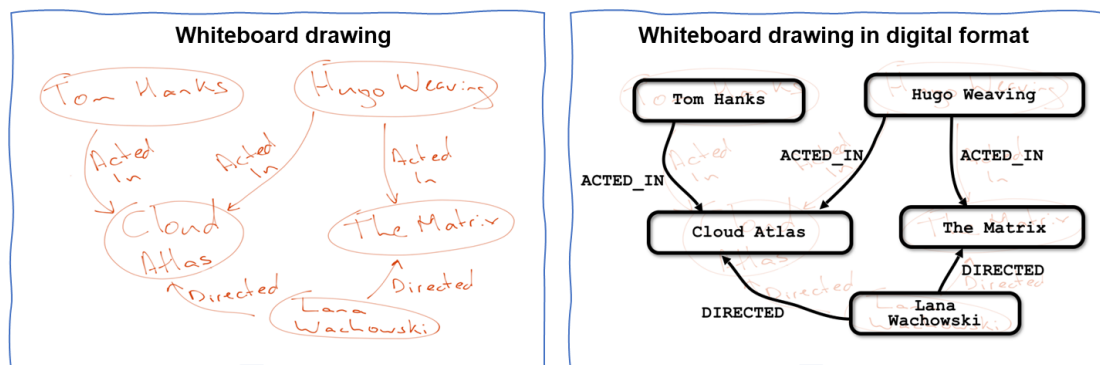


Figure 16. Conceptual data model as drawn in a meeting with business stakeholders and then digitalized (picture adapted from figures in [37])

The conceptual data model, having the form of a graph, both in the design process for a relational DB and a graph DB, is used as an input for the logical data model. Before moving to the logical design phase, it is an excellent practice for the relational DB to verify the model against transactions, which the future users will perform on the data [6]. Transactions mean the updating, searching, retrieving, inserting, and deleting of the data [6]. In graph DB design, the suggestion for testing comes after the logical design phase before implementing the model in a graph DB [5] [7].

### 2.4.3. Logical data model

The **logical data model** produced in the logical design phase defines how the DB should be implemented without specifying which DBMS should be used [33]. When designing the relational DB, the high-level **ER model** is mapped into tables to fulfill the expectations of the logical structure of the **relational model** [6]. For the graph DB, the conceptual data model remains a graph when considering the data model requirements of the **graph data model** [6]. This model adds further details to the conceptual data model and functions as the base for the physical data model design [33].

Chapters 2.3.1 and 2.3.2 presented an example of a university relational DB being designed by creating the ER and then the relational models. Involved in the creation of this model are usually data architects and business analysts [33]. Together, they develop a technical map of rules and data structures based on technical and performance requirements [33]. Attribute types are specified with exact precisions and lengths. To avoid duplicate data entries and to ensure only related data is stored in each table, normalization is usually applied until the *third normal form* (3NF) [33]. In the relational DB design, an attribute on a relationship or other attribute is allowed in the conceptual data model, but no longer in the logical data model, (see chapter 2.3.2).

The conceptual data model for a graph DB is **enriched** by clarifying relevant roles and labels, attributes and properties, and relationships [5]. Figure 17 shows how the conceptual data model in Figure 16 has been transformed into a **logical data model**. Bechberger and Perryman highlight that the usual differences between the conceptual data model and logical data model are that something that was an entity in the conceptual data model is implemented as a property on a node in the logical data model [7]. In graph DB design, the logical data model is the final design step. Hence, it is emphasized to test this model before implementing it in a graph DB [5] [7]. In the testing phase, the questions identified in *understanding the problem* and the *conceptual data model* are utilized to verify if it is possible to traverse through the model to find answers to the questions [7].



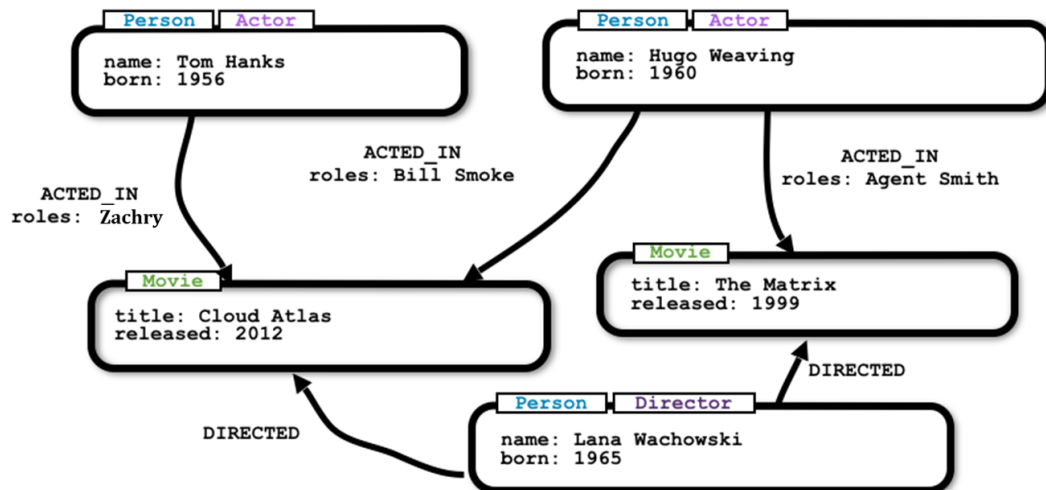


Figure 17. The logical data model for the same movie example as in Figure 16 (picture source [37])

#### 2.4.4. Physical data model

The logical data model needs refinement to a physical data model when the DB system requires the schema to be explicitly specified [7]. As the **GDBMS are schemeless**, this is seldom the case for a graph DB [5] [7]. For the relational DB with its rigid schema, the **physical data model** defines how the DB shall be implemented utilizing a **specific RDBMS** [33]. This model contains the specification of the physical features of the RDBMS [6].

Physical features can decide the form of file organization as well as views and index structures [6] [33]. Naming conventions of the RDBMS need to be followed for naming the tables and attributes, actual data types need to be set for the attributes, and constraints, such as primary and foreign keys, need to be specified [33] [38]. For better performance in DB queries, the normalization made in the logical data model can be abandoned in the physical data model [5]. This is called **denormalization** [5]. A simple example of this is shown in Figure 18, where the upper frame shows normalized data and the lower frame shows denormalized data. Through denormalization, the email attribute has been inlined in the user table. This reduces the penalty of joining operations in queries but introduces data redundancy [5]. For the relational DB, the physical data model is used in the actual implementation of the DB [33]. Therefore, it is easy to transform into SQL scripts that are utilized in creating the DB schema [38].

3NF normalized data					
<b>User</b>				<b>Email</b>	
User Id	Forename	Surname	Join date	User Id	email
1100022	Johanna	Ilonen	28.5.2022	1100022	<a href="mailto:johanna@abo.fi">johanna@abo.fi</a>
1100023	Sara	Zassi	30.5.2022	1100022	<a href="mailto:johanna@wartsila.com">johanna@wartsila.com</a>
				1100022	<a href="mailto:johanna@gmail.com">johanna@gmail.com</a>
				1100023	<a href="mailto:saraZassi@hotmail.com">saraZassi@hotmail.com</a>
				1100023	<a href="mailto:sara@abo.fi">sara@abo.fi</a>

Denormalized data				
<b>User</b>				
User Id	Forename	Surname	Join date	email
1100022	Johanna	Ilonen	28.5.2022	<a href="mailto:johanna@abo.fi">johanna@abo.fi</a>
1100022	Johanna	Ilonen	28.5.2022	<a href="mailto:johanna@wartsila.com">johanna@wartsila.com</a>
1100022	Johanna	Ilonen	28.5.2022	<a href="mailto:johanna@gmail.com">johanna@gmail.com</a>
1100023	Sara	Zassi	30.5.2022	<a href="mailto:saraZassi@hotmail.com">saraZassi@hotmail.com</a>
1100023	Sara	Zassi	30.5.2022	<a href="mailto:sara@abo.fi">sara@abo.fi</a>

Figure 18. Example of 3NF normalized versus denormalized data (author's picture)

### 3. LITERATURE REVIEW

Chapter 1 mentioned Wärtsilä's increased interest in the graph DB. The graph data DB is assumed suitable for a dynamic business environment where it reduces query times and handles relationships between data better than a relational DB. In this chapter, a motivation pattern for a possible scenario when an enterprise considers shifting from the relational DB to graph DB, is followed. The aim is to understand if the graph DB is reasonable for enhancing manufacturing collaboration in internal and partner networks.

Some weaknesses of the relational DB are listed in **chapter 3.1**. **Chapter 3.2** shows how the graph DB solves the most significant drawback. In **chapter 3.3**, the market is investigated, and it is realized that the market leaders use a graph DB. In **chapter 3.4**, a success story matching this study's use case is searched for. In **chapter 3.5**, it is recognized that the problem at hand needs to be understood first and that an analytical approach needs to be used to guide the decision. **Chapter 3.6** summarizes the findings with a specific focus on whether the graph DB is more dynamic, handles relationships better, and performs better in queries than the relational DB.

This chapter thus presents the findings from the literature for **RQ1**, and more specifically, the answers to these questions:

- How to choose between a relational DB and a graph DB implementation? What are the benefits and drawbacks, and what are the current trends?
- Are graph data models and graph DB implementations dynamic?
- Is managing and handling relationships among data elements easier in a graph DB than in a relational DB?

#### 3.1. The relational DB and its weaknesses

The popularity of the relational DB goes back to 1980 [9]. It is still frequently used, and for some use cases, it is still the best option for storing and organizing data [9]. Hurlburt et al. suggest that just as the television did not replace the radio, the graph DB will not replace the relational DB [32]. The relational DB is excellent for data aggregations [32]. Its high data integrity and consistency make it well-

suites for applications where the security of transactions needs to be ensured [17]. For example, extensive credit card processing systems that require reliable non-stop operation rely on the relational DB [32]. It is not suitable for use cases requiring frequent updates to the DB schema, [9] [32] [17] nor when data volumes proliferate [17].

Introducing a structural change is risky and can take weeks or even months [5]. One reason for the **high maintenance cost** is that the relational DB stores structured data in tables with predefined columns [9]. Each row in the table represents a record, and the intersection between the row and the column represents a specific data value [17]. There cannot be duplicate rows in a table, as it would cause ambiguity when executing queries [18]. To prevent duplicate rows, each table has a primary key consisting of one or several columns with values unique for every row [18]. The table has another name, relation, and a column in a link is the attribute [6]. Each relation represents records of a specific type [6]. The name of the relational DB can hence be misleading, where one wrongly assumes that the relation is the link between data elements and not the actual tables [9].

**Handling relationships is a significant weakness** in the relational DB [9] [39]. Instead of storing relationships, they are computed through expensive join operations in query executions [17]. The join operations are costly due to the underlying relational model which, in a query, builds a set of all possible answers before filtering to arrive at the correct solution [5]. With today's highly connected data needs, any enterprise failing to understand connections when making important data-driven decisions will lack crucial insight [9].

Bechberger and Perryman highlight that for every hundred **queries** used in a modern application, the relational DB can handle only eighty-eight queries [7]. The remaining twelve queries deal with complex data links and connections [7]. Especially the queries requiring investigations deeper than three hierarchical levels, will show the degradation in performance in a relational DB compared to a graph DB [39]. Robinson et al. [5], bring up an example for query times between an RDBMS and the Neo4j GDBMS to be according Figure 19. Query times for Neo4j remain stable no matter the hierarchy dept, while the RDBMS cannot deliver a query result on the hierarchy dept level five [5].

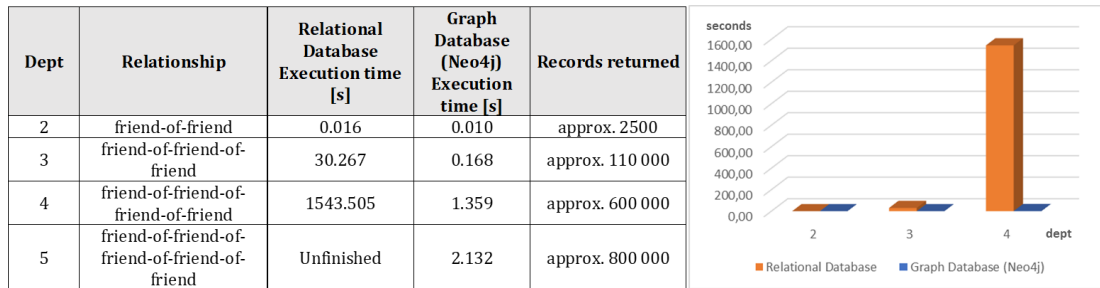


Figure 19. Comparison of execution times in a social network using RDBMS versus GDBMS (author's picture adopted from values in [5])

Another weakness of the relational DB is that **data mapping into tables is not how data exists in the real world** [9]. In the real world, data exist as objects and as relationships between these objects [9]. Cao et al. [39] warn that some knowledge about the data is lost when data is stored in a relational DB. When product component data is stored in a relational DB, you only understand which components the product consists of [39]. Holding the same components and their relationships in a graph DB ensures that the entire product structure is known [39]. In this case, the graph enables the product to be represented in a 3D modeling environment as it can be seen in real life [39]. From this connected structure, design changes are easy to manage, as a change in one component could, in real-time, indicate which other parts are affected by the change [39]. The product component nodes can be further connected to supply chain data to understand the supplier and customer data [39], which gives an even further real-world perception, where claims concerning specific components are easy to track [39].

### 3.2. Graph DB addressing the weakness of relational DB

The **graph DB** can be an alternative or an additional option if the relational DB does not manage well with increased attributes, more data, higher speed requirements in business agility or data accessibility, and significantly more connections between data elements [9]. The graph DB consists of two main elements, a node and an edge [9]. The node represents an entity, and the edge describes the relationship between two nodes [9]. When several nodes and edges are assembled, they form connected structures called graphs that define a specific problem domain [9]. In contrast to the relational DB, the graph DB is **explicitly**

**designed to handle and store relationships between data elements** [9]. In a graph DB, a relationship is seen as essential or even more important than the data element itself [7]. It is worth noting that the other NoSQL DB types visualized in Figure 5 are not explicitly designed to handle relationships [5].

The graph makes adding new nodes and edges easy when **adapting to new business requirements** [13]. Additions do not require data migrations as the original data and the purpose remain intact [5]. Bechberger and Perryman disagree and indicate that changes in a graph DB can still cause data migration needs. Therefore, they note the importance of not making changes because of poor design decisions and only due to business changes [7].

### 3.3. Market leaders using graph DB

*LinkedIn, Google, Facebook, and PayPal* are early adopters of graph DB, who today are market leaders who have formed their business value on data relationships [9] [17]. LinkedIn can be used as an example. They cover all their users with a graph. Hence, when browsing one's LinkedIn account, all different connection-level contacts and mutual connections can be seen in real-time [40]. Also, the giants in e-commerce, *Amazon.com* and *Wish.com*, utilize graphs to rapidly query information from a scattered and rapidly growing dynamic network of data to give users spot-on recommendations [40].

A white paper from October 2021 states that more than seventy-five percent of [Fortune 500](#) companies use graph DB technology [21]. Among these are:

1. Seven of the world's top ten retailers
2. Three of the top five aircraft manufacturers
3. Eight of the top ten insurance companies
4. All North America's top twenty banks
5. Eight of the top ten automakers
6. Three of the world's top five hotels
7. Seven of the top ten telecommunications companies

### 3.4. Use cases gaining success through graph DB

Despite the growing trend of graph DB utilization, further investigation is needed to understand if this is a suitable choice for our problem area. Webber and Robinson provide a list of five generic use cases where a graph DB brings benefits to any enterprise [13]:

1. Real-time fraud detection
2. Real-time recommendations to users
3. Master data management
4. Network and Information Technology Operations
5. Identity and accesses management

Saarela agrees with this list and adds use cases for compliance with regulations, analytics, digital asset management, context-aware services, semantic search, and situational awareness [40].

Reflecting on **chapter 1** and Wärtsilä' s desire to know if and how a GDBMS can enhance manufacturing collaboration in internal and partner networks, it could be assumed that all of the use cases mentioned above could be encountered in the requirements for these networks, making the graph DB appealing. For the scope of this study, only one use case is selected for further investigation. Master data management is chosen. This selection is based on the assumption that master data sharing for increased transparency, traceability, and quality improvements will be of vital interest in the manufacturing ecosystem.

**Master data** usually consists of data concerning customers, products, accounts, vendors, and partners [41]. It is highly dynamic and sharable data that is difficult to fit into a static and generic data model [41]. It is also challenging to assume that all master data could be physically stored in one location and that one system could serve all the needs in master data management [41]. Therefore, enterprises end up with separate systems covering different needs of master data management [41]. This creates a risk for information silos, where the data needed for decisions is not available in real-time [41].

Where data is stored is not relevant. Critical is the availability of consistent and meaningful views of master data, and that value can be derived from the data and its relationships [41]. Building relationships between the scattered master data

elements and achieving real-time query performance is seen as challenging and expensive in a relational DB [41]. The graph DB, with its characteristic of mastering relationships between data elements in a dynamic data structure, makes it an optimal choice for managing master data within an enterprise [41] [40] and beyond.

### 3.5. Analytical approach for deciding DB type

Choosing the graph DB over the relational DB based on reading success stories and use cases described on the Internet is, according to Bechberger and Perryman, not a good approach [7]. The risk is getting confused by drastic oversimplifications, such as, “everything is a graph problem” [7]. In contrast, there is a risk that the developers choose the familiar relational DB as a form of convenience or ignorance [7]. However, Robinson et al. note that moving from a well-established and well-known data platform to graph DB must indicate immediate and remarkable practical benefits in query performance, flexibility, and agility [5].

A graph DB is more elegant than a relational DB in problems needing recursive queries, different result types, or paths [7] [5]. An **analytical approach** is proposed to understand if the issue at hand holds these needs. The initial question in this analysis is: “What problem are we trying to solve?” [7]. Sorting this out creates an understanding of what data will be stored and how it will be retrieved [7]. Generalized, any problem fits into one of the following categories [7]:

- Selection/search
- Related or recursive data
- Aggregation
- Pattern matching
- Centrality, clustering, and influence

Figure 20 summarizes how Bechberger and Perryman describe these separate categories and how they are utilized for selecting between the relational DB and the graph DB.



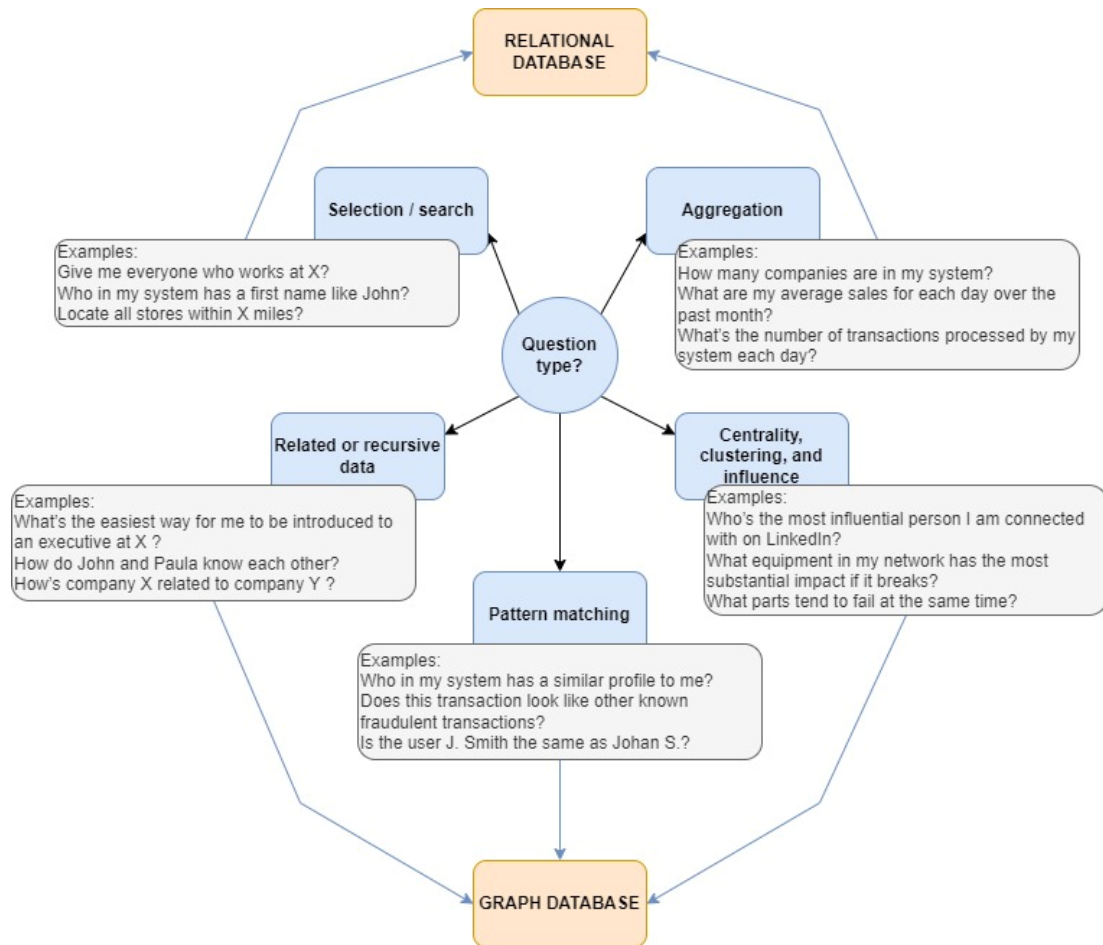


Figure 20. By sorting out the questions that will be answered based on the data in the DB, the correct DB type can be chosen (author's picture adopted from the text in [7])

If sorting out the category does not clarify which DB type to use, the decision tree created by Bechberger and Perryman, visualized in Figure 21, can be utilized [7]. Bechberger and Perryman have placed the most vital question first, and answering “yes” to this question directly indicates that the graph DB is the best choice [7]. Following this decision tree, the graph DB seems to be a good choice for manufacturing collaboration in internal and partner networks where the relations will play a vital part, and frequent evolvments in the systems can be expected. However, there is no clear indication of immediate and remarkable practical benefits in areas like query performance, flexibility, and agility. The specific problem in the domain of interest should according to Robinson et al. be sorted out before deciding to move from a familiar DB type to the graph DB [5].

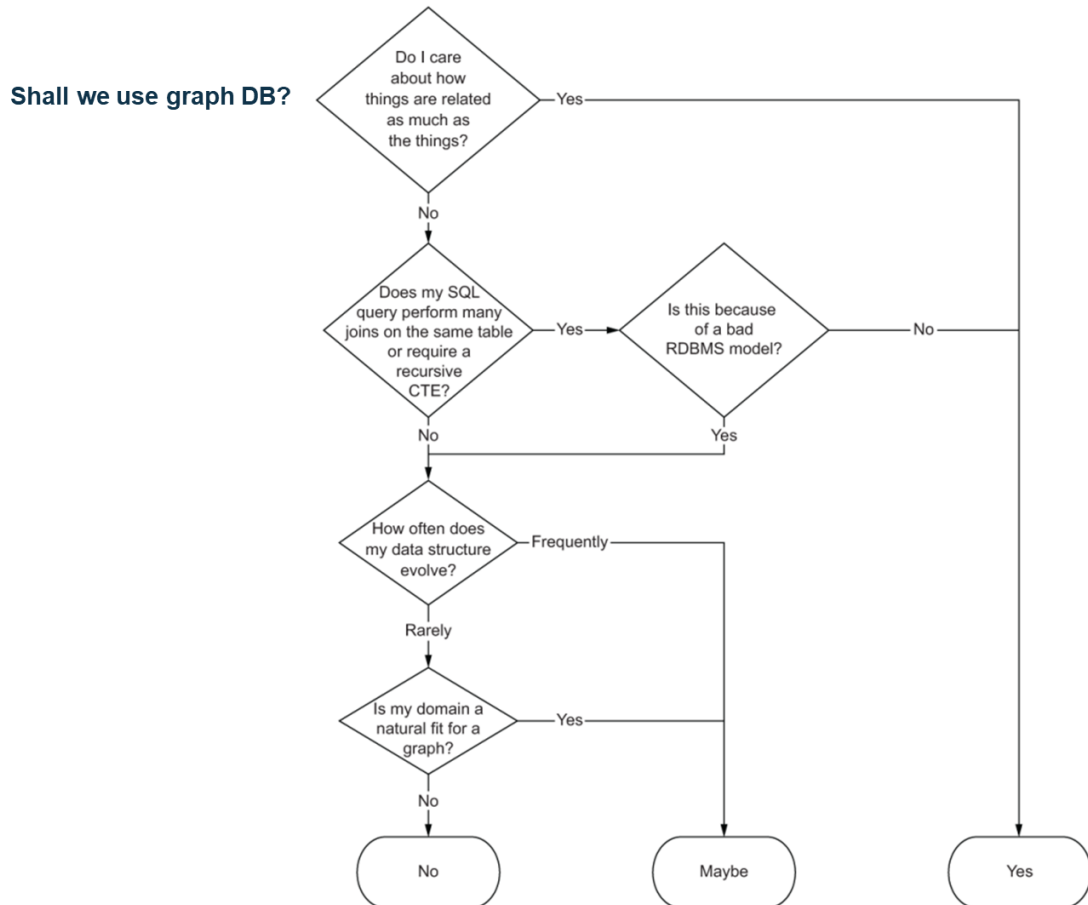


Figure 21. Decision tree to decide between a relational DB or a graph DB (Picture source: [7])

### 3.6. Dynamic data model

The data models used today need to be flexible and scalable to respond to the changing demands from within the company and beyond [4]. They need to be designed to handle complex data and enable rapid insight from data connections [7]. The data models must be dynamic, allowing the processes to be carried out with different approaches but still manage situations where the result is combined and presented [4]. This chapter investigates arguments to understand if the data models produced for the relational DB versus the graph DB are dynamic.

The data models produced for a relational DB are rigid table constructions designed to reflect the data needs of a business at a certain point in time [4]. Their response to change is weak and requires expensive configurations to reflect

changes in business needs [4] [5]. When the data models are designed, there is usually a need to predict future needs and integrate these requirements into the models [4]. When the design process is performed, it is often wrongly assumed that the business is committed similarly throughout the company [4]. If differences exist, multiple data models may be required [4].

The relational DB design approach is challenging in a world where change is constant and the future difficult to predict [4]. Robinson et al. [5] recommend using the graph data modeling approach instead of moving through the relational DB design process of translating a graph representation into tables, which is done when the ER model is transformed into the relational model. They claim that each step in the design process, *conceptual data model* → *logical data model* → *physical data model*, increases the gap between the conceptual world and the model understood by business stakeholders versus how the DB is implemented. This gap causes challenges when business needs are changed and must be translated into concrete actions for the DB implementation. The relational DB design phases are slow and cause the system to lag behind the evolution of the business. Figure 22 visualizes the gap and the increased risk of misunderstandings between conceptual and DB implementation when performing data modeling for a relational DB versus a graph DB.

The graph data model approach is simple, intuitive, and business stakeholder friendly [5]. The intuitive graph representation of the conceptual world remains as a graph no matter the design phase [5]. For Bechberger and Perryman [7], this approach is the solution to fewer design mistakes and easier data model changes. Adding new elements to the graph model and DB implementation is easy and straightforward [5] [7]. It remains unclear if data migrations are needed when a change is made in the data model. Robinson et al. states that no costly and risky data migrations are required [5]. At the same time, Bechberger et al. notes that changes in the graph model need changes in the DB implementation, leading to code changes and some data migration [7].

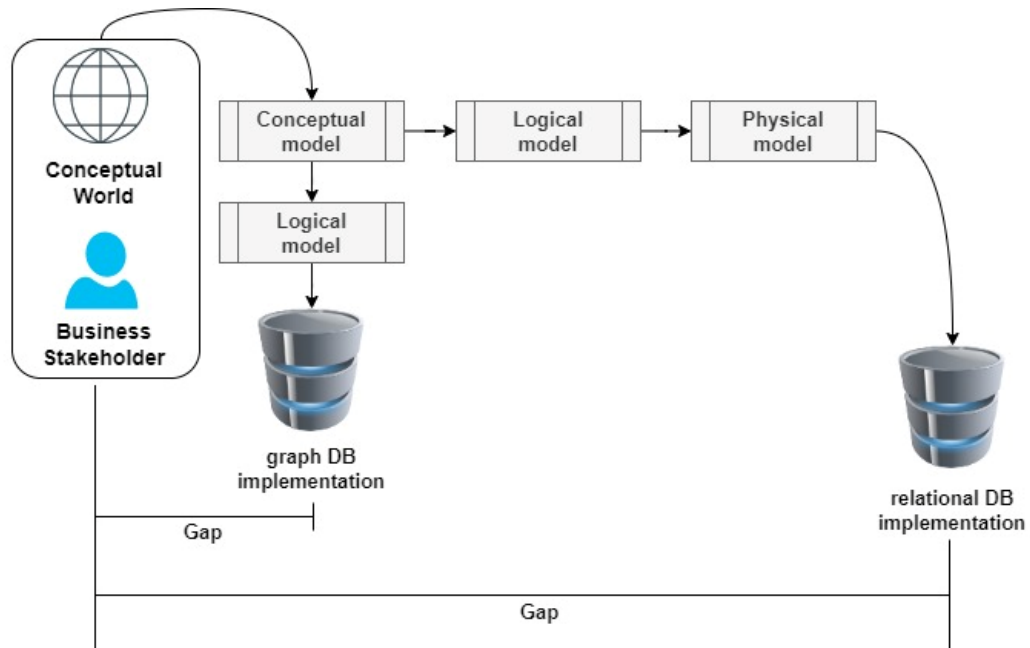


Figure 22. The gap between the conceptual world and the design steps for a relational DB versus a graph DB implementation (author's picture)

### 3.7. Conclusion

Chapter 3 discussed different situations where the relational DB is suited and where the graph DB is a better choice. It became clear that this decision should not be taken based on developers' preferred choice of DB type or by looking at success stories from use cases on the Internet. The problem at hand must be understood before selecting the DB type. Understanding which type of queries will be addressed on the data in the DB is vital. Mapping the questions according to the example in Figure 20 can help to choose the correct DB type. The DB type choice is not straightforward, and sometimes the optimal choice is a hybrid with several DB types.

The interest in a graph DB existed in Wäertsilä before the agreement for this thesis work was made. Based on the literature review, the graph DB seems to be the correct choice for enhancing manufacturing collaboration in internal and partner networks. However, a hybrid approach is the most realistic scenario for this type of enterprise network. With the hybrid system, the optimal data for a relational DB are stored in a relational DB, and the optimal data for a graph DB is stored in

a graph DB. Everything depends on the enterprise network's requirements and use cases.













When going through the scenario for why an enterprise would move to a graph DB from a relational DB, the characteristics listed in

Table 4 are recognized. In the table, the happy smiley indicates a positive aspect, the sad smiley indicates a downside, and the meaning of the neutral smiley is clarified in the table. In chapter 1, the requirement for dynamic data models and data models excelling in data queries involving joins were mentioned as two essential elements.

Based on the arguments in chapter 3.6, it is understood that the data model for a relational DB is not dynamic. The data model for a graph DB is dynamic and more intuitive than the relational data model. Hence, it is better understood by business stakeholders, making discussions and alignment with modification needs easier. The literature review did not find any investigations where it is measured how much more effort and time is required for relational versus graph data modeling.

Based on the literature review, handling relationships is a significant weakness in the relational DB [9] [39]. Instead of storing relationships, they are computed through expensive join operations [17]. Join operations become costly due to the underlying relational model that, in a query, builds a set of all possible answers before filtering to arrive at the correct solution [5]. The graph DB stores relationships directly in the graph structure and quickly returns a result for queries that would require joins between tables in a relational DB [5].

Table 4. Recognized benefits and drawbacks of the characteristics described in chapter 3

Characteristic	Relational DB	Graph DB
<b>High data integrity and consistency</b>	 <p>Relational DBs are known to be fully ACID compliant.</p>	 <p>Neo4j is entirely ACID compliant. However, this is not the case for all graph DBs. NoSQL DBs are generally only BASE compliant (Table 2).</p>
<b>Handling relationships</b>		
<b>Maintenance cost</b>		
<b>Query execution time</b>	 <p>Depending on the size of the DB and how many hierarchical levels need to be searched. Performance decline when more than three hierarchical levels.</p>	 <p>Quite stable query execution time, no matter the hierarchical level.</p>
<b>Intuitiveness (data mapping corresponds to how we perceive it in the real world)</b>		
<b>Handling the dynamic business environment and changing requirements</b>		

## 4. CASE STUDY

This chapter presents the case study consisting of the following:

1. Relational and graph data modeling experiment design and case study (to answer **RQ2**)
2. Graph data model implementation in Neo4j standalone desktop version (to answer **RQ3**).

The literature indicates that the relational data model's response to change is weak and requires expensive configurations to reflect changes in business needs [4] [5]. In contrast, the graph data model is described to be dynamic. It is easy to add new data elements when adapting to new business requirements in the graph data model [13]. Despite the dynamic capability being highlighted as a benefit of the graph data model, we note that research comparing the relational domain to the graph domain mainly focuses on DB query execution times and handling relationships between data elements. The literature review did not find any experiments investigating the difference in effort and time needed to implement and modify a relational data model versus a graph data model.

The data models implemented in this case study cover equal data needs. The case study is set up with an experimental design and analysis approach. The implementation level is the logical data model for a relational DB and a graph DB. The data modeling tools selected are the **ERDPlus** [42] for relational data modeling and the **arrows.app** [43] for graph data modeling. The data needs are understood from discussions with business stakeholders and an analysis of an engine's DBOP. The DBOP is available in an Excel file with a table of 9210 rows and 38 columns. The content of the DBOP is further explained in chapter 4.2.1.

To understand if the graph data model is more dynamic than the relational model, an experiment is planned and tested as a case study. The exact data needs were modeled as a relational data model and a graph data model. In the case study the author represents the subject. The limitation of having only one subject available creates the limitation of not running a statistically relevant experiment. In a future experiment ten to twenty Information Technology students would be a reasonable subject scope.



The dynamic capability is seen as an indirect measure. To calculate the dynamic capability, the following data is realistic to collect:

- The time needed for data analysis and discussions with business stakeholders to understand the data needs
- The data modeling time
- The number of data elements in the data model

In addition, the subjective qualitative and subjective analysis of the difficulty level of building the relational data model versus the graph data model gives valuable insight.

In the case study an independent domain expert reviewed the models to understand if the implementation meets the expectations of Wärtsilä. Quantitative implementation measures, such as query execution times, are beyond the scope of this case study and has neither been planned for in the experiment design.

The design of the experiment is described in **chapter 4.1**. **Chapter 4.2** describes the data modeling process and the resulting data models. **Chapter 4.3** describes the steps and results of the graph data model implementation in the Neo4j standalone desktop version.

## **4.1. Experimental design**

This chapter describes the outcome of the experimental planning stage. The chapter covers the reporting structure suggested by Wohlin et al. [44], Figure 23.

Experimental design	Describes the outcome of the experimental planning stage
Goals, hypotheses and variables	Presents the refined research objectives
Design	Define the type of experimental design
Subjects	Defines the methods used for subject sampling and group allocation
Objects	Defines what experimental objects were used
Instrumentation	Defines any guidelines and measurement instruments used
Data collection procedure	Defines the experimental schedule, timing and data collection procedures
Analysis procedure	Specifies the mathematical analysis model to be used
Evaluation of validity	Describes the validity of materials, procedures to ensure participants keep to the experimental method, and methods to ensure the reliability and validity of data collection methods and tools

Figure 23. The reporting structure for the experimental design (picture source: [44])

### 4.1.1. Goal definition

The experiment is motivated by the lack of evidence for the claim that the graph data model is more dynamic than the relational data model. Dynamic is seen as equal to the effectiveness and efficiency of creating a data model and later implementing changes to this model. The time for analysis and data modeling, together with a qualitative review of challenges faced during the implementation, indicate the implementation's efficiency. Effectiveness is a measure of how well the model meets the expected result. The best option in a dynamic environment is a model that is easy to implement and adapts to changes easily. If both the data modeler and the business stakeholder understand the implemented model, it is easier to discuss and align on needed changes.

The **object of study** is a relational data model and a graph data model. Often in DB design, three levels of data models are created. These levels are presented in chapter 2.4. This experiment is limited to the logical data model. With this selection, the specific requirements of DB providers are avoided, but it is still possible to identify a difference between the graph and the relational models.

The **purpose** of the experiment is to understand the dynamic capability of the relational data model and the graph data model.

The **perspective** is from the point of view of the author. Seeking to understand if the literature describing the graph data model to be more dynamic than the relational data model can be verified.

The **quality focus** for the **dynamic** capability is on the effectiveness and efficiency of implementing the selected data model types. The time it takes to understand the data needs and later implement the relational data model versus the graph data model is measured in minutes. Also, the number of elements in each model is counted. In the relational data model, the tables and relations between the tables are calculated. The number of nodes and edges for the graph data model is considered. The effectiveness and efficiency of the data modeling are computed by summing the time it takes for data analysis and modeling, and then dividing this total time by the number of elements in the data model.

**Context.** When the experiment is carried out three sequential steps are recommended. The first step is the data analysis and discussion with business stakeholders. The second step is relational data modeling and the third is graph data modeling. The data analysis and modeling shall be carried out in a disturbance-free environment. A maximum length of 90 minutes per session shall be set to ensure proper focus during the analysis and modeling. The date and time used per session shall be recorded. In the calculations, only each data model's complete analysis and data modeling time shall be used.

The goal summary was defined using the Wohlin et al. goal template, Figure 24.

Analyze <Object(s) of study>  
for the purpose of <Purpose>  
with respect to their <Quality focus>  
from the point of view of the <Perspective>  
in the context of <Context>.

Figure 24. Wohlin et al. goal template (Picture source: [44])

*Analyze the graph data model and relational data model dynamic capability  
for the purpose of evaluation  
with respect to their effectiveness and efficiency  
from the point of view of the author  
in the context of a subject, first analyzing the data needs and then modeling the  
relational and graph data model for an engine DBOP.*

### 4.1.2. Hypothesis formulation

The basis for the hypotheses is that the graph data model is assumed to be more dynamic than the relational data model. The null hypothesis defines the graph and relational data models as equally dynamic. The alternative hypothesis explains the graph data model as more dynamic than the relational data model. Table 5 presents the null and alternative hypotheses and measures needed in the experiment.

Table 5. Null hypothesis and alternative hypothesis, together with their mathematical formulation and measures needed for understanding the dynamic capability of the data models

The null hypothesis, H <sub>0</sub> :	The alternative hypothesis, H <sub>1</sub> :
<b>H<sub>0_create</sub></b> : Analyzing the data needs and implementing a graph data model requires as much effort and time as the corresponding relational data model.	<b>H<sub>1_create</sub></b> : Analyzing the data needs and implementing a graph data model requires less effort and time than the corresponding relational data model.
<b>Mathematical formulation:</b>	
<b>H<sub>0_create</sub></b> : $(\text{AnalysisTime}(\text{graph}) + \text{CreateTime}(\text{graph})) / \text{Elements}(\text{graph}) = (\text{AnalysisTime}(\text{relational}) + \text{CreateTime}(\text{relational})) / \text{Elements}(\text{relational})$  Simplified: <b>H<sub>0_create</sub></b> : $\text{CreateEff}(\text{graph}) = \text{CreateEff}(\text{relational})$	<b>H<sub>1_create</sub></b> : $(\text{AnalysisTime}(\text{graph}) + \text{CreateTime}(\text{graph})) / \text{Elements}(\text{graph}) > (\text{AnalysisTime}(\text{relational}) + \text{CreateTime}(\text{relational})) / \text{Elements}(\text{relational})$  Simplified: <b>H<sub>1_create</sub></b> : $\text{CreateEff}(\text{graph}) > \text{CreateEff}(\text{relational})$
<b>Measures needed:</b>	
<p><b>When the data needs are understood:</b>            During analysis and discussion sessions, the time used is recorded in minutes. When the model is completed, all session times are summed and registered as, <i>AnalysisTime</i>.</p> <p><b>When the models are implemented:</b>            During the modeling sessions, the time used is recorded in minutes. When the model is completed, all session times are summed and registered as, <i>CreateTime</i>.</p> <p><b>After model implementation:</b>            The number of elements in the data model is calculated when the model is complete. For the relational data model, the number of tables and the relations between the tables are counted. The number of nodes and edges for the graph data model are measured.</p> <p>The effectiveness and efficiency of implementing the data model for a specific data model type, <i>CreateEff</i>, is calculated with the formula also used in the mathematical formulation of the hypothesis:  <math display="block">\text{CreateEff} = (\text{AnalysisTime}(\text{modelType}) + \text{CreateTime}(\text{modelType})) / \text{Elements}(\text{modelType})</math> </p>	

### 4.1.3. Variables

The variables used in the experiment are summarized in Table 6. The data model type is the only independent variable. It has two nominal levels: a *graph* data model and a *relational* data model.

The variables of the subject experience in graph data modeling and relational data modeling are controlled. The subject's experience is mapped per modeling type and measured on an ordinal scale with the levels:

1. No prior experience.
2. Followed a course or read a book.
3. Less than six months of industrial experience.
4. More than six months of industrial experience.

The dependent variables are related to the time spent on data analysis and modeling and the number of elements in each data model. The time measured is objective. The decision of whether the model is correctly implemented is subjective. Also, the number of elements can be considered subjective. The reason is that fulfilling specific data needs can be modeled in numerous ways. To minimize the bias in these measurements, the 3NF normalized form for the relational model and guidelines on building a labeled property graph for the graph data model are followed. In determining if the model is correct, a qualitative review is performed with an independent and experienced data modeling expert and a business stakeholder.

The effectiveness and efficiency of the model are calculated as the summed data analysis and modeling time, divided by the number of elements in the data model. This measure gives an understanding of the dynamic capability of the data model type.

Table 6. Variables used in the experiment

Name	Values	Description
ModelType	{graph, relational}	The subject creates two alternative logical data model types: a <i>graph</i> data model and a <i>relational</i> data model.
GraphExp	Ordinal	The subject's experience with modeling graphs is measured on a four-level ordinal scale.
RelationalExp	Ordinal	The subject's experience with relational data modeling is measured on a four-level ordinal scale.
AnalysisTime	Integer	The total time subject uses when analyzing the data needed for a data model. The unit is minutes.
CreateTime	Integer	The total time subject uses when creating a data model. The unit is minutes.
Elements	Integer	The number of elements in a data model.
CreateEff	$(\text{AnalysisTime} + \text{CreateTime}) / \text{Elements}$	The effectiveness and efficiency of the implementation of the data model. The units are left out in the <i>CreateEff</i> calculation.

#### 4.1.4. Design

When designing the experiment, the hypothesis was used as a starting point to understand which statistical analysis to perform to reject the null hypothesis [44]. The experiment was designed as a set of tests from which the data needed for the statistical analysis was collected. General design principles of randomization, blocking, and balancing were followed in the design [44].

Randomization was used to fulfill the requirement of collecting the data for statistical analysis from independent random variables [44]. Randomization can be applied to the objects, subjects, and the order of the tests in the experiment [44].

The subjects for the experiment are recommended to be a natural random selection of ten to twenty M.Sc. Information Technology students at Åbo Akademi. The subjects perform data analysis and discussions with business stakeholders to understand the problem and to create the data models. The order in which the models are created could be randomized. Instead, a conscious decision to create the data model for which the subject has better experience first. This balances the

subject performance possibilities when creating the two alternative data models. Starting data modeling on an unfamiliar data set takes time. It is hence considered fair to take the hit in ramp-up time when modeling the more familiar modeling type. This way, undertaking the more unfamiliar data modeling type will only cause ramp-up time in the modeling approach and not the data set. Due to the small size of the experiment, no blocking was applied.

Wohlin et al. also present some frequently used experiment design types, which range from simple experiments with only one factor to more complex ones with many factors [44]:

- One factor with two treatments.
- One factor with more than two treatments.
- Two factors with two treatments.
- More than two factors, each with two treatments.

The factor is the independent variable on which treatments are applied [44]. The only factor is the *ModelType*. There are two treatments: *graph* and *relational*. This experiment's design type is one **factor with two treatments**. This type of experiment intends to compare two treatments against each other [44]. The dependent variables of this experiment are defined in chapter 4.1.3. The main interest is the *CreateEff*, which indicates the dynamic capability of the data model.

The most used approach for the one factor with two treatments experiment design type is to have a **completely randomized design** where the subject uses only one treatment. In this experiment, the same subject uses both treatments. This design type is defined to be a **paired comparison design** or a **crossover design**. The risk in the paired comparison design is that the subject utilizes experience from treatment one when applying the second treatment. This risk is identified as an opportunity. In case where the *relational* is more familiar, the *relational* is selected to be used as the first treatment. The experience from *relational* data modeling can then be utilized when applying the *graph* treatment.

### 4.1.5. The subject

The subjects for the experiment are recommended to be a natural random selection of ten to twenty M.Sc. Information Technology students at Åbo Akademi.

In the case study the subject is the author, who is an M.Sc. Information Technology student at Åbo Akademi, Vaasa. The author has experience level 4, *more than six months of industrial experience*, in relational data model design. The graph data modeling experience level is 2, *followed a course or read a book*. Level 2 in graph data modeling is acquired from the literature review made for this study.

### 4.1.6. The object

The **objects of the study** are a relational data model and a graph data model. Both are on a logical data model level and describe the same data scope. The data scope is the DBOP for an engine produced in the Wärtsilä STH delivery center in Vaasa, Finland.

### 4.1.7. Instrumentation

Wohlin et al. [44] recognize three types of instruments to be chosen in the planning phase of an experiment. These are objects, guidelines, and measurement instruments. The following are needed:

- The exact data scope will be used for the graph and relational data models.
- Mapping the experience level of the subject.
- Basic understanding of data modeling concepts, chapter 2.
- Selection of and familiarization with data modeling tools.
- A timing watch for measuring the length of the data modeling sessions.

The data modeling tools selected are **ERDPlus** [42] for relational data modeling and **arrows.app** [43] for graph data modeling. These were chosen because both are free web-based data modeling tools. Both are easy to use due to their graphical modeling capability which does not require specific data modeling



language skills. Both offer the possibility of exporting the created data models as query commands to be used when a DB is created. The query language used in the export by ERDPlus is SQL [42]. Arrows.app exports query commands as Cypher clauses [43].

#### 4.1.8. Data collection

The case study to evaluate the experiment design was carried out during the fall of 2022. The measurements needed and the data collection approach is defined in Table 6.

#### 4.1.9. Analysis procedure

The mathematical analysis model is selected based on the experiment design type. The condition of the experiment hypothesis is accepted or rejected based on the observed p-values. The p-value is the lowest possible significance that can reject the null hypothesis. [44]

In chapter 4.1.4, the design type was specified to be **paired comparison design**. Examples of analytical models suitable for the paired comparison design are a **paired t-test**, a **sign test**, and a **Wilcoxon** [44]. The paired t-test is a parametric test that requires some of the parameters involved in the test to be normally distributed and the values to be on an interval scale [44]. The sign test and the Wilcoxon are non-parametric tests that do not require a specific distribution of the involved parameters [44]. The analyzed parameters (time in minutes, number of elements, effectiveness, and efficiency) are on a ratio scale, and normal distributions are not guaranteed. Based on this, the **Wilcoxon** test was selected for the analysis. A significance level of 0.05 was chosen to consider the result significant. In other words, the probability of not getting a random result was 95%.

### 4.1.10. Evaluation of validity

A valid result can be ensured by considering the experiment's validity already in the planning stage [44]. Wolin et al. [44] describe different types of threats to the validity of an experiment.

**Conclusion** validity concerns correct conclusions concerning the relationship between the treatment and the experiment result [44]. Possible problems are choosing the wrong statistical analysis or performing mathematical calculations wrong [44]. The conclusion validity also highly depends on the data quality used in the calculations [44]. **Internal** validity addresses issues where some uncontrolled factors affect the result of the experiment [44]. **Construct** validity discusses problems with how the experiment is designed [44]. **External** validity concerns if we can generalize the experiment to other environments, subjects, and contexts [44].

Validity in the experiment can be ensured by:

- Noting the importance of not searching for a specific experiment result.
- Having an independent data modeling expert verify the correctness of the data models.
- Including the time-consuming task of understanding the data needs as a measure in the experiment.
- Understanding the basic concepts of relational data modeling and graph data modeling. This is ensured by performing the background study and literature review before the experiment.
- Recognizing the need to map the experience of the subjects. This will help understand if the result is screwed due to unbalanced relational and graph data modeling skill levels.
- Disturbances are eliminated by keeping the experiment in a quiet environment and not allowing more than 90 minutes for each data analysis or modeling session.

The generalization of the result is limited to relational and graph data models. Understanding the dynamic capability of other data models needs separate investigation. Additionally, the limitation to the logical data model shall be noted. More data model levels, like conceptual and physical models, would be included

in an actual DB design and implementation scenario. An increased amount of data models increases the complexity and data modeling time.

Generalizing the result to an industrial setting with more experienced subjects cause a situation where the time needed to carry out the data analysis and modeling is reduced. Also, the data scope and cooperation with business stakeholders affect the result of the experiment. The data analysis time is reduced if the business stakeholders communicate their needs. While more data analysis time is needed in case discussions with business stakeholders are minimal. It is expected that the ratio of the dynamic capabilities of the relational data model versus the graph data model remains the same no matter the data in the scope and the involvement of the business stakeholders.

## 4.2. Data modeling

In our case study an incremental data modeling process is used where the steps described in chapter 2.4 are followed. The design process starts with understanding the problem and business needs. This is done through discussions with business stakeholders and analyzing any relevant material. The second step is to create the conceptual data model. We do not create a conceptual model but utilize an existing Excel file with a table describing the hierarchical structure of an engine DBOP. The third step is to make the logical data model. As the aim is to implement the graph data model in Neo4j, the labeled property graph is selected as the graph model type. For the relational data model, no specific RDBMS provider needs are considered. Still, we decided on a 3NF-normalized relational data model. In the 3NF normalized data model, we aim to build tables with a realistic minimum of duplicated data. At the same time also reduces the number of null values. The data model correctness is verified against the data needs and through an evaluation by an independent data modeling expert and business stakeholder feedback.

The remaining part of this chapter covers the result of the process in Figure 25. The black text describes what was done and the blue text the measurements collected from evaluating the experiment design. The variables are explained in chapter 4.1.3. The measures are explained in chapter 4.1.2. **Chapter 4.2.1** presents the data modeling needs. **Chapter 4.2.2** presents the decisions made

when creating the relational and graph logical data models, including modifications to fulfill the expectations of the independent data modeling expert and business stakeholder.

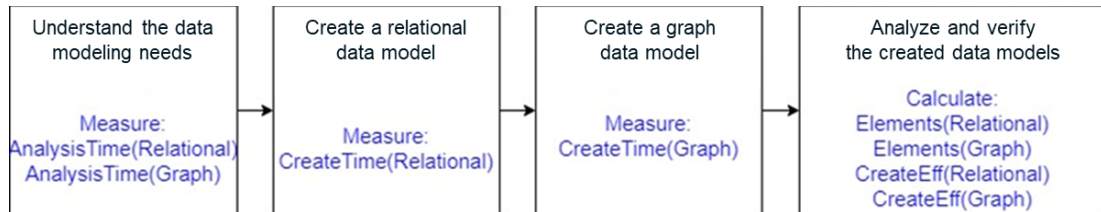


Figure 25. Data modeling process, including the measures and calculations needed for the experiment  
(Author's picture)

### 4.2.1. Understanding data modeling needs

An ideal scenario for data modeling is where the data modeler and business stakeholders discuss and align on the data needs. Discussions with business stakeholders are minimal in this experiment. Instead, focus is put on understanding the data needs from the DBOP Excel file. The business stakeholder describes the DBOP as the manufacturing steps of an engine built at the Wärtsilä STH manufacturing site in Vaasa. The DBOP is engine configuration specific and matches the needs of a unique engine. Other variations in the DBOP are due to the layout of the manufacturing site.

In Figure 26 to Figure 28, the 29 first rows of the DBOP can be seen. The data is obfuscated not to disclose company-sensitive data. According to the business stakeholder, the key column in the DBOP Excel file is the *StrLevel*. In contrast, the data modeling expert selected the *TcType* column. Further explanations of the DBOP were not given. Due to this minimal input that does not provide much guidance for the data modeling, only 10 minutes from the discussions with business stakeholders were recorded to the *AnalysisTime* variable for both the relational and graph treatment.

The DBOP Excel file data analysis took place on the 17<sup>th</sup> – 20<sup>th</sup> of August 2022. The total analysis time was 597 minutes. During the analysis, it was not easy to distinguish which part was dedicated to the relational versus the graph data model needs. The data modeling phase, described in chapter 4.2.2, revealed that the data analysis is ninety percent focused on the requirements for the relational

data model. The remaining ten percent focused on the needs for the graph data model, the investigation concerning the hierarchical structure of the DBOP. Based on this understanding, the 537 minutes were added to the *AnalysisTime(relational)* and 60 minutes to the *AnalysisTime(graph)*. When adding the 10 minutes of discussion with the business stakeholder, the total *AnalysisTime(relational)* is 547 minutes, and *AnalysisTime(graph)* is 70 minutes.

A	B	D	E	F	G	H	K	L	M	O	P	Q	R
ParentID	ParentRevID	StrLevel	Seq Nr	Qty	ID	RevID	Name	ICType	ReleaseDate	DocID	Description	Plant	EngineNumber
1	root	0	0	1	XAAC11111	A	DBOP-PRCP-W12V46TSDP A CDA A35 Power Supply SP/05555.3	WPlant Process Revision	2022-02-17 08:16:00.000	NULL	DBOP-PRCP-W12V46TSDP A CDA A35 Power Supply SP/05555.3	NULL	XAAC121212
2	XAAC11111	1	10	1	XAAC22222	A	D46-F106-PLANT	WPlant Process Revision	2022-02-17 08:16:00.000	NULL	D46-F106-PLANT	NULL	NULL
3	XAAC22222	2	10	1	XAAC33333	-	PRODUCT PREPARATION AND TESTING	WZone Process Revision	2022-02-17 08:16:00.000	NULL	PRODUCT MAIN ASSEMBLY	NULL	NULL
4	XAAC22222	2	20	1	XAAC739638	-	PRODUCT MAIN ASSEMBLY	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
5	XAAC22222	2	30	1	XAAC44444	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
6	XAAC22222	2	30	1	XAAC55555	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
7	XAAC33333	3	10	1	XAAC66666	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
8	XAAC33333	3	10	1	XAAC66666	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
9	XAAC44444	3	20	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
10	XAAC44444	3	30	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
11	XAAC44444	3	30	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
12	XAAC44444	3	40	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
13	XAAC44444	3	50	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
14	XAAC44444	3	60	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
15	XAAC44444	3	70	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
16	XAAC44444	3	80	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
17	XAAC44444	3	90	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
18	XAAC44444	3	100	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
19	XAAC44444	3	110	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
20	XAAC44444	3	120	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
21	XAAC44444	3	130	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
22	XAAC44444	3	140	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
23	XAAC44444	3	150	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
24	XAAC44444	3	160	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
25	XAAC44444	3	170	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
26	XAAC44444	3	180	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
27	XAAC44444	3	190	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
28	XAAC55555	4	10	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL
29	XAAC66666	4	10	1	XAAC739638	-	MODULE SUBASSEMBLES	WZone Process Revision	2022-02-17 08:16:00.000	NULL	MODULE SUBASSEMBLES	NULL	NULL

Figure 26. The 29 first rows of the DBOP Excel file. Part 1 of 3. The data is obfuscated. (Author's picture)

S	T	U	V	W	X	Y	Z	AA	AB	AC	AD
EngineAbbreviation	EngineDescription	Mag ParentItem	PhaseLevel	PlantLevel	ProcessType	QualityKey	AlternateProcess	CombiningParameter	ConsumedAssembly	PurchaseCode	ReallocationID
1	W12V46TSD F	W12V46TSD F A CDA A35 Power Supply SP/05555.3	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
2	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
3	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
6	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
7	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
8	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
9	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
10	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
11	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
12	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
13	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
14	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
15	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
16	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
17	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
18	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
19	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
20	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
21	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
22	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
23	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
24	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
25	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
26	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
27	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
28	NULL	NULL	NULL	CRG-ME-PREP-COUPLING PREPARATION FOR MAIN ENGINE	NULL	NULL	NULL	NULL	NULL	NULL	NULL
29	NULL	NULL	NULL	ERCS-SUB-ENGINE BLOCK + CRANKSHAFT MODULE (DCT)	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 27. The 29 first rows of the DBOP Excel file. Part 2 of 3. The data is obfuscated. (Author's picture)

	AE	AF	AG	AH	AI	AJ	AK	AL
1	RealizationsRevisionID	OwnIngrUser	OwnIngrGroup	PSA	MgrProcessRevision	MEtargItemID	SortSeq	DrawnPredation
2	NULL	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
3	NULL	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
4	NULL	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
5	NULL	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
6	NULL	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
7	-	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
8	-	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
9	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
10	-	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
11	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
12	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
13	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
14	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
15	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
16	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
17	-	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
18	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
19	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
20	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
21	-	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
22	-	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
23	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
24	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
25	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
26	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
27	A	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
28	NULL	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL
29	NULL	grpadm	4st Engines,Product Engineering,Delivery	NULL	NULL	NULL	NULL	NULL

Figure 28. The 29 first rows of the DBOP Excel file. Part 3 of 3. The data is obfuscated. (Author's picture)

In this chapter, the findings from the DBOP Excel analysis are presented. Due to company-specific and sensitive data, some data is obfuscated or hidden. Instead of focusing on specific data values, the aim is to:

- Check and handle duplicate data.
- Check and handle columns containing only null values.
- Understand relationships between data elements and how the hierarchical structure of the DBOP is constructed.
- Identify groups of data. The primary keys are mapped in the identified groups for the needs of the relational data model.

The size of the DPOB is 9210 rows and 38 columns. This large Excel file requires structured and efficient data analysis. **Python 3.6** was selected as the programming language. **Jupyter Notebook's version 4.5.6** web-based interactive computing platform was chosen as the programming environment.



This selection was made due to the author's preference. Building the data analysis with a Python script also enables code to be reused for other engine DBOP analyses.

Figure 29 shows the Python libraries used in the data analysis. **Pandas** is the most central library used. The data is stored in a Pandas data frame when reading the data from the Excel file. From the Pandas data frame, data is manipulated, viewed, and used. Pandas is an open-source data analysis and manipulation tool that is fast, powerful, and easy to use [45]. **NumPy** is used in Python array computations because it is fifty times faster than traditional Python lists [46]. NumPy also includes many supporting functions that make working with arrays easy [46]. **Matplotlib**, **Seaborn**, **NetworkX**, and **Pydot** are used to visualize data. Table 7 summarizes the library versions used in the analysis.

```
import numpy as np #arrays
import pandas as pd # data processing
import matplotlib.pyplot as plt #plotting
import seaborn as sns #plotting
import networkx as nx #graph presentation
import pydot
from networkx.drawing.nx_pydot import graphviz_layout
```

Figure 29. Libraries used in the data analysis (Author's code)

Table 7. Library versions used in the data analysis

Library	Version
<b>Numpy</b>	1.19.5
<b>Pandas</b>	0.25.3
<b>Matplotlib</b>	3.2.2
<b>Seaborn</b>	0.11.2
<b>NetworkX</b>	2.5.1
<b>Pydot</b>	1.4.2

Figure 30 shows the code for viewing the DBOP data frame info. In the result, Figure 31, the number of rows and columns in the data frame and the name and datatype of each column can be seen. The number of non-null values per a specific column is also presented. Five columns in the data frame contain only null values. These columns are:

- *occuid*
- *Plant*
- *CombidingParameter*

- *MfgProcessRevision*
- *METargetItemID*

The columns without values do not bring value to the data analysis or the data model and are thus removed. Columns marked as irrelevant by the business stakeholders are also removed from the data frame. These are:

- *ParentRevUID*
- *RevUID*
- *ReleaseStatus*
- *OccType*

```
print("A list of all column names in the DPOB dataframe. ")  
print("Including a summary of number of non-null values.\n")  
  
dfDBOP.info()
```

Figure 30. The Pandas info method is utilized for viewing info about the DBOP data frame (Author's code)

```

A list of all column names in the DPOB dataframe.
Including a summary of number of non-null values.

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9210 entries, 0 to 9209
Data columns (total 38 columns):
ParentID                9210 non-null object
ParentRevID             9209 non-null object
ParentRevUID            9210 non-null object
StrLevel                9210 non-null int64
Seq_Nr                  9210 non-null int64
Qty                     9210 non-null object
ID                      9210 non-null object
RevID                   9210 non-null object
RevUID                  9210 non-null object
ReleaseStatus           8202 non-null object
Name                    9210 non-null object
TcType                  9210 non-null object
ReleaseDate             8187 non-null object
occpuid                 0 non-null float64
OccType                 4182 non-null object
Description              7030 non-null object
Plant                   0 non-null float64
EngineNumber            3 non-null object
EngineAbbreviation      3 non-null object
EngineDescription       3 non-null object
Ma9_ParentItem          4072 non-null object
PhaseLevel              370 non-null object
PlantLevel              331 non-null object
ProcessType             331 non-null object
QualityKey              73 non-null object
AlternateProcess        331 non-null object
CombiningParameter      0 non-null float64
ConsumedAssembly        20 non-null object
PurchaseCode            46 non-null object
RealizationID           21 non-null object
RealizationRevisionID   21 non-null object
OwningUser              9208 non-null object
OwningGroup             9208 non-null object
PSA                     331 non-null object
MfgProcessRevision     0 non-null float64
METargetItemID         0 non-null float64
SortString              333 non-null object
DrawingRelation         257 non-null object
dtypes: float64(5), int64(2), object(31)
memory usage: 2.7+ MB

```

Figure 31. Info about the DBOP data frame (Author's code)

Before starting a more detailed data analysis, it was checked if the DBOP data frame has duplicate data rows. The code and result for this check are presented

in Figure 32. The result shows that there are no duplicate rows in the data frame that would need to be addressed.

```
dfDuplicateDataDBOP = dfDBOP[dfDBOP.duplicated()]

#visualizing how many duplicate rows were found in format (rows, columns)
print('Number of duplicated rows in DBOP: ', dfDuplicateDataDBOP.shape[0]);

Number of duplicated rows in DBOP: 0
```

Figure 32. Code and result to check if duplicate data rows exist in the DBOP data frame. (Author's code)

A contradicting message concerning a pivotal column to use in the data analysis was received in the initial discussions with the business stakeholder and the data modeling expert. One of them marked the *StrLevel* as the essential column, while the other specified the *TcType* as important. Hence, it is relevant to understand if there is a direct relationship between these two columns.

A relationship is confirmed if a unique value of *TcType* returns a unique value of *StrLevel* and if a unique value of *StrLevel* returns a unique value of *TcType*. Figure 33 shows the code for a function plotting how many unique values of *columnName* each *group* value has. Figure 34 is the result of calling this function to understand how many *TcType* each *StrLevel* has. The result shows that there are values of *StrLevel* returning more than one *TcType* value. Hence, the *TcType* is not dependent on the *StrLevel*.

```
def uniqueValueGraphWithTicks(dataframeName, group, columnName, graphTitle, rotationValue):
    dataframeName.groupby(group)[columnName].nunique().plot.bar(rot=0)

    plt.title(graphTitle, size=15)
    plt.ylabel('Count')
    plt.xticks(rotation=rotationValue)
```

Figure 33. Function for plotting how many values for the column given in the *columnName* argument exist for each column included in the *group* argument in the data frame given the *dataframeName* argument. The *graphTitle* is the argument for the title of the plot. The *rotationValue* specifies which direction the x-axis ticks shall have. (Author's code)

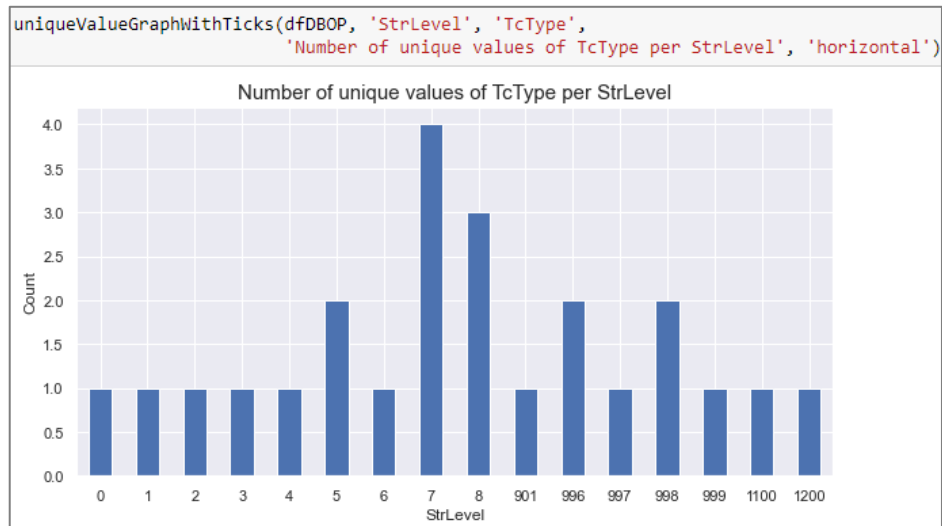


Figure 34. Number of unique values of TcType per StrLevel (Author's code)

Figure 35 shows how many unique values of *StrLevel* each *TcType* value has. The result indicates that five *TcType* values return more than one unique *StrLevel* value. Based on the outcome, it can be concluded that the *StrLevel* is not dependent on the *TcType*, and there is no direct relationship between the *TcType* and *StrLevel*.

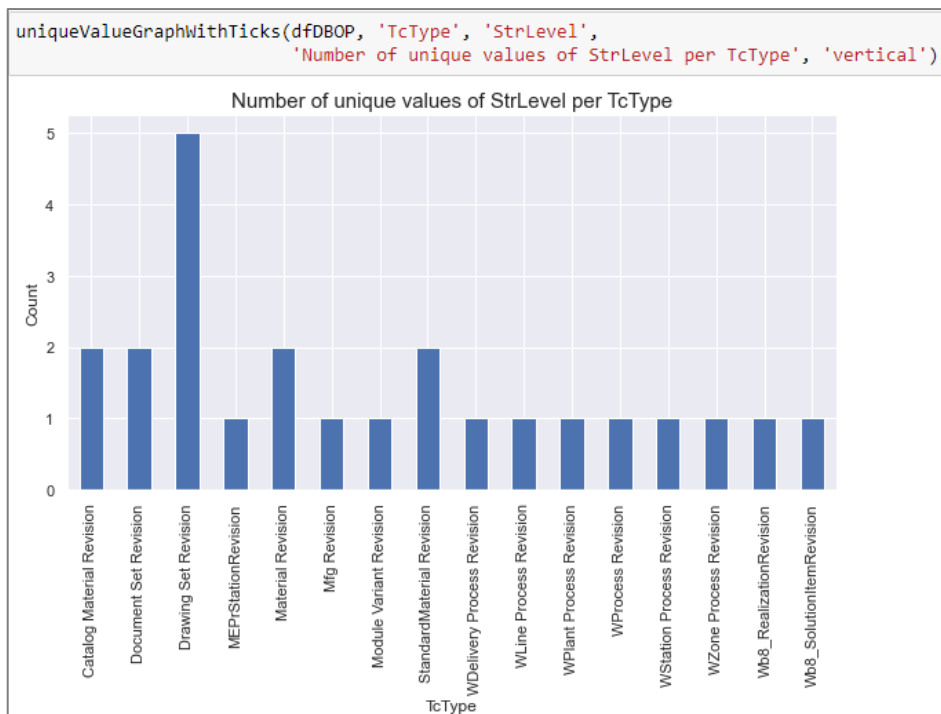


Figure 35. Number of unique values of StrLevel per TcType (Author's code)

By looking at the column names, it was assumed that:

- an *OwningUser* always belongs to the same *OwningGroup*
- an *OwningGroup* has one or many *OwningUsers*

Through investigation, this assumption was proved to be partly false. Figure 37 shows that an *OwningUser* can belong to eight *OwningGroups*. Figure 38 confirms that an *OwningGroup* can have one or many *OwningUsers*. The function called when creating the graphs in Figure 37 and Figure 38 is visible in Figure 36.

The *OwningUser* and *OwningGroup* values are hidden for confidentiality reasons. The *OwningUser* values represent the user identification in the *Surname, Forename* format, or as a code like *grpadm*. For the relational data model, where normalization rules are followed, consideration to split the *forename* and *surname* into separate columns is needed. The *OwningGroup* is a company organizational code in text format.

```
def uniqueValueGraph(dataframeName, group, columnName, graphTitle):
    dataframeName.groupby(group)[columnName].nunique().plot.line(rot=0)

    plt.title(graphTitle, size=15)
    plt.ylabel('Count')

    #changing the y-axis range from decimals to integers
    locs, labels = plt.yticks()
    yint = []

    for each in locs:
        yint.append(int(each))
    plt.yticks(yint)

    plt.xticks([]) #hiding x-axis ticks
```

Figure 36. Function for plotting how many values for the column in the *columnName* argument exist for each column included in the *group* argument in the data frame in the *dataframeName* argument. The *graphTitle* argument specifies the title of the plot. (Author's code)

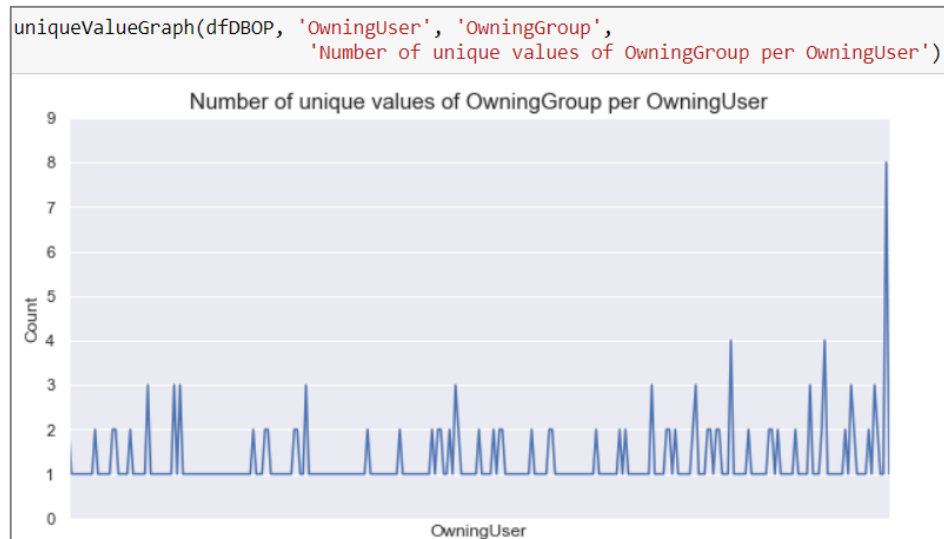


Figure 37. The number of unique values of OwningGroup per OwningUser (Author's code)

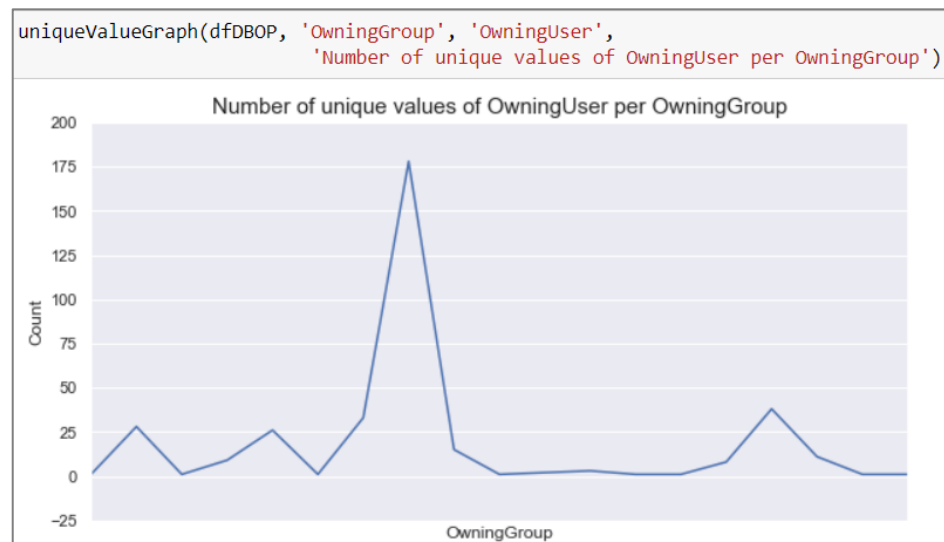


Figure 38. The number of unique values of OwningUser per OwningGroup (Author's code)

By looking at the column names and order of the columns, it is assumed that:

- *ParentRevID* is the revision of the *ParentID*
- *RevID* is the revision of the *ID*
- *RealizationRevisionID* is the revision of the *RealizationID*

Storing a revision value for the id calls for the understanding that an id can have several revisions. Figure 39 shows that the DBOP data frame has 2395 unique *ParentID* values and 11 unique *ParentRevID* values. The function in Figure 36 checks the number of unique values of *ParentRevIDs* for each *ParentID*. Figure 40 shows that most of the *ParentID* values have one *ParentRevID*. Looking carefully

at the graph in Figure 40, a line drop to *Count* level 0 can be seen on the right edge. This drop is for the *ParentID* value equal to the *root*. The *root* is the first row in the DBOP Excel file and the only *ParentID* without a revision value.

```
print('Number of unique values ParentID: ', dfDBOP.ParentID.nunique())
print('Number of unique values ParentRevID: ', dfDBOP.ParentRevID.nunique())
```

Number of unique values ParentID: 2395  
Number of unique values ParentRevID: 11

Figure 39. The number of unique values of *ParentID* and *ParentRevID* in the DBOP data frame (Author's code)

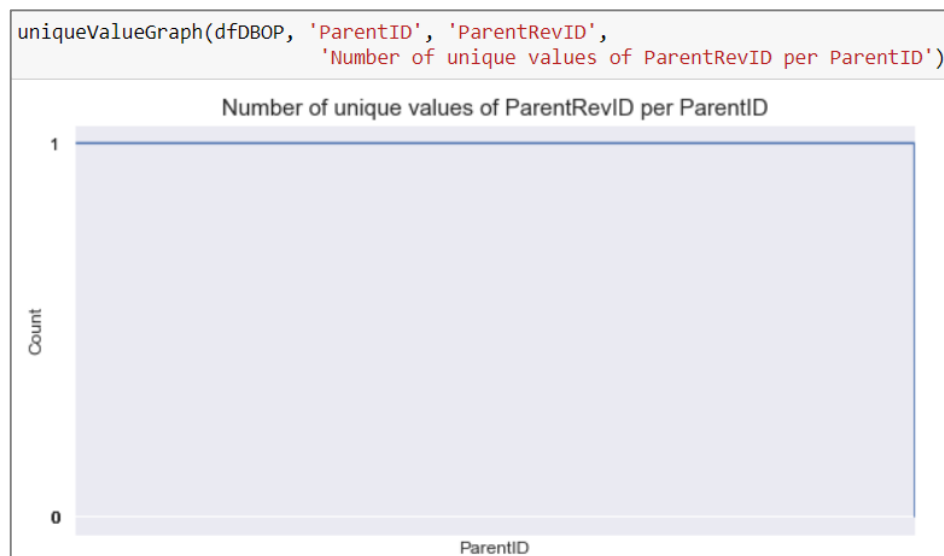


Figure 40. The number of unique values of *ParentRevID* per *ParentID* (Author's code)

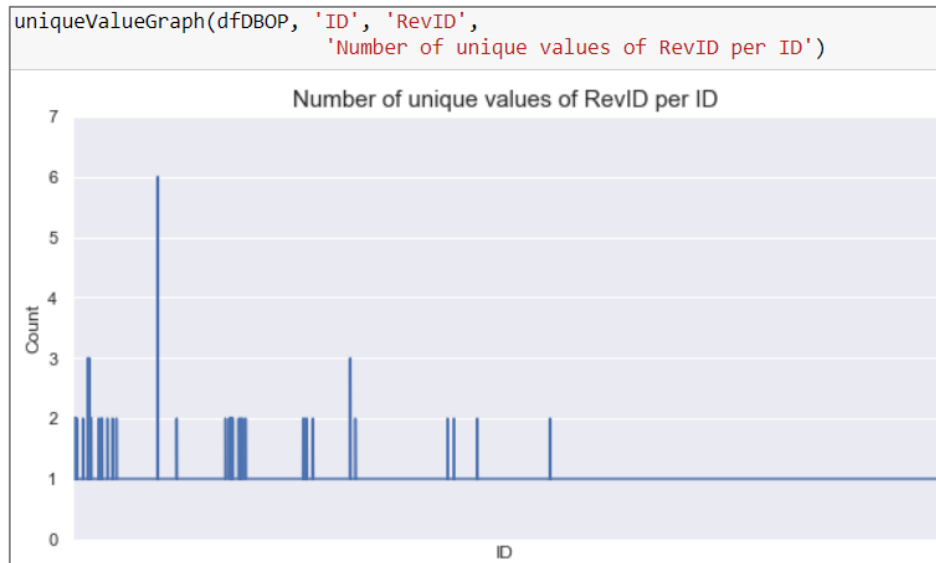
In the DBOP data frame, there are 5225 unique *ID* values and 18 unique *RevID* values, Figure 41. By calling the function in Figure 36, it is checked how many unique values of *RevID* each value of *ID* has. Figure 42 shows that most of the *ID* values have one *RevID*. Some *ID* values also have two, three, or six different *RevID* values.

```
print('Number of unique values ID: ', dfDBOP.ID.nunique())
print('Number of unique values RevID: ', dfDBOP.RevID.nunique())
```

Number of unique values ID: 5225  
Number of unique values RevID: 18

Figure 41. The number of unique values of *ID* and *RevID* in the DBOP data frame (Author's code)





From the analysis result of the three different id columns and their respective revision columns, it is recognized that each value of *ParentID* and *RealizationID* has one unique revision. It can be dangerous to assume that each id will always have one revision. Instead, it is believed that having decided to include the id and its revision in the DBOP, these columns will be considered as a pair. The situation with the *ID* and the *RevID* pair is a likely scenario for the *ParentID* and the *ParentRevID* pair and the *RealizationID* and the *RealizationRevisionID* pair.

The Id and the revision pairs are also recognized in the *Ma9\_ParentItem* column. In this column, the id and the revision values are combined and separated with a / character. This column has one or many ids and revision pairs in one value. Each pair is split with the | character. This column is hence multivalued. An example of values in the *Ma9\_ParentItem* column can be seen in Figure 45.

<b>Ma9_ParentItem</b>
PABA191919/A PABA181818/A
PABA252525/A
PABA077777/- PABA055555/A PAAF666666/A

Figure 45. *Ma9\_ParentItem* example values in the DBOP Excel file. The data has been obfuscated. (Author's picture)

A separate data frame is created from the DBOP data frame to investigate the *Ma9\_ParentItem*. Only rows with a *Ma9\_ParentItem* value are selected for the new data frame. Additionally, all columns containing only null values for the selected rows are dropped. The code and information about the new data frame are presented in Figure 46. The new data frame is named *dfMa9\_ParentItem* and contains 4072 rows and 15 columns.

```

dfMa9_ParentItem = dfDBOP[~dfDBOP['Ma9_ParentItem'].isnull()]
dfMa9_ParentItem

dfMa9_ParentItem = dfMa9_ParentItem[dfMa9_ParentItem.columns[~dfMa9_ParentItem.isnull().all()]]

print('Number of rows in the DBOP with Ma9_ParentItem value: ', dfMa9_ParentItem.shape[0], '\n')

dfMa9_ParentItem.info()

Number of rows in the DBOP with Ma9_ParentItem value: 4072

<class 'pandas.core.frame.DataFrame'>
Int64Index: 4072 entries, 1396 to 5539
Data columns (total 15 columns):
ParentID          4072 non-null object
ParentRevID       4072 non-null object
StrLevel          4072 non-null int64
Seq_Nr           4072 non-null int64
Qty               4072 non-null object
ID                4072 non-null object
RevID             4072 non-null object
Name              4072 non-null object
TcType           4072 non-null object
ReleaseDate       4072 non-null object
Description        3116 non-null object
Ma9_ParentItem    4072 non-null object
PurchaseCode      42 non-null object
OwningUser        4072 non-null object
OwningGroup       4072 non-null object
dtypes: int64(2), object(13)
memory usage: 509.0+ KB

```

Figure 46. The data frame where every row has a *Ma9\_ParentItem* value, and none of the columns contain only null values (Author's code)

For investigating the *Ma9\_ParentItem* id and the revision pairs, a dedicated data frame containing only the *Ma9\_ParentItem* column is created. In this data frame, the *Ma9\_ParentItem* id and revision pairs are separated into pairwise rows, and the id and revision values are split into separate columns. The code for this is visible in Figure 47. The code rows with a comment sign (#) in front of the *table\_Ma9\_ParentItem* have been used to output intermediate results during coding.

```

#creating a copy from dfMa9_ParentItem containing only Ma9_ParentItem column
table_Ma9_ParentItem = dfMa9_ParentItem[['Ma9_ParentItem']].copy()

#dropping duplicates
table_Ma9_ParentItem=table_Ma9_ParentItem.drop_duplicates()

#copying the Ma9_ParentItem column to keep the original as a reference to the DBOP datafarme,
#and have a copy of this that can be splitted to have one value / cell in the table
table_Ma9_ParentItem['Ma9_ParentItemSplit'] = table_Ma9_ParentItem['Ma9_ParentItem']
#table_Ma9_ParentItem

#splitting on |
#new rows are created for each split and the original Ma9_ParentItem is copied to each new row
table_Ma9_ParentItem['Ma9_ParentItem'] = table_Ma9_ParentItem['Ma9_ParentItem'].str.split('|')
table_Ma9_ParentItem = table_Ma9_ParentItem.explode('Ma9_ParentItem').reset_index(drop=True)
cols = list(table_Ma9_ParentItem.columns)
cols.append(cols.pop(cols.index('Ma9_ParentItemSplit')))
table_Ma9_ParentItem = table_Ma9_ParentItem[cols]

#table_Ma9_ParentItem

#splitting the ID from revision based on the separator string: /
table_Ma9_ParentItem[['Ma9_ParentItemID','Ma9_ParentItemRevID']]=table_Ma9_ParentItem.Ma9_ParentItem.str.split('/',
                                                                    expand=True)
#table_Ma9_ParentItem

#removing unnecessary column
table_Ma9_ParentItem.drop('Ma9_ParentItem', inplace=True, axis=1)

#renaming columns
table_Ma9_ParentItem.rename(columns = {'Ma9_ParentItemSplit':'Ma9_ParentItem'}, inplace = True)
table_Ma9_ParentItem

```

Figure 47. The code for creating a separate data frame where the *Ma9\_ParentItem* id and the revision pairs are split into individual rows, and the id and the revision values are separated into columns.

Figure 48 shows an example of running the code in Figure 47. The *Ma9\_ParentItem* column contains the original value. Comparing the split data to its initial value indicates how the data is split. Figure 49 shows the principle for the *Ma9\_ParentItem* data splitting.

Ma9_ParentItem	Ma9_ParentItemID	Ma9_ParentItemRevID
PABA198198/A PABA183183/A	PABA198198	A
PABA198198/A PABA183183/A	PABA183183	A
PABA251251/A	PABA251251	A
PAAF811811/-	PAAF811811	-
PAAF812812/C	PAAF812812	C
PABA777775/- PABA777755/A PAAF666666/A	PABA777775	-
PABA777775/- PABA777755/A PAAF666666/A	PABA777755	A
PABA777775/- PABA777755/A PAAF666666/A	PAAF666666	A
PABA215215/A PAAF667667/A	PABA215215	A
PABA215215/A PAAF667667/A	PAAF667667	A

Figure 48. Example result where the *Ma9\_ParentItem* id revision pairs are split into their own rows and separate columns are created for the id and the revision. The data has been obfuscated. (Author's picture)

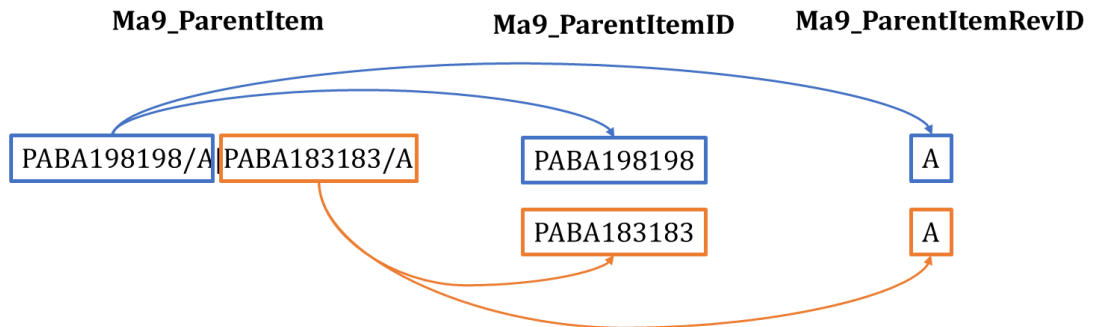


Figure 49. The principle for splitting the *Ma9\_ParentItem* values (Author's picture)

Figure 50 shows that there are 225 unique *Ma9\_ParentItemID* values and seven unique *Ma9\_ParentItemRevID* values in the *DBOP* data frame.

```
print('Number of unique values Ma9_ParentItemID: ', table_Ma9_ParentItem.Ma9_ParentItemID.nunique())
print('Number of unique values Ma9_ParentItemRevID: ', table_Ma9_ParentItem.Ma9_ParentItemRevID.nunique())
```

Number of unique values Ma9\_ParentItemID: 225  
Number of unique values Ma9\_ParentItemRevID: 7

Figure 50. The number of unique values of *Ma9\_ParentItemID* and *Ma9\_ParentItemRevID* in the *DBOP* data frame (Author's code)

The function in Figure 36 is used to check how many unique *Ma9\_ParentItemRevID* values each *Ma9\_ParentItemID* value has. Figure 51 shows that each *Ma9\_ParentItemID* value has one *Ma9\_ParentItemRevID*. With the same arguments as for the previous id and revision pairs, it can be concluded that an id and its revision need to be considered as a pair.

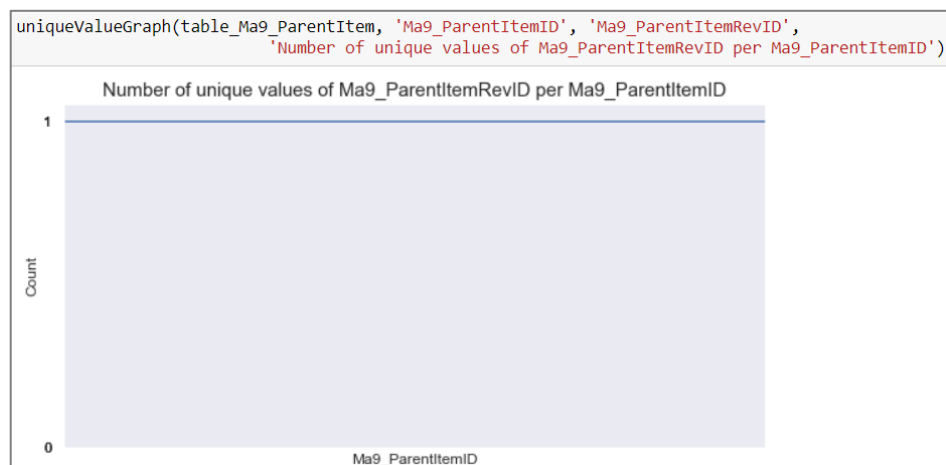


Figure 51. The number of unique values of *Ma9\_ParentItemRevID* per *Ma9\_ParentItemID* (Author's code)

A pattern between *ID* and *RevID* pairs and *ParentID* and *ParentRevID* pairs are recognized when outputting the first ten rows of the DBOP. Figure 52 shows the principle for the identified pattern. Where the *ID* and *RevID* pair at row zero becomes the *ParentID* and *ParentRevID* pair at row one. The *ID* and *RevID* in row one become the *ParentID* and *ParentRevID* pair in rows two, three, and four. This pattern is assumed to form the hierarchical structure in the DBOP. If the same *ParentID* and *ParentRevID* pair exists on several rows with the same *StrLevel* values, the *Seq\_Nr* value differs. From the *Ma9\_ParentItem* column name, a possible connection to the *ParentID* and *ParentRevID* pair is also assumed. The recognized hierarchical structure and possible relation to *Ma9\_ParentItem* values are investigated to understand the following:

- Does the hierarchical structure exist throughout the DBOP?
- Is the *Ma9\_ParentItem* involved in forming the hierarchical structure of the DBOP?

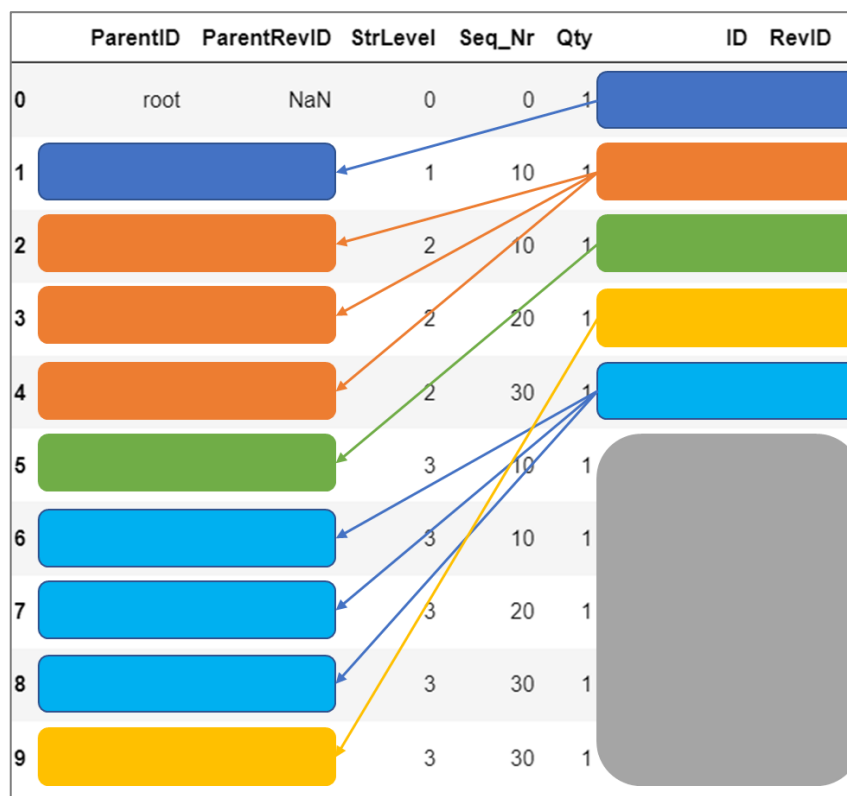


Figure 52. Principal sketch of the hierarchical structure in the DBOP. The gray area from row 5 forward contains *ID* and *RevID* pairs for which there are rows further down in the DBOP with matching *ParentID* and *ParentRevID* pairs. (Author's picture)

Figure 53 presents the code for creating two separate data frames from the DBOP data frame. One of them contains unique values of *ParentID* and *ParentRevID*

pairs. The other has unique values of *ID* and *RevID* pairs. These data frames are merged with the **Pandas merge** function, which automatically adds a column named *\_merge*. The *\_merge* column identifies if the id and revision pair are found in both original data frames, or only in one of these data frames. If the *\_merge* value is *left\_only*, the id and rev pair are only found in the *dfParentIDandRev* data frame, and if the *\_merge* value is *right\_only*, the id and rev pair are only found in the *dfIDandRev* data frame. The id and revision pairs where the *\_merge* value is *both* indicate that the id and revision pair is found in both data frames. When found in both data frames, it equals an identified link between the *ID* and *RevID* pair to the *ParentID* and *ParentRevID* pair. What the link means in this context can be seen in Figure 52.

```
#selecting only relevant columns from the DBOP dataframe and dropping duplicates in the new dataframe
dfParentIDandRev= dfDBOP[['ParentID', 'ParentRevID']].copy()
dfParentIDandRev.drop_duplicates(inplace =True)

#selecting only relevant columns from the DBOP dataframe and dropping duplicates in the new dataframe
dfIDandRev= dfDBOP[['ID', 'RevID']].copy()
dfIDandRev.drop_duplicates(inplace =True)

#renaming column names in the dataframe to perform an comparison between the content of the dfParentIDandRev and dfIDandRev
dfParentIDandRev.columns = ['id', 'rev']
dfIDandRev.columns = ['id', 'rev']

#merging the dataframes on id, rev columns. The merge function adds an additional column with the result of the comparison
m_differences = dfParentIDandRev.merge(dfIDandRev, on=['id', 'rev'], how='outer', suffixes=['', '_'], indicator=True)
m_differences
```

Figure 53. The code for investigating if an id, revision pair is found in both *ParentID* and *ParentRevID* column pair and *ID* and *RevID* column pair or only in either one of these column pairs (Author's code)

Figure 54 presents an example result of running the code in Figure 53. The id values in Figure 54 are obfuscated. If the *\_merge* value is *right\_only*, the particular *ID* and *RevID* pair form an end node in the hierarchical structure. This means that no further links to lower levels in the hierarchical structure exist from that specific *ID* and *RevID* pair. If the *\_merge* value is *left\_only*, it implies that the *ParentID* and *ParentRevID* pair has been linked from another column other than the *ID* and *RevID* pair. The rows whit *left\_only* require further investigation to identify another pattern in the hierarchical structure that, in addition to the *ID* and *RevID* pair, forms links to the *ParentID* and *ParentRevID* pair.

id	rev	_merge
root		left_only
XAAC749690	A	both
XAAC749627	A	both
XAAC749628	-	both
XAAC749648	-	both
DAAF521031	B	right_only
DAAF523346	B	right_only
DAAF527352	-	right_only
XAAC399556	-	right_only
XAAC539259	AB	right_only

Figure 54. An example of a result of investigating if the *id* and revision pair is found in both the *ParentID* and *ParentRevID* pair and the *ID* and *RevID* pair. The *id* values are obfuscated. (Author's picture)

With the code in Figure 55, the *ParentID* and *ParentRevID* pairs with no link from the *ID* and *RevID* pair are copied into a separate data frame. This data frame is named *dfParentIDandRevMissingIDandRev* and modified to exclude the *\_merge* column. Its *id* and *rev* column names are changed to *ParentID* and *ParentRevID*. The reason for changing the column names back to their original form is that **Pandas merge** requires that the column names in the comparison are equal in both data frames. The **Pandas merge** operation is now performed on the *dfParentIDandRevMissingIDandRev* and a copy of the original DBOP data frame.

```
#interested in ParentID ParentRevID combinations which do not have a Link from an ID, RevID combination
dfParentIDandRevMissingIDandRev = m_differences[m_differences['_merge'] == 'left_only']
dfParentIDandRevMissingIDandRev

#dropping the _merge column
dfParentIDandRevMissingIDandRev = dfParentIDandRevMissingIDandRev[['id', 'rev']].copy()

#changing the column names to fit the original dataframe
dfParentIDandRevMissingIDandRev.rename(columns = {'id':'ParentID', 'rev':'ParentRevID'}, inplace = True)
dfParentIDandRevMissingIDandRev
```

Figure 55. Creating a data frame containing only the *ParentID* and *ParentRevID* column pairs with no link from the *ID* and *RevID* column pairs. (Author's code)

The code for the second merge operation is shown in Figure 56. In this merge operation, the *\_merge* values equal to *both* are in focus. These are the DBOP rows where the *ParentID* and *ParentRevID* pair do not have a link from the *ID* and *RevID* pair. The rows with *\_merge* value *both* are stored in a data frame named *dfRowsWithNoLink\_IDandRevID\_ParentIDandParentRevID*. The row with *ParentID* equal to the *root* is dropped. This is the first node in the hierarchical structure for which no link from any previous node is expected. Also, the *\_merge* column and columns with only null values are dropped. The remaining size of the



`dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID` is 153 rows and 14 columns, Figure 57. The following investigates if the remaining 153 rows are linked from the `Ma9_ParentItem` column.

```
#creating copy of the original dataframe
dfDBOP_copy = dfDBOP

#comparing the dfDBOP_copy to dfParentIDandRevMissingIDandRev
m_differences2 = dfDBOP_copy.merge(dfParentIDandRevMissingIDandRev,
                                  on=['ParentID', 'ParentRevID'], how='outer', suffixes=['', '_'], indicator=True)
m_differences2

dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID = m_differences2[m_differences2['_merge'] == 'both']
dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID

#removing root as this is the initial node, not expected to have previous linkage from the ID, RevID columns
dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID = dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID[
    dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID['ParentID'] != 'root']
dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID

#removing the _merge column and columns with only null values
dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID.drop('_merge', inplace=True, axis=1)
dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID = dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID[
    ~dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID.columns[
        ~dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID.isnull().all()]]

dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID.info()
```

Figure 56. The code for selecting the rows in the DBOP data frame with no link from the ID and RevID column pair to the ParentID and ParentRevID column pair. The resulting data frame is named `dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID` (Author's code)

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 153 entries, 9031 to 9199
Data columns (total 14 columns):
ParentID          153 non-null object
ParentRevID       153 non-null object
StrLevel          153 non-null int64
Seq_Nr            153 non-null int64
Qty               153 non-null object
ID                153 non-null object
RevID             153 non-null object
Name              153 non-null object
TcType           153 non-null object
ReleaseDate       153 non-null object
Description        135 non-null object
OwningUser        153 non-null object
OwningGroup       153 non-null object
DrawingRelation   153 non-null object
dtypes: int64(2), object(12)
memory usage: 17.9+ KB
```

Figure 57. The `dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID` contain 153 rows and 14 columns. (Author's code)

Figure 58 presents the code where a third merge operation is performed. This time the merge is performed on copies of the data frame created in Figure 47 for the `Ma9_ParentItem` investigation and the data frame made in Figure 56. Only

columns containing an id and a revision are kept in the copied data frames. Duplicated values of the id and revision pairs are dropped before performing the **Pandas merge** operation.

```
#creating copy of the table_Ma9_ParentItem with only columns Ma9_ParentItemID and
#Ma9_ParentItemRevID included
#dropping duplicates
dfMa9_ParentItemIDandRev= table_Ma9_ParentItem[['Ma9_ParentItemID', 'Ma9_ParentItemRevID']].copy()
dfMa9_ParentItemIDandRev.drop_duplicates(inplace =True)

#dfMa9_ParentItemIDandRev

#creating copy of the dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID with only columns ParentID and
#ParentRevID included
#dropping duplicates
dfParentIDandRev2= dfRowsWithNoLink_IDandRevID_ParentIDandParentRevID[['ParentID', 'ParentRevID']].copy()
dfParentIDandRev2.drop_duplicates(inplace =True)

#dfParentIDandRev2

#renaming columns to match in order to make a comparison
dfMa9_ParentItemIDandRev.columns = ['id', 'rev']
dfParentIDandRev2.columns = ['id', 'rev']

#comparing dfParentIDandRev2 to dfMa9_ParentItemIDandRev on the columns id and rev
m_differences3 = dfParentIDandRev2.merge(dfMa9_ParentItemIDandRev, on=['id', 'rev'], how='outer',
                                         suffixes=['', '_'], indicator=True)
```

Figure 58. The code for creating a merged column of the *Ma9\_ParentItem* id and revision pairs and the *ParentID* and *ParentRevID* pairs that are missing links from the *ID* and *RevID* columns (Author's code)

Figure 59 presents the result of the merge operation. It can be seen that all the *ParentID* and *ParentRevID* pairs that did not have a connection from the *ID* and *RevID* pairs have a relationship from the *Ma9\_ParentItem*. Additionally, there are no remaining *ParentID* and *ParentRevID* pairs with a missing link from either the *ID* and *RevID* pair or the *Ma9\_ParentItem*. The investigation also reveals that

values with no link forward exists in the *Ma9\_ParentItem* column. As for the *ID* and *RevID* pairs, these form the so-called end nodes in the hierarchical structure.

```
#where _merge = both we have a connection between Ma9_ParentItem and ParentID, ParentRevID
dfRowsWithLink_Ma9_ParentItem_ParentIDandParentRevID = m_differences3[m_differences3['_merge'] == 'both']

print('Number of rows with a connection between Ma9_ParentItem and ParentID, ParentRevID:',
      dfRowsWithLink_Ma9_ParentItem_ParentIDandParentRevID.shape[0])

Number of rows with a connection between Ma9_ParentItem and ParentID, ParentRevID: 141

#where _merge = left_only there are still ParentID, ParentRevID combinations we have
#not found a link to in the dataframe
dfRowsWithNoLink_Ma9_ParentItem_ParentIDandParentRevID = m_differences3[m_differences3['_merge'] == 'left_only']

print('Number of rows where a connection to the ParentID, ParentRevID combinations is still missing:',
      dfRowsWithNoLink_Ma9_ParentItem_ParentIDandParentRevID.shape[0])

Number of rows where a connection to the ParentID, ParentRevID combinations is still missing: 0

#where _merge = right_only there are Ma9_ParentItem values that has no link forward
dfMa9_ParentItem_endNode = m_differences3[m_differences3['_merge'] == 'right_only']

print('Number of rows where a Ma9_ParentItem value has no link forward:',
      dfMa9_ParentItem_endNode.shape[0])

Number of rows where a Ma9_ParentItem value has no link forward: 84
```

Figure 59. Investigating the links from *Ma9\_ParentItem* to *ParentID* and *ParentRevID* shows that there are 141 unique id and revision pairs with links from the *Ma9\_ParentItemID* and *Ma9\_ParentItemRevID* column pair to the *ParentID* and *ParentRevID* column pair. In the DBOP structure, there are no *ParentID* and *ParentRevID* pairs that are not linked from either the *ID* and *RevID* pair or the *Ma9\_ParentItem*. In the DBOP, 84 unique *Ma9\_ParentItemID* and *Ma9\_ParentItemRevID* pairs have no link forward in the hierarchical structure. (Author's code)

The investigation of the hierarchical structure shows that a *ParentID* and a *ParentRevID* pair is linked from a higher hierarchical level of an *ID* and a *RevID* pair or an id and a revision pair in the *Ma9\_ParentItem* column. Figure 60 shows a simplified sketch of the recognized hierarchical structure.

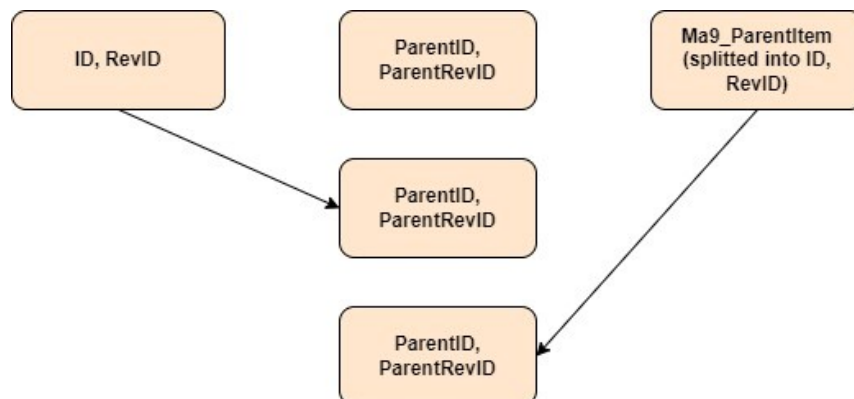


Figure 60. Simplified sketch of the principle of links to the *ParentID* and *ParentRevID* pairs. This forms the hierarchical structure of the DBOP. (Author's picture)

The entire hierarchical structure of the DBOP can be visualized with the **NetworkX** library. A graph with the dependencies between *ID* and *RevID* pairs and *ParentID* and *ParentRevID* pairs is first created. Then a graph with the dependencies between *Ma9\_ParentItem* and *ParentID* and *ParentRevID* pairs follows. These NetworkX graphs are then merged.

A copy of the DBOP data frame is created to prepare for the visualization. The code in Figure 61 creates a column connecting the *ID* and *RevID* pairs and a column combining the *ParentID* and *ParentRevID* pairs. Figure 62 shows how these new columns are utilized when creating the NetworkX graph of the DBOP hierarchical structure formed between the *ID* and *RevID* pairs and the *ParentID* and *ParentRevID* pairs. The result in Figure 63 shows a gap in the DBOP hierarchical structure.

```
#creating a dataframe where there is dedicated columns where ID and Revisions combined
dfDBOPForGraphs = dfDBOP
dfDBOPForGraphs['ParentIDandRev'] = dfDBOPForGraphs['ParentID'] + '/' + dfDBOPForGraphs['ParentRevID']
dfDBOPForGraphs['IDandRev'] = dfDBOPForGraphs['ID'] + '/' + dfDBOPForGraphs['RevID']
dfDBOPForGraphs['ParentIDandRev'].iloc[0] = 'root'
```

Figure 61. The code for creating a data frame of the DBOP data where the id and revision pairs are combined in a separate column for both ID and RevID pairs and the ParentID and ParentRevID pairs (Author's code)

```
G_DBOP_ParentIDLink = nx.from_pandas_edgelist(
    df = dfDBOPForGraphs,
    source='IDandRev',
    target='ParentIDandRev')

G_DBOP_ParentIDLink_color_map = []

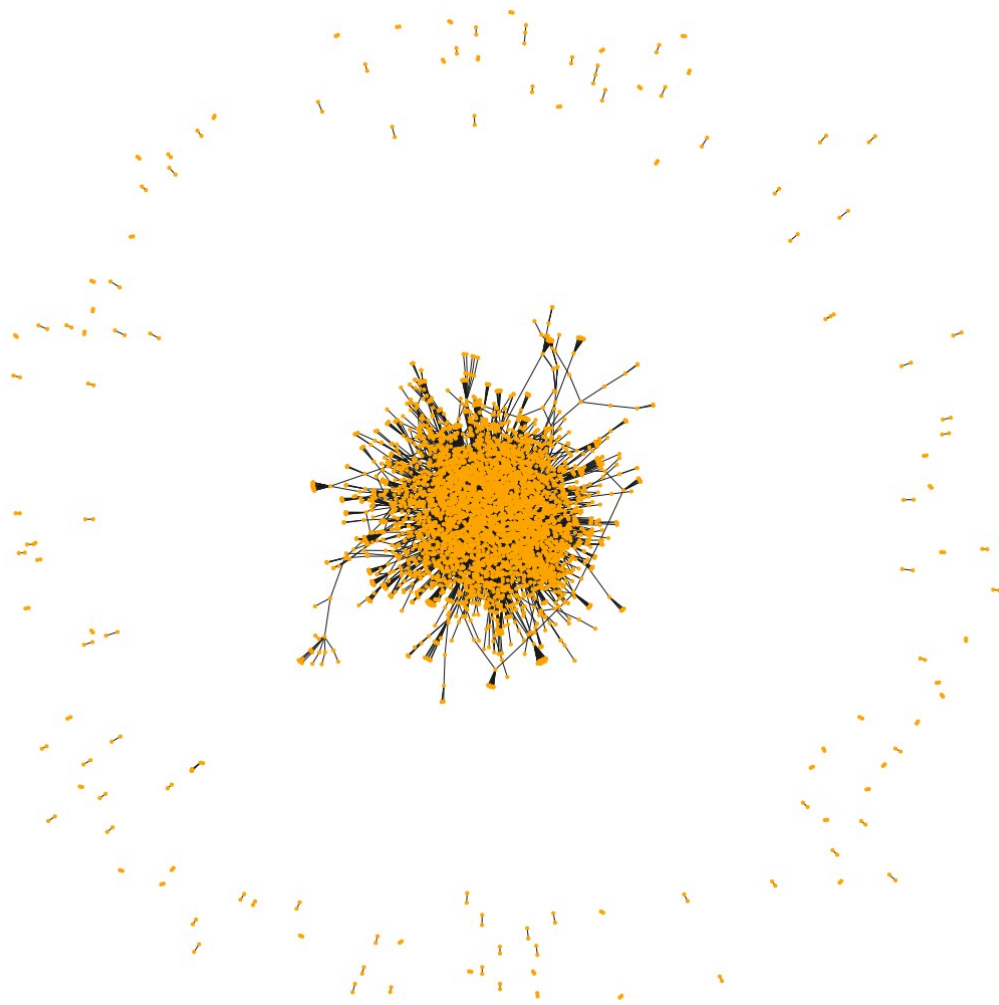
for node in G_DBOP_ParentIDLink:
    if node == 'root':
        G_DBOP_ParentIDLink_color_map.append('green')
    else:
        G_DBOP_ParentIDLink_color_map.append('orange')

plt.figure(3,figsize=(15,15))

nx.draw(G_DBOP_ParentIDLink, with_labels=False, node_size=10,
        node_color=G_DBOP_ParentIDLink_color_map)

plt.show()
```

Figure 62. T code for creating a graph with edges connected from the combined ID and Rev ID (IDandRev) node to the combined ParentID and ParentRevID node (ParentIDandRev) (Author's code)



*Figure 63. A NetworkX graph visualization of the DBOP hierarchical structure formed between ID and RevID pairs and ParentID and ParentRevID pairs (Author's code)*

The gap in Figure 63 can be filled by knowing that some data points are connected from the *Ma9\_ParentItem* column to *ParentID* and *ParentRevID* pairs. When preparing for the second graph, a copy of the data frame created in Figure 61 is made to reuse the combined *ParentID* and *ParentRevID* column. The new data frame is then modified to contain dedicated rows for each *Ma9\_ParentItem* id and revision pairs initially stored in the same value. The knowledge that the | -sign separates each pair is utilized in the split. The code for creating this data frame is presented in Figure 64.

```

#creating a copy from dfDBOP
dfDBOPForGraphs_Ma9 = dfDBOPForGraphs

#copying the Ma9_ParentItem column to keep the original
dfDBOPForGraphs_Ma9['Ma9_ParentItemIDandRev'] = dfDBOPForGraphs_Ma9['Ma9_ParentItem']

#splitting on |
#new rows are created for each split and the original data on the row is copied to the new row.
#Only difference is Ma9_ParentItem data that is splitted

dfDBOPForGraphs_Ma9['Ma9_ParentItem'] = dfDBOPForGraphs_Ma9['Ma9_ParentItem'].str.split('|')
dfDBOPForGraphs_Ma9 = dfDBOPForGraphs_Ma9.explode('Ma9_ParentItem').reset_index(drop=True)

cols = list(dfDBOPForGraphs_Ma9.columns)
cols.append(cols.pop(cols.index('Ma9_ParentItemIDandRev')))

dfDBOPForGraphs_Ma9 = dfDBOPForGraphs_Ma9[cols]

#splitting the ID from revision based on the separator string: /
dfDBOPForGraphs_Ma9[['Ma9_ParentItemID', 'Ma9_ParentItemRevID']] = dfDBOPForGraphs_Ma9.Ma9_ParentItem.str.split('/',
                                                                                                       expand=True)

```

Figure 64. The code where the data frame created in Figure 61 is copied to a new data frame where the *Ma9\_ParentItem* id and revision pairs are split into individual rows. (Author's code)

Figure 65 presents the code for creating the second NetworkX graph of the DBOP data, utilizing the connection between *Ma9\_ParentItem* and *ParentID* and *ParentRevID* pairs. Figure 66 presents the result. There is a dense data cluster forming in the middle and some data points surrounding it.

```

G_DBOP_Ma9ParentItemParentIDLink = nx.from_pandas_edgelist(
    df = dfDBOPForGraphs_Ma9,
    source='Ma9_ParentItemIDandRev',
    target='ParentIDandRev')

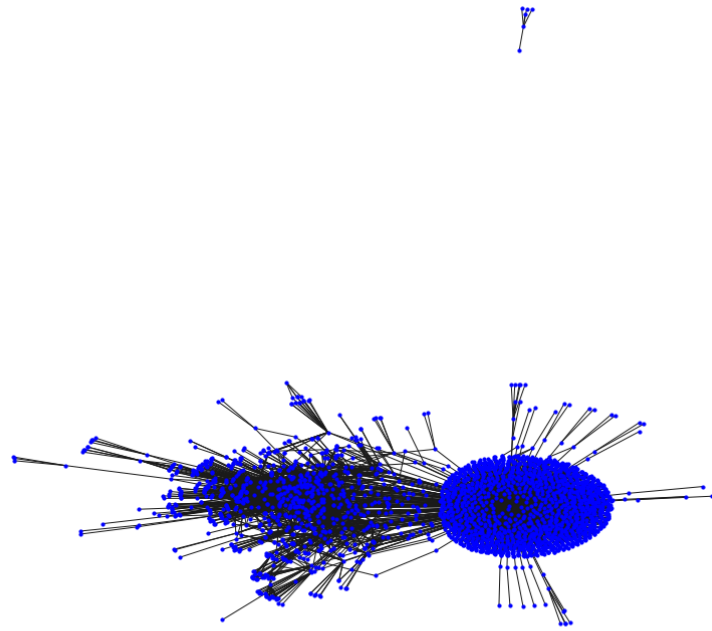
plt.figure(3,figsize=(15,15))

nx.draw(G_DBOP_Ma9ParentItemParentIDLink, with_labels=False, node_size=10, node_color='blue')

plt.show()

```

Figure 65. The code for creating a graph with edges connected from the *Ma9\_ParentItem* id and revision pair (*Ma9\_ParentItemIDandRev*) node to the combined *ParentID* and *ParentRevID* node (*ParentIDandRev*) (Author's code)



*Figure 66. A NetworkX graph visualization of the DBOP hierarchical structure formed between Ma9\_ParentItem id and revision pairs and ParentID and ParentRevID pairs (Author's code)*

The two separate graphs are combined into one with the **NetworkX compose** functionality. The code is presented in Figure 67. The node colors are kept as in Figure 63 and Figure 66. The result of the combined graph is presented in Figure 68. The result shows a dense data cluster with no disconnected data points. A meticulous reader may even spot the first node in the hierarchical structure, the *root* colored in green.

```
GH_DBOP_hierarchy = nx.compose(G_DBOP_ParentIDLink,G_DBOP_Ma9ParentItemParentIDLink)

#creating list of the Ma9_ParentItem nodes to be able to colour these differently
lstNodesInG_DBOP_Ma9ParentItemParentIDLink = list(dfDBOPForGraphs_Ma9.Ma9_ParentItemIDandRev)

#defining different colours for the nodes depending of "source graph"
GH_DBOP_color_map = []

for node in GH_DBOP_hierarchy:
    if node in lstNodesInG_DBOP_Ma9ParentItemParentIDLink:
        GH_DBOP_color_map.append('blue')
    else:
        if node == 'root':
            GH_DBOP_color_map.append('green')
        else:
            GH_DBOP_color_map.append('orange')

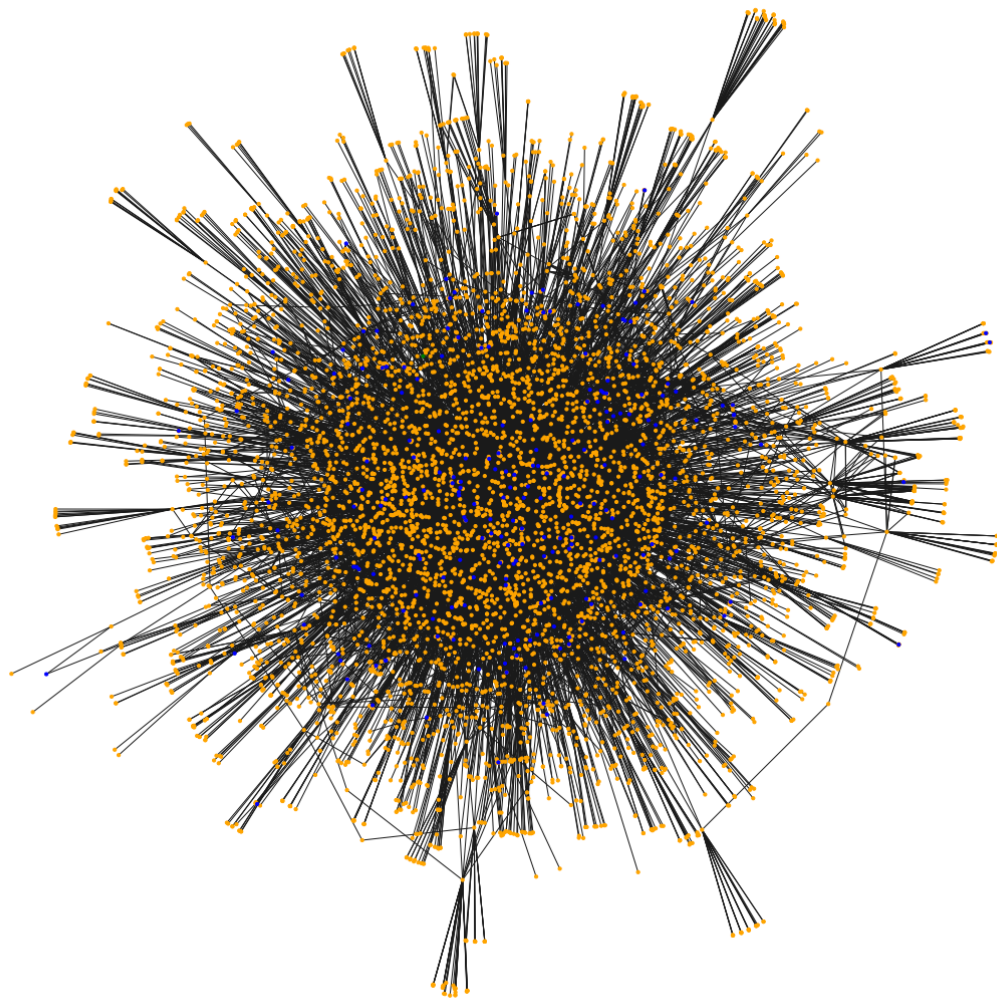
plt.figure(3,figsize=(15,15))

nx.draw(GH_DBOP_hierarchy, with_labels=False, node_color=GH_DBOP_color_map, node_size=10)

plt.show()
```

Figure 67. Creating a graph visualizing the complete DBOP hierarchical structure (Author's code)





*Figure 68. A NetworkX graph visualization of the DBOP hierarchical structure formed between ID and RevID pairs and ParentID and ParentRevID pairs (Author's code)*

Identifying the hierarchical structure in the graph visualizations with over 9000 data points is problematic. A couple of alternative visualizations with a reduced number of data points are shown in Figure 70 and Figure 72. Figure 69 presents the code where the selection of data points is made. The data points are limited based on their *StrLevel* value, which can be considered as a level in the hierarchical structure. In the starting node, the root has *StrLevel* 0 value. The nodes on the next level have values *StrLevel* 1, 1100, and 1200. The nodes after *StrLevel* 1 have *StrLevel* 2 value. The nodes after *StrLevel* 2 have *StrLevel* 3 value etc. The selection of data points is limited to include *StrLevel* values from 0 to 5. As all the data points in this selection are connected from the *ID* and *RevID* pair to the *ParentID* and *ParentRevID* pair, a copy of the data frame created in Figure

61 is used as a base when creating the data frame with limited data points. Figure 69 presents the limited data frame, which covers 435 data points.

After selecting the data points, Figure 69 presents how the NetworkX graph is created. The desired node color and size are set for the graph, and the **NetworkX graphviz\_layout** of type **dot** is used to visualize the graph, as seen in Figure 70.

```

strLevellist = [0,1,2,3,4,5,1100,1200]
dfDBOPForGraphs_limited = dfDBOPForGraphs[dfDBOPForGraphs["StrLevel"].isin(strLevellist)]
dfDBOPForGraphs_limited.shape[0]

435

G_dfDBOP_limited = nx.from_pandas_edgelist(
    df = dfDBOPForGraphs_limited,
    source='IDandRev',
    target='ParentIDandRev',
    edge_attr = ['StrLevel', 'Seq_Nr', 'Qty'])

#defining different colours for the nodes
G_DBOP_limited_color_map = []

for node in G_dfDBOP_limited:
    if node == 'root':
        G_DBOP_limited_color_map.append('green')
    else:
        G_DBOP_limited_color_map.append('orange')

intNodesize = 200
bollabels = False

plt.figure(3,figsize=(15,15))

pos = graphviz_layout(G_dfDBOP_limited, prog="dot")
nx.draw(G_dfDBOP_limited, pos, with_labels=bollabels, node_size=intNodesize,
        node_color=G_DBOP_limited_color_map)

plt.show()

```

Figure 69. The code for limiting the DBOP data frame to contain only five first levels in the DBOP hierarchical structure and for visualizing this as a NetworkX graph with graphviz\_layout of type dot. (Author's code)

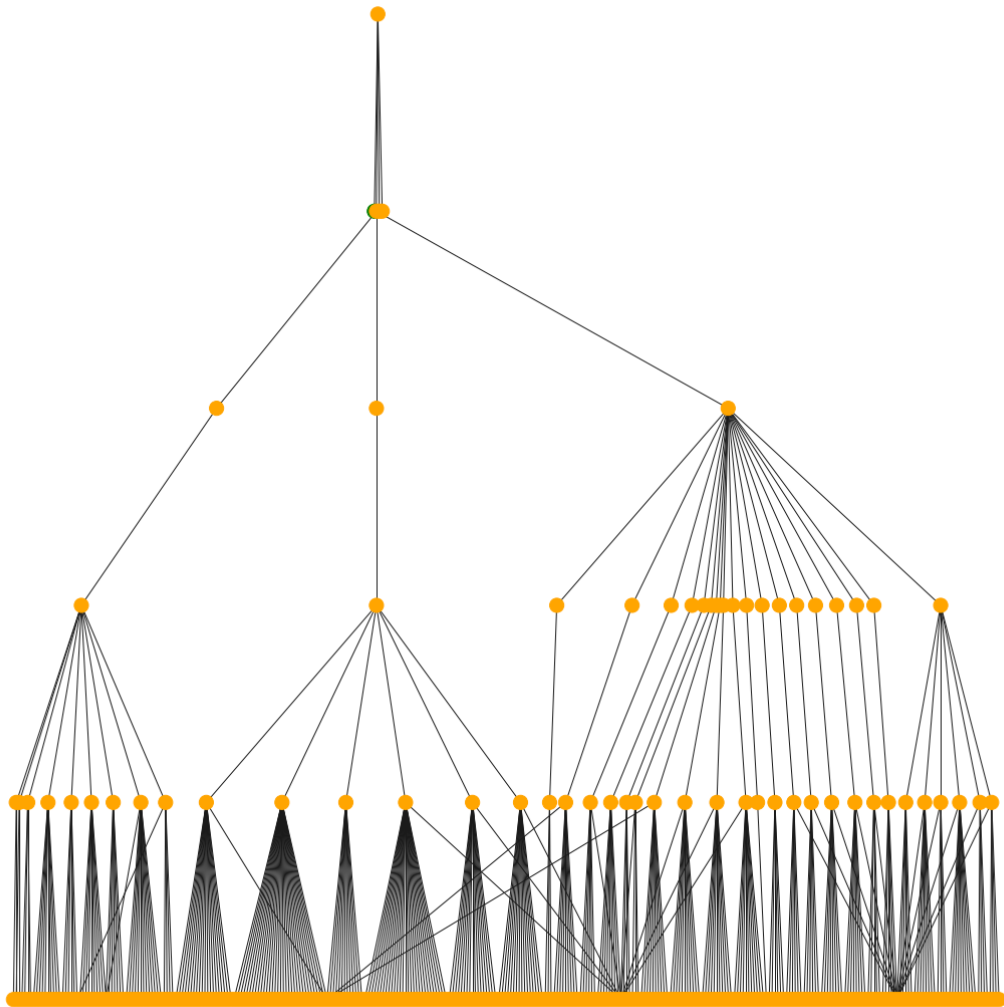


Figure 70. The five first levels in the DBOP hierarchical structure are visualized as a NetworkX graph with graphviz\_layout of type dot. (Author's code)

Figure 71 presents the code for changing the graphviz\_layout to another type. In this example, the layout algorithm **dot** is changed to **sfdp**. The result is presented in Figure 72.

```
plt.figure(3,figsize=(15,15))
pos = graphviz_layout(G_dFOBOP_limited, prog="sfdp")
nx.draw(G_dFOBOP_limited, pos, with_labels=bolLabels, node_size=intNodesize,
        node_color=G_DFOBOP_limited_color_map)
plt.show()
```

Figure 71. The code for an alternative NetworkX graphviz\_layout. This time of type sfdp. The five first levels in the DBOP hierarchical structure are visualized. (Author's code)

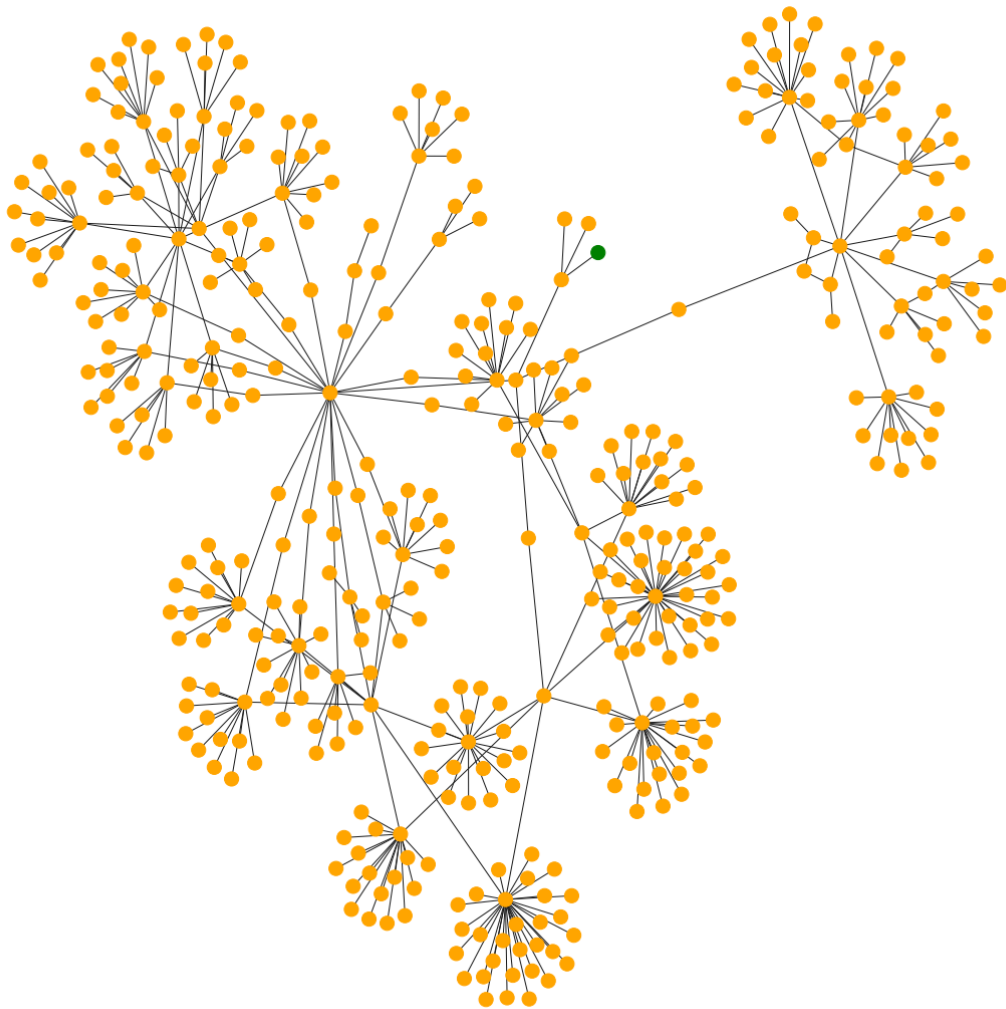


Figure 72. The five first levels in the DBOP hierarchical structure are visualized as a NetworkX graph with `graphviz_layout` of type `sfdp`. (Author's code)

The idea of using the number of non-null values as an indicator for possible data groups is derived from Figure 31. This chapter continues with the investigation of likely data groups.

The columns *EngineNumber*, *EngineAbbreviation*, and *EngineDescription*, have three *non-null* objects each. From the author's own Wärtsilä experience, it is known that the *EngineNumber* is a unique identifier. This means that there cannot be more than one engine with a specific *EngineNumber*. The business stakeholders further clarify that a DBOP always belongs to one specific *EngineNumber*. And an *EngineNumber* can only have one *EngineAbbreviation*

value and one *EngineDescription* value in the DBOP. These statements are confirmed with the code and result in Figure 73.

```
print('Number of unique values of EngineAbbreviation & EngineDescription per EngineNumber:')
dfDBOP.groupby('EngineNumber')['EngineAbbreviation', 'EngineDescription'].nunique()
```

Number of unique values of EngineAbbreviation & EngineDescription per EngineNumber:

EngineNumber	EngineAbbreviation	EngineDescription
[REDACTED]	1	1

Figure 73. A specific *EngineNumber* has one specific *EngineAbbreviation* value and one specific *EngineDescription* value. The *EngineNumber* is hidden in this picture. (Author's code)

The *EngineNumber*, *EngineAbbreviation*, and *EngineDescription* columns have values in text format. The actual values are irrelevant for this study. Creating a separate table for these columns with the *EngineNumber* as the primary key is relevant for the relational data model design. It is considered beneficial to highlight that the DBOP belongs to a specific engine with *EngineNumber*, *EngineAbbreviation*, and *EngineDescription* properties in the graph data model.

In the DBOP data frame, columns *RealizationID* and *RealizationRevisionID* each have 21 non-null row values. In Figure 74, the rows with a *RealizationID* value are selected from the DBOP data frame. This selection is stored in the *dfDBOPWithRealizationID* data frame. All columns containing only null values are dropped from this data frame. Fifteen columns remain.

```

#Selecting rows in DBOP with a RealizationID value and removing columns for
#this selection where column values are only null
dfDBOPWithRealizationID = dfDBOP[~dfDBOP['RealizationID'].isnull()]
dfDBOPWithRealizationID

dfDBOPWithRealizationID = dfDBOPWithRealizationID[dfDBOPWithRealizationID.columns[
    ~dfDBOPWithRealizationID.isnull().all()]]

print('Number of rows in the DBOP with RealizationID value: ',
      dfDBOPWithRealizationID.shape[0], '\n')

dfDBOPWithRealizationID.info()

Number of rows in the DBOP with RealizationID value:  21

<class 'pandas.core.frame.DataFrame'>
Int64Index: 21 entries, 5 to 25
Data columns (total 15 columns):
ParentID          21 non-null object
ParentRevID       21 non-null object
StrLevel          21 non-null int64
Seq_Nr           21 non-null int64
Qty              21 non-null object
ID               21 non-null object
RevID            21 non-null object
Name             21 non-null object
TcType           21 non-null object
ReleaseDate      21 non-null object
Description       21 non-null object
RealizationID    21 non-null object
RealizationRevisionID 21 non-null object
OwningUser       21 non-null object
OwningGroup      21 non-null object
dtypes: int64(2), object(13)
memory usage: 2.6+ KB

```

Figure 74. The code for creating a data frame with rows containing a *RealizationID* and a *RealizationRevisionID*. (Author's code)

It is interesting to understand how the *RealizationID* and *RealizationRevisionID* relate to the DBOP hierarchical structure. According to Figure 60, central elements in the hierarchical structure are *ID* and *RevID* pairs, *ParentID* and *ParentRevID* pairs, and *Ma9\_ParentItem*. In the *dfDBOPWithRealizationID* data frame, there is no *Ma9\_ParentItem* value, and we can conclude that there is no direct relationship between the *RealizationID* and *RealizationRevisionID* pair and *Ma9\_ParentItem*. *ID*, *RevID*, *ParentID*, and *ParentRevID* columns exist and are further investigated.

Figure 75 presents a function for plotting the number of unique values of columns in a *columnList* argument per column combination in a *group* argument.

```

def uniqueValueGraphForColumnList(dataframeName, group, columnList, graphTitle):
    dataframeName.groupby(group)[columnList].nunique().plot.bar(rot=0)

    plt.title(graphTitle, size=15)
    plt.ylabel('Count')

    #changing the y-axis range from decimals to integers
    locs, labels = plt.yticks()
    yint = []

    for each in locs:
        yint.append(int(each))
    plt.yticks(yint)

    plt.xticks([]) #hiding x-axis ticks

```

Figure 75. A function for plotting how many values for the columns given in the `columnList` argument exist for the columns in the `group` argument. The calculation is performed on the data in the data frame specified in the `dataframeName` argument. The `graphTitle` argument specifies the title of the plot. (Author's code)

Figure 76 presents the result of calling the function in Figure 75 for plotting the number of unique values of *RealizationID* and *RealizationRevisionID* per *ID* and *RevID* pair. As each *ID* and *RevID* pair has one unique *RealizationID* and *RealizationRevisionID*, we can determine that the *RealizationID* and *RealizationRevisionID* depend on the *ID* and *RevID* pair.

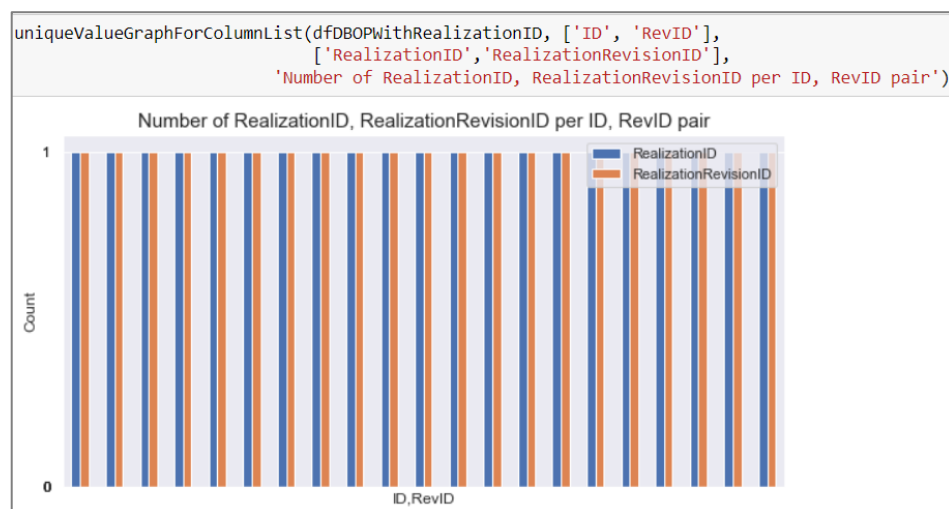


Figure 76. The number of unique values of *RealizationID* and *RealizationRevisionID* per *ID* and *RevID* pair (Author's code)

Figure 77 presents the result of calling the function in Figure 75 to plot the number of unique values of *RealizationID* and *RealizationRevisionID* per *ParentID* and *ParentRevID* pair. The *RealizationID* and *RealizationRevisionID* are not dependent on *ParentID* and *ParentRevID* pairs as a unique *ParentID* and *ParentRevID* pair can have up to 19 different *RealizationID* values and 2 *RealizationRevisionID* values.



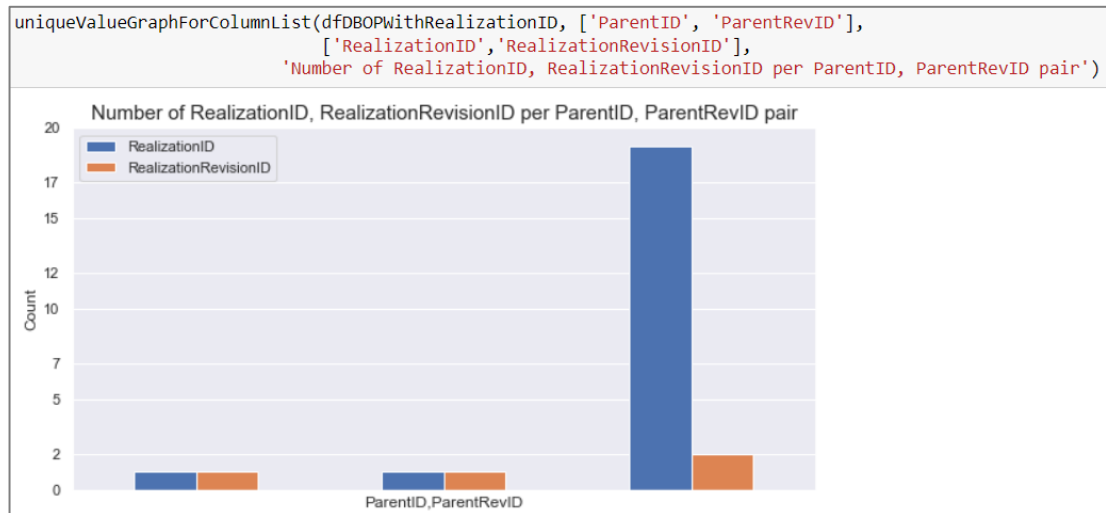


Figure 77. The number of unique values of *RealizationID* and *RealizationRevisionID* per *ParentID* and *ParentRevID* pair (Author's code)

A direct relationship is confirmed between the *RealizationID* and *RealizationRevisionID* and *ID* and *RevID* pairs. The *RealizationID* values are similar to those in *ID*, *ParentID*, and *Ma9\_ParentItem*. Hence, it is decided to investigate if the *RealizationID* and *RealizationRevisionID* pairs are involved in forming the DBOP hierarchical structure. Figure 78 presents the code for checking if values of the *RealizationID* are found in the *ID*, *ParentID*, or *Ma9\_ParentItem* columns. The result shows that the *RealizationID* is not found in these columns. Hence, we can conclude that the *RealizationID* and *RealizationRevisionID* pair do not contribute to the DBOP hierarchical structure.



```

dfDBOP_IDRealizationIDMatch =dfDBOP[dfDBOP['ID'].isin(lstRealizationID)]

print('Number of rows in DBOP where a RealizationID found in the ID column: ',
      dfDBOP_IDRealizationIDMatch.shape[0])

Number of rows in DBOP where a RealizationID found in the ID column: 0

dfDBOP_IDRealizationIDMatch =dfDBOP[dfDBOP['ParentID'].isin(lstRealizationID)]

print('Number of rows in DBOP where a RealizationID found in the ParentID column: ',
      dfDBOP_IDRealizationIDMatch.shape[0])

Number of rows in DBOP where a RealizationID found in the ParentID column: 0

dfMa9_ParentItemRealizationIDMatch =table_Ma9_ParentItem[
    table_Ma9_ParentItem['Ma9_ParentItemID'].isin(lstRealizationID)]

print('Number of rows in DBOP where a RealizationID found in the Ma9_ParentItem column: ',
      dfMa9_ParentItemRealizationIDMatch.shape[0])

Number of rows in DBOP where a RealizationID found in the Ma9_ParentItem column: 0

```

Figure 78. The code for verifying that the *RealizationID* and *RealizationRevisionID* do not contribute to the hierarchical structure of the DBOP (Author's code)

From Figure 31, reveals that the *DrawingRelation* has 257 occurrences with non-null values. The possible values for *DrawingRelation* are presented in Figure 79. The remaining columns in a data frame when extracting the rows with a *DrawingRelation* and then removing columns with only null values are presented in Figure 80. In this data frame, there are 257 rows and 17 columns. Among these columns are the *ParentID* and *ParentRevID* pair and the *ID* and *RevID* pair. As these are central elements in the DBOP hierarchical structure, it needs to be investigated if there is a relation between these pairs and the *DrawingRelation*.

```

print('The alternative DrawingRelation found in DBOP are:')

lstDrawingRelation = list(dfDBOP.DrawingRelation.unique())

[print(x) for x in lstDrawingRelation]

The alternative DrawingRelation found in DBOP are:
nan
Tool Drawing
Described By
Related Drawings
Ma9_Relate_DBOP
Wb8_RelatedDesign

```

Figure 79. *DrawingRelation* values (Author's code)

```

#Selecting rows with DrawingRelation value and removing columns with only null values
dfDBOPWithDrawingRelation = dfDBOP[~dfDBOP['DrawingRelation'].isnull()]
dfDBOPWithDrawingRelation

dfDBOPWithDrawingRelation = dfDBOPWithDrawingRelation[dfDBOPWithDrawingRelation.columns[
    ~dfDBOPWithDrawingRelation.isnull().all()]]

print('Number of rows in the DBOP with DrawingRelation value: ',
      dfDBOPWithDrawingRelation.shape[0], '\n')

dfDBOPWithDrawingRelation.info()

Number of rows in the DBOP with DrawingRelation value: 257

<class 'pandas.core.frame.DataFrame'>
Int64Index: 257 entries, 8953 to 9209
Data columns (total 17 columns):
ParentID          257 non-null object
ParentRevID       257 non-null object
StrLevel          257 non-null int64
Seq_Nr            257 non-null int64
Qty               257 non-null object
ID                257 non-null object
RevID             257 non-null object
Name              257 non-null object
TcType            257 non-null object
ReleaseDate       255 non-null object
Description        230 non-null object
EngineNumber      2 non-null object
EngineAbbreviation 2 non-null object
EngineDescription 2 non-null object
OwningUser        255 non-null object
OwningGroup       255 non-null object
DrawingRelation   257 non-null object
dtypes: int64(2), object(15)
memory usage: 36.1+ KB

```

Figure 80. The code and result when selecting the rows in the DBOP where a *DrawingRelation* value exists. After the row selection, the columns with only null values are removed. (Author's code)

The function in Figure 36 plots the number of unique *DrawingRelation* values per *ID* and *RevID* pair. Figure 81 presents the result. The result shows that there are three *ID* and *RevID* pairs with two different values of *DrawingRelation*. The remaining 254 *ID* and *RevID* pairs have a unique *DrawingRelation* value. Figure 82 presents the result of calling the function in Figure 36 to understand how many *DrawingRelation* values exist for each *ParentID* and *ParentRevID* pair. The result returns 12 *ParentID* and *ParentRevID* pairs with two *DrawingRelation* values (Figure 82). Calling the function in Figure 36 with the group argument extended to *ParentID* and *ParentRevID*, *StrLevel*, *ID*, and *RevID* columns, shows that two occurrences with two different *DrawingRelation* values are still returned (Figure 83).

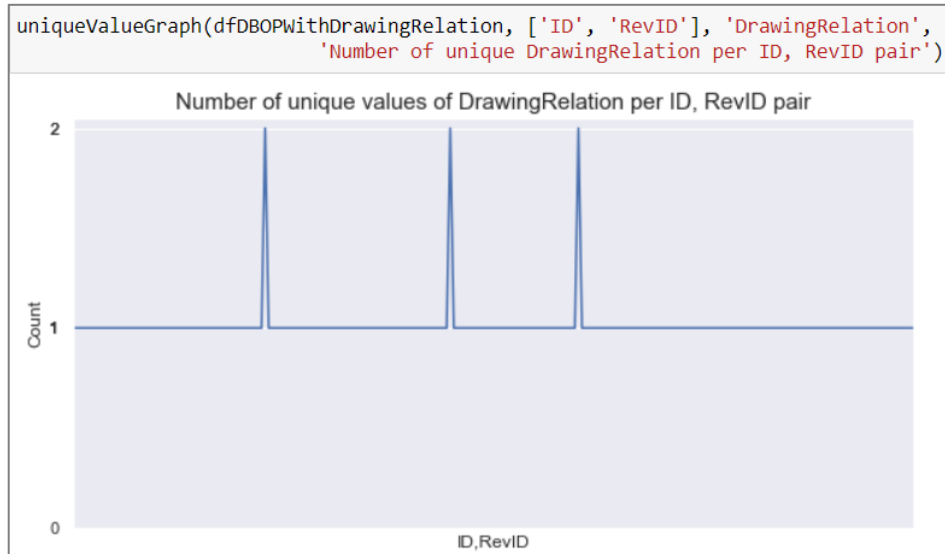


Figure 81. The number of unique DrawingRelation values per ID and RevID pair (Author's code)

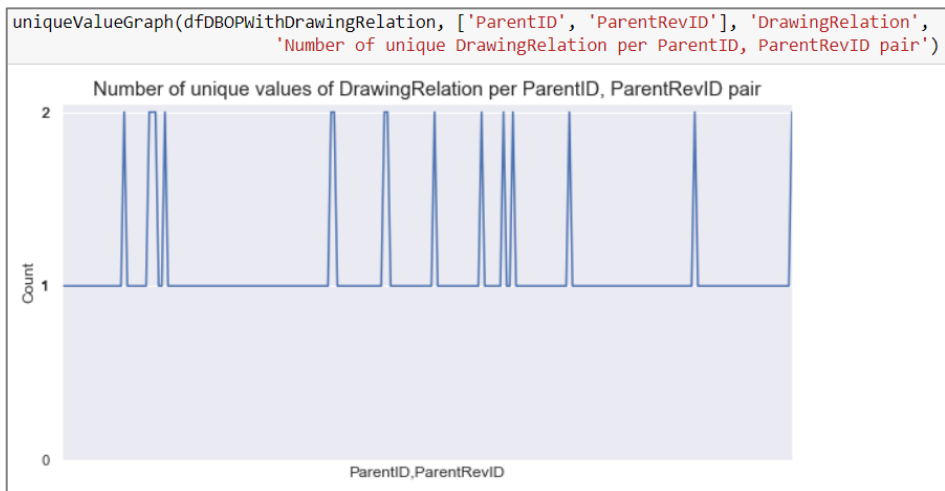


Figure 82. The number of unique DrawingRelation values per ParentID and ParentRevID pair (Author's code)

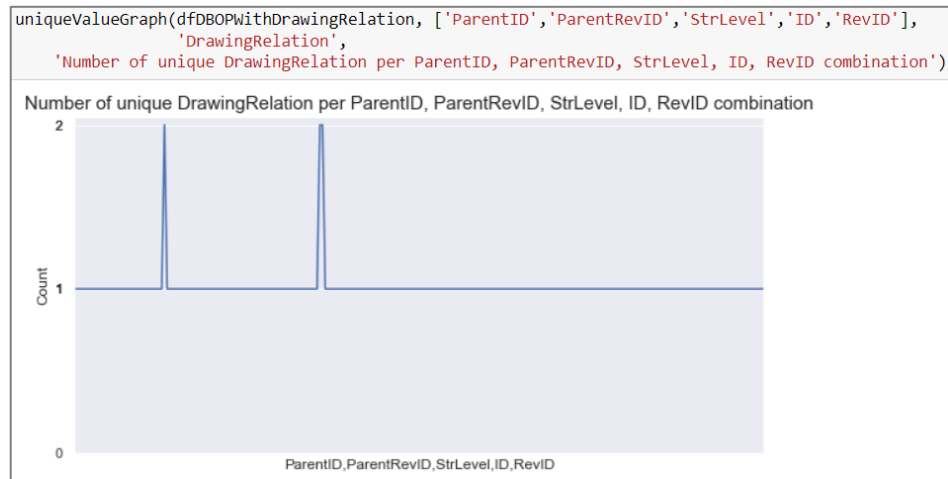


Figure 83. The number of unique *DrawingRelation* values per *ParentID* and *ParentRevID*, *StrLevel*, *ID*, and *RevID* combination (Author's code)

Based on the author's Wärtsilä experience, the expectation is that each *ID* and *RevID* pair returns a unique value of *DrawingRelation*. The reason why this does not happen is investigated by outputting the data rows with *ID* and *RevID* pair duplicates. The code for this is visible in Figure 84. The result is not made visible in this report. With a visual inspection of the result, it is noticed that two data rows require all the available columns to be used to identify a unique *DrawingRelation* value. The author assumes this is a data quality error and that each *ID* and *RevID* pair will have a unique value of *DrawingRelation*.

```
dfDBOPWithDrawingRelation[dfDBOPWithDrawingRelation[['ID', 'RevID']].duplicated() == True]
```

Figure 84. The code for investigating the rows with an *ID* and *RevID* pair occurring on several rows. Aiming to understand what the *DrawingRelation* values on these rows are. (Author's code)

It is investigated next if the columns with a non-null value above 7000 could form a data group with the *ID* and *RevID* column pair as a primary key. The investigation is performed by checking if each *ID* and *RevID* pair returns a unique column value for each of the following columns:

- *StrLevel* (9210 non-null values)
- *Seq\_Nr* (9210 non-null values)
- *Qty* (9210 non-null values)
- *ID* (9210 non-null values)
- *RevID* (9210 non-null values)
- *Name* (9210 non-null values)
- *TcType* (9210 non-null values)

- *ReleaseDate* (8187 non-null values)
- *Description* (7030 non-null values)
- *OwningUser* (9208 non-null values)
- *OwningGroup* (9208 non-null values)

The *uniqueValueGraph* function in Figure 36 is called to plot the number of unique values for each column per *ID* and *RevID* pair. Each *ID* and *RevID* pair is expected to return one unique column value to identify a dependence between a column.

Figure 85 reveals that an *ID* and *RevID* pair return one or two unique *StrLevel* values. This means that *StrLevel* is not dependent on *ID* and *RevID* pairs.



Figure 85. The number of unique values of *StrLevel* per *ID* and *RevID* pair (Author's code)

Figure 86 reveals that an *ID* and *RevID* pair return either one or many unique *Seq\_Nr* values. This means that *Seq\_Nr* is not dependent on *ID* and *RevID* pairs.

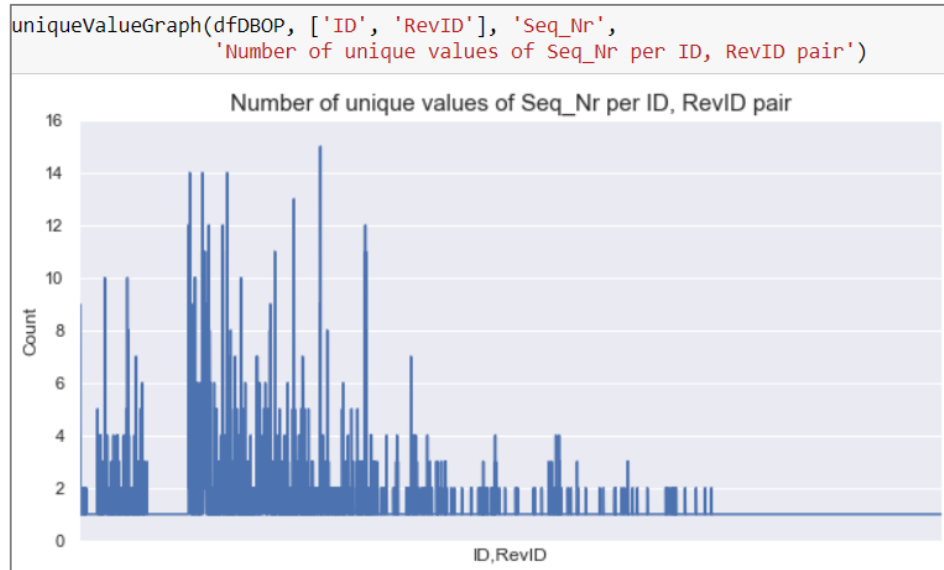


Figure 86. The number of unique values of *Seq\_Nr* per *ID* and *RevID* pair (Author's code)

Figure 87 reveals that an *ID* and *RevID* pair return either one or many unique *Qty* values. This means that *Qty* is not dependent on *ID* and *RevID* pairs.

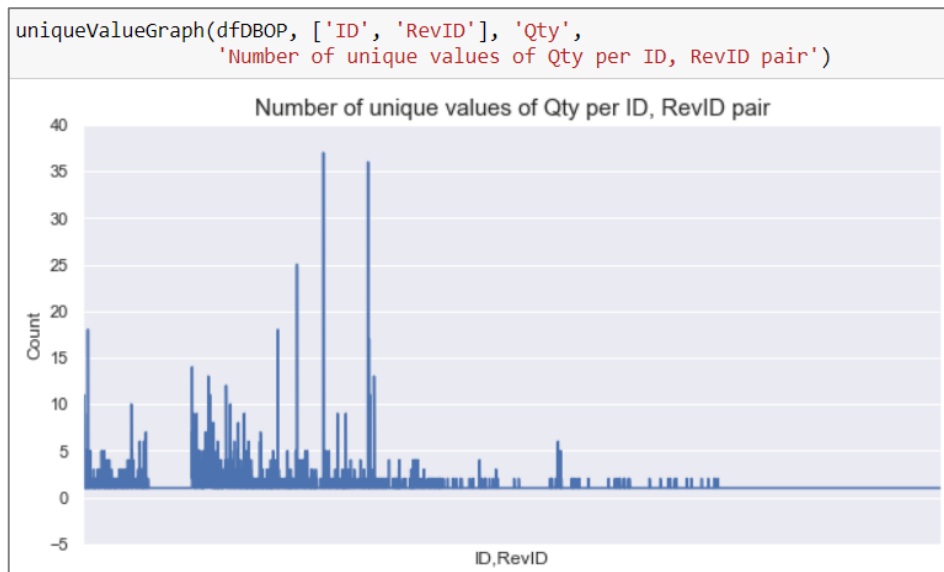


Figure 87. The number of unique values of *Qty* per *ID* and *RevID* pair (Author's code)

Figure 88 reveals that all *ID* and *RevID* pairs return a unique *Name* value. This means that *Name* is dependent on *ID* and *RevID* pairs.

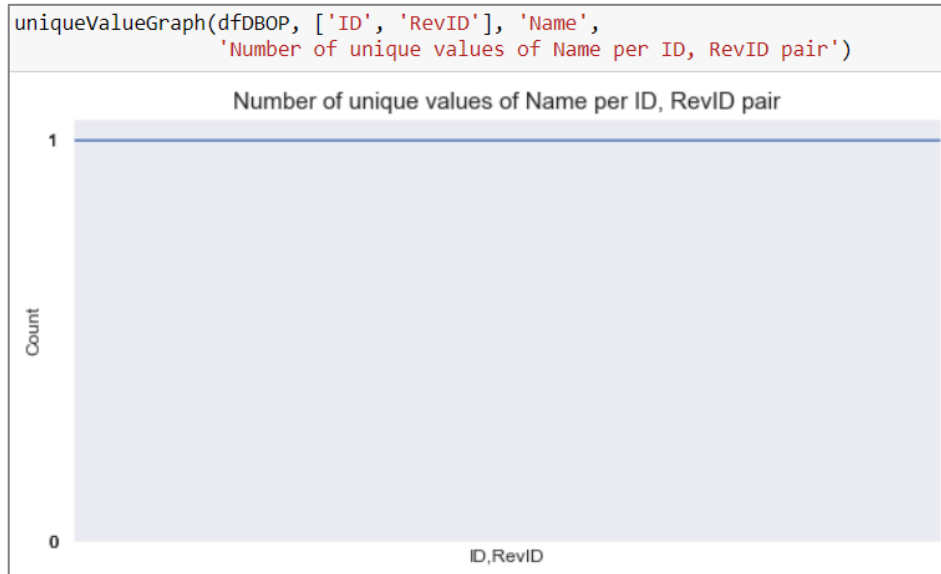


Figure 88. The number of unique values of Name per ID and RevID pair (Author's code)

Figure 89 reveals that all *ID* and *RevID* pairs return a unique *TcType* value. This means that *TcType* is dependent on *ID* and *RevID* pairs.

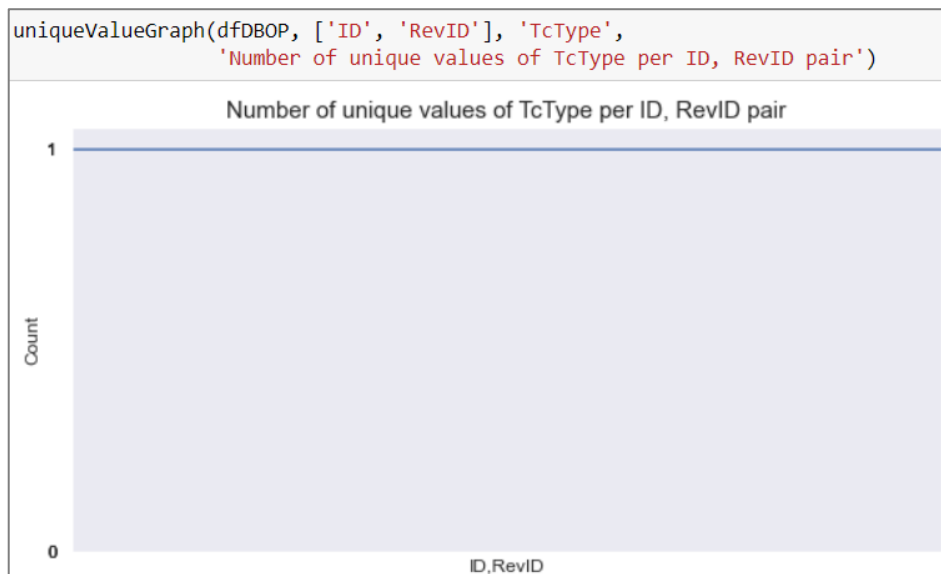


Figure 89. The number of unique values of TcType per ID and RevID pair (Author's code)

As 8187 of the 9210 rows in the DBOP data frame have a *ReleaseDate* value, it was decided to filter out these rows to a different data frame to get a cleaner plot. The code for creating a separate data frame is visible in Figure 90. Figure 91 reveals that all rows with a *ReleaseDate* have a unique *ReleaseDate* for each *ID* and *RevID* pair. This means that the *ReleaseDate* is dependent on *ID* and *RevID* pairs.

```
dfDBOP_ReleaseDate = dfDBOP[~dfDBOP['ReleaseDate'].isnull()]
dfDBOP_ReleaseDate

dfDBOP_ReleaseDate = dfDBOP_ReleaseDate[dfDBOP_ReleaseDate.columns[
    ~dfDBOP_ReleaseDate.isnull().all()]]

print('Number of rows in the DBOP with ReleaseDate value: ',
      dfDBOP_ReleaseDate.shape[0], '\n')

Number of rows in the DBOP with ReleaseDate value: 8187
```

Figure 90. The code for creating a data frame for the rows with a ReleaseDate value. After the row selection, the columns with only null values are dropped from the data frame. (Author's code)

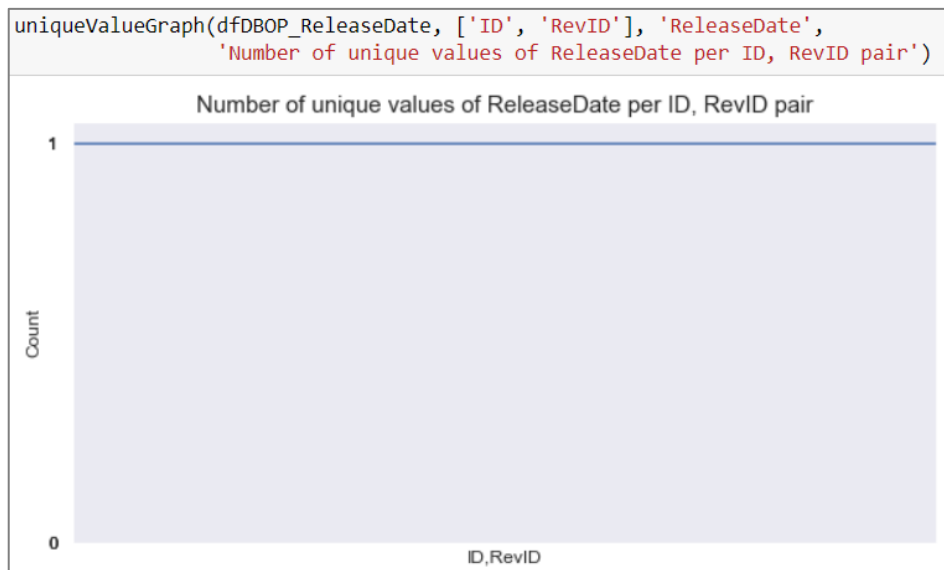


Figure 91. The number of unique values of ReleaseDate per ID and RevID pair (Author's code)

As 7030 of the 9210 rows in the DBOP data frame have a *Description* value, it was decided to filter out these rows to a different data frame to get a cleaner plot. The code for creating a separate data frame is visible in Figure 92. Figure 93 reveals that all rows with a *Description* value return a unique *Description* value for each *ID* and *RevID* pair. This means that the *Description* is dependent on *ID* and *RevID* pairs.



```

dfDBOP_Description = dfDBOP[~dfDBOP['Description'].isnull()]
dfDBOP_Description

dfDBOP_Description = dfDBOP_Description[dfDBOP_Description.columns[
    ~dfDBOP_Description.isnull().all()]]

print('Number of rows in the DBOP with Description value: ',
      dfDBOP_Description.shape[0], '\n')

Number of rows in the DBOP with Description value: 7030

```

Figure 92. The code for creating a data frame for the rows with a Description value. After the row selection, the columns with only null values are dropped from the data frame. (Author's code)

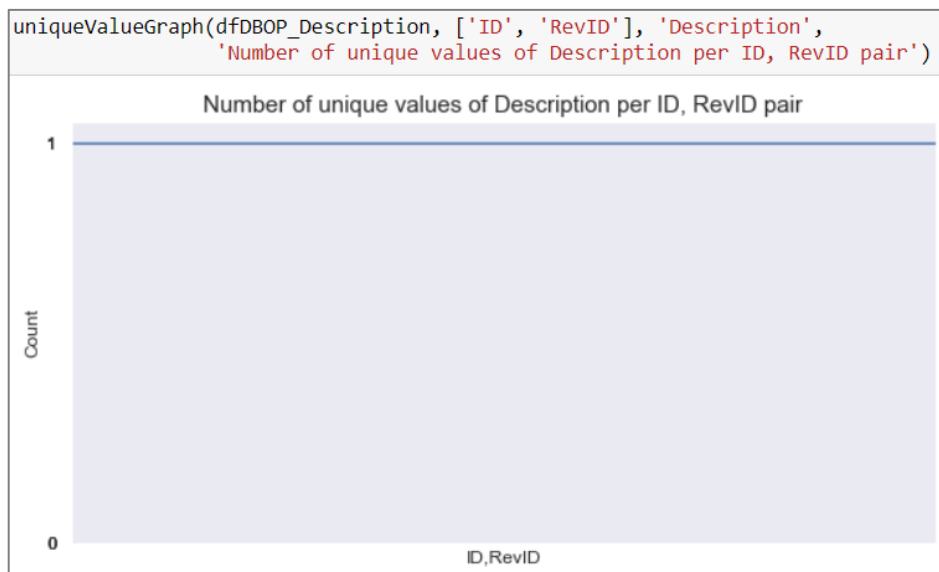


Figure 93. The number of unique values of Description per ID and RevID pair (Author's code)

The plots in Figure 94 and Figure 95 show that all *ID* and *RevID* pairs return a unique *OwningUser* and *OwningGroup* value. This means that both the *OwningUser* and the *OwningGroup* depend on *ID* and *RevID* pairs. The drop to count level 0 represents two data rows in the DBOP with missing *OwningUser* and *OwningGroup* values.

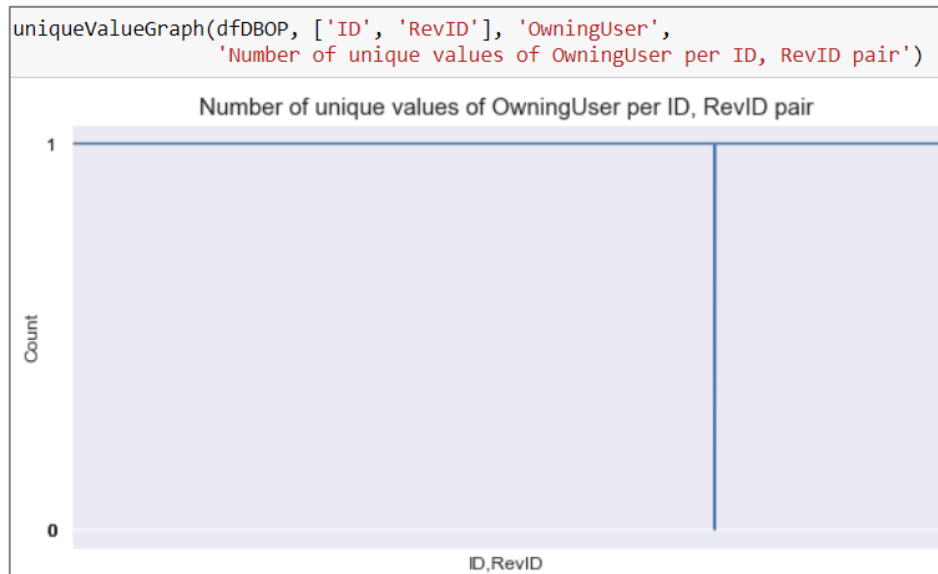


Figure 94. The number of unique values of *OwningUser* per *ID* and *RevID* pair (Author's code)

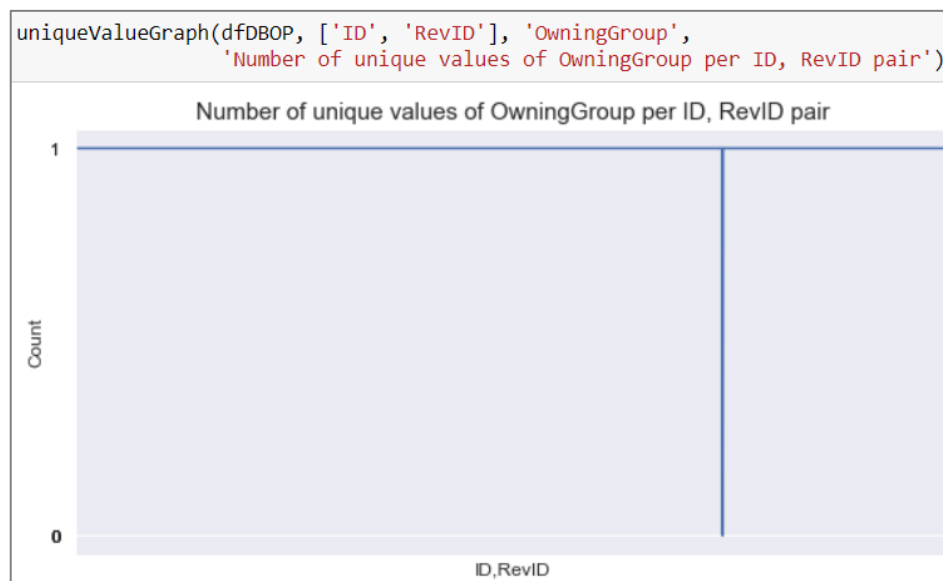


Figure 95. The number of unique values of *OwningGroup* per *ID* and *RevID* pair (Author's code)

The investigation of the columns with non-null values above 7000 reveals that *Name*, *TcType*, *ReleaseDate*, *Description*, *OwningUser*, and *OwningGroup* depend on the *ID* and *RevID* column pair. *StrLevel*, *Seq\_Nr*, and *Qty* are not dependent on the *ID* and *RevID* column pair. Based on the data frame content in Figure 52, the *StrLevel* and *Seq\_Nr* are assumed to be directly involved in forming the DBOP process steps together with *ParentID* and *ParentRevID*. *Qty* is interpreted to indicate how many repetitions of a specific process step are needed.

The following columns have around 300 non-null values in the DBOP data frame:

- *PhaseLevel* (370 non-null values)
- *PlantLevel* (331 non-null values)
- *Process Type* (331 non-null values)
- *AlternateProcess* (331 non-null values)
- *PSA* (331 non-null values)
- *SortString* (333 non-null values)

The function in Figure 96 is utilized for plotting the number of unique values for the columns included in the *columnList* argument per the column or columns included in the *group* argument. It is first assumed that all these columns belong to the same group. If this is true, and one of the columns is selected as the primary key, there is only one alternative column to choose as the primary key. This is the column with the highest number of non-null values, the *PhaseLevel* with 370 non-null values. None of the other columns can be the primary key, as a primary key cannot have null values. For example, the *PlantLevel* column requires  $370 - 331 = 39$  null values in a table to cover the 370 non-null values of the *PhaseLevel* column.

```
def uniqueValueGraphForColumnListLineFormat(dataframeName, group, columnList, graphTitle):
    dataframeName.groupby(group)[columnList].nunique().plot()

    plt.title(graphTitle, size=15)
    plt.ylabel('Count')

    #changing the y-axis range from decimals to inntegers
    locs, labels = plt.yticks()
    yint = []

    for each in locs:
        yint.append(int(each))
    plt.yticks(yint)

    plt.xticks([]) #hiding x ticks
```

Figure 96. The function for plotting the number of unique values of XXXXX. (Author's code)

Figure 97 reveals that the only column possibly dependent on *PhaseLevel* is the *AlternateProcess*. All other columns have several unique values per *PhaseLevel*. Figure 98 confirms that every distinctive value of *PhaseLevel* returns a unique *AlternateProcess* value.

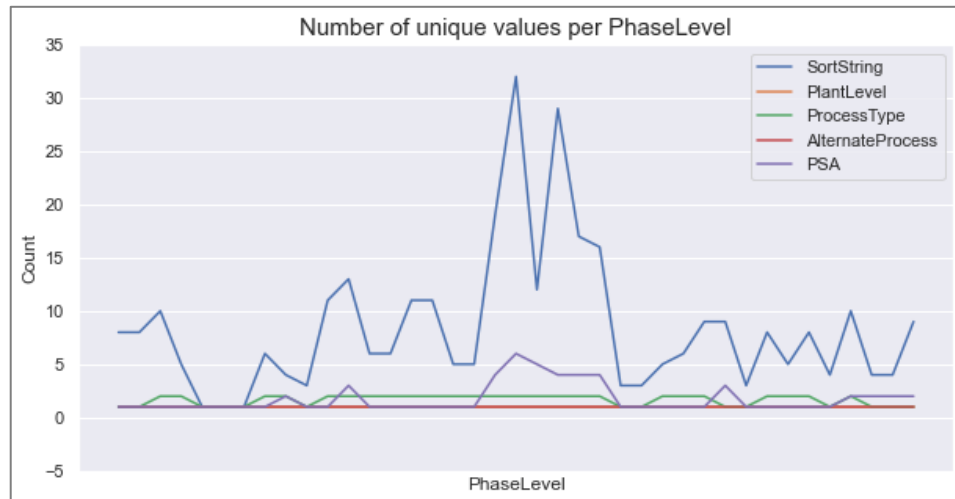


Figure 97. The number of unique *SortString*, *PlantLevel*, *ProcessType*, *AlternateProcess*, and *PSA*, values per *PhaseLevel*. (Author's code)

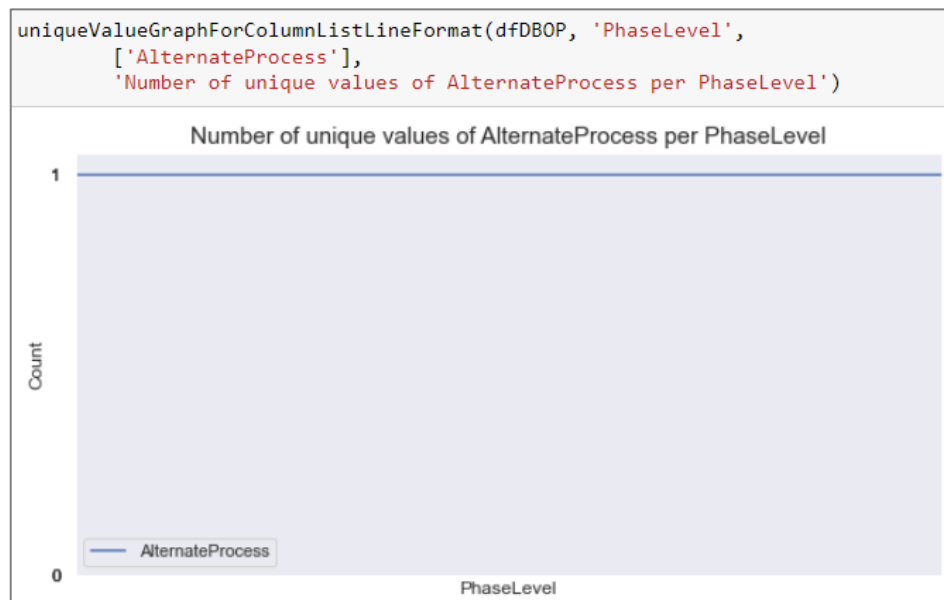


Figure 98. The number of unique *AlternateProcess* values per *PhaseLevel* value (Author's code)

*SortString* is the column with the following highest number of non-null values. When selecting *SortString* as the primary key, the *PhaseLevel* no longer fits in the group. The reason is that *PhaseLevel* has more rows than *SortString*. When *SortString* is the primary key, only 333 rows with non-null values for *SortString* are included in the group. The code and result for this selection are visible in Figure 99. When looking at the *PhaseLevel* non-null value, it is noted that this value has dropped from 370 to 331. This confirms that the *PhaseLevel* cannot be in a group with the *SortString* as the primary key, because it loses 39 *PhaseLevel* values.

```

dfDBOP_withSortStringValue = dfDBOP[~dfDBOP['SortString'].isnull()]
dfDBOP_withSortStringValue

dfDBOP_withSortStringValue = dfDBOP_withSortStringValue[dfDBOP_withSortStringValue.columns[
    ~dfDBOP_withSortStringValue.isnull().all()]]

print('Number of rows in the DBOP with SortString value: ',
      dfDBOP_withSortStringValue.shape[0], '\n')

dfDBOP_withSortStringValue.info()

Number of rows in the DBOP with SortString value: 333

<class 'pandas.core.frame.DataFrame'>
Int64Index: 333 entries, 78 to 5740
Data columns (total 20 columns):
ParentID          333 non-null object
ParentRevID       333 non-null object
StrLevel          333 non-null int64
Seq_Nr           333 non-null int64
Qty               333 non-null object
ID                333 non-null object
RevID             333 non-null object
Name              333 non-null object
TcType            333 non-null object
ReleaseDate       332 non-null object
Description        331 non-null object
PhaseLevel        331 non-null object
PlantLevel        331 non-null object
ProcessType       331 non-null object
QualityKey        73 non-null object
AlternateProcess   331 non-null object
OwningUser        333 non-null object
OwningGroup       333 non-null object
PSA               331 non-null object
SortString        333 non-null object
dtypes: int64(2), object(18)
memory usage: 54.6+ KB

```

Figure 99. The code for creating a data frame for the rows with a SortString value. (Author's code)

Figure 100 reveals that *PlantLevel*, *ProcessType*, *AlternateProcess*, and *PSA* are suited in a group where the *SortString* is selected as the primary key. Each non-primary key column will have a null value on two of the 333 rows. This is identified from the drop to count level 0 in Figure 100 and the indicated number of non-null values per column in Figure 99.

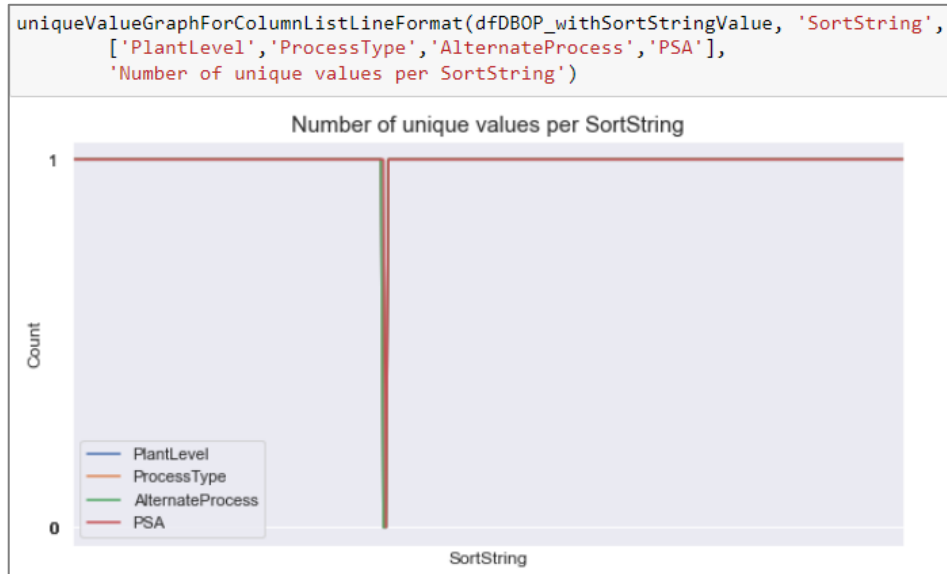


Figure 100. The number of unique PlantLevel, ProcessType, AlternateProcess, and PSA values per SortString. (Author's code)

Figure 101 verifies that selecting the *ID* and *RevID* pair as the key for this group is impossible. The main reason for this is that the number of unique *SortString* values per *ID* and *RevID* is more than one.

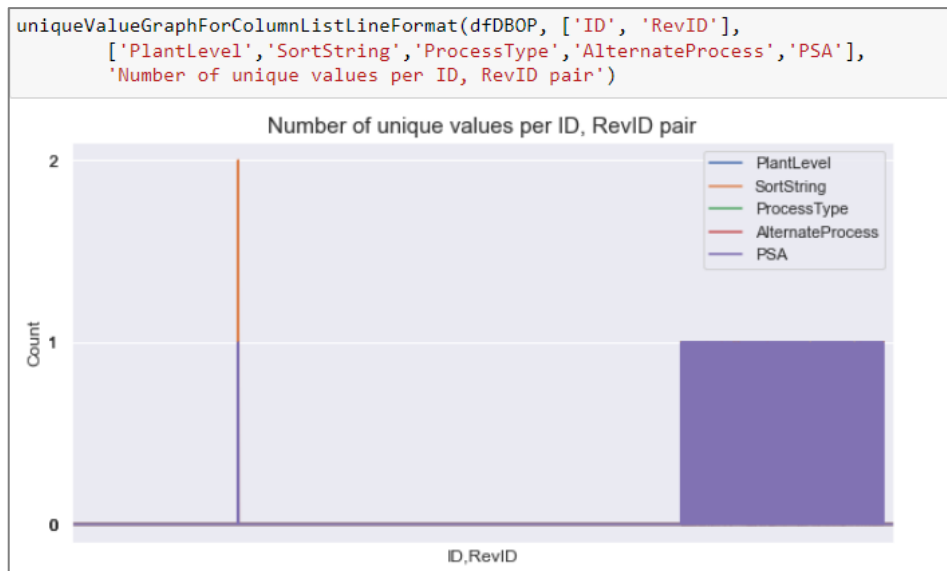


Figure 101. Verifying that the *ID* and *RevID* pair is not possible as the primary key in the group of PlantLevel, SortString, ProcessType, AlternateProcess, and PSA. (Author's code)

As *PhaseLevel* does not suit the *PlantLevel*, *SortString*, *ProcessType*, *AlternateProcess* and *PSA* group, it was next investigated if the *PhaseLevel* is dependent on the central elements in the DBOP hierarchical structure:

- *ID* and *RevID* pair
- *ParentID* and *ParentRevID* pair

Investigating *PhaseLevel* dependencies on *Ma9\_ParentItem*, the third column involved in forming the DBOP hierarchical structure is unnecessary. This decision was made based on Figure 102, where it can be noted that *Ma9\_ParentItem* is not included in the data frame containing the rows with a *PhaseLevel* value.

```
dfDBOP_withPhaseLevelValue = dfDBOP.loc[~dfDBOP["PhaseLevel"].isnull()]
dfDBOP_withPhaseLevelValue = dfDBOP_withPhaseLevelValue[dfDBOP_withPhaseLevelValue.columns[
~dfDBOP_withPhaseLevelValue.isnull().all()]]
dfDBOP_withPhaseLevelValue.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 370 entries, 26 to 432
Data columns (total 20 columns):
ParentID          370 non-null object
ParentRevID       370 non-null object
StrLevel          370 non-null int64
Seq_Nr            370 non-null int64
Qty               370 non-null object
ID                370 non-null object
RevID             370 non-null object
Name              370 non-null object
TcType           370 non-null object
ReleaseDate       369 non-null object
Description        370 non-null object
PhaseLevel        370 non-null object
PlantLevel        331 non-null object
ProcessType        331 non-null object
QualityKey         73 non-null object
AlternateProcess   331 non-null object
OwningUser         370 non-null object
OwningGroup        370 non-null object
PSA                330 non-null object
SortString         331 non-null object
dtypes: int64(2), object(18)
memory usage: 60.7+ KB
```

Figure 102. The code for creating a data frame for the rows with a *PhaseLevel* value. (Author's code)

The result presented in Figure 103, Figure 104, and Figure 105 reveals that *PhaseLevel* is dependent on the *ID* and *RevID* pair. As *ParentID* and *ParentRevID* pairs can return several unique *PhaseLevel* values, the *PhaseLevel* is not reliant on this pair. When extending the *ParentID* and *ParentRevID* pair with *StrLevel*, and *Seq\_Nr*, it is noted that this combination returns unique *PhaseLevel* values and is dependent on this combination.

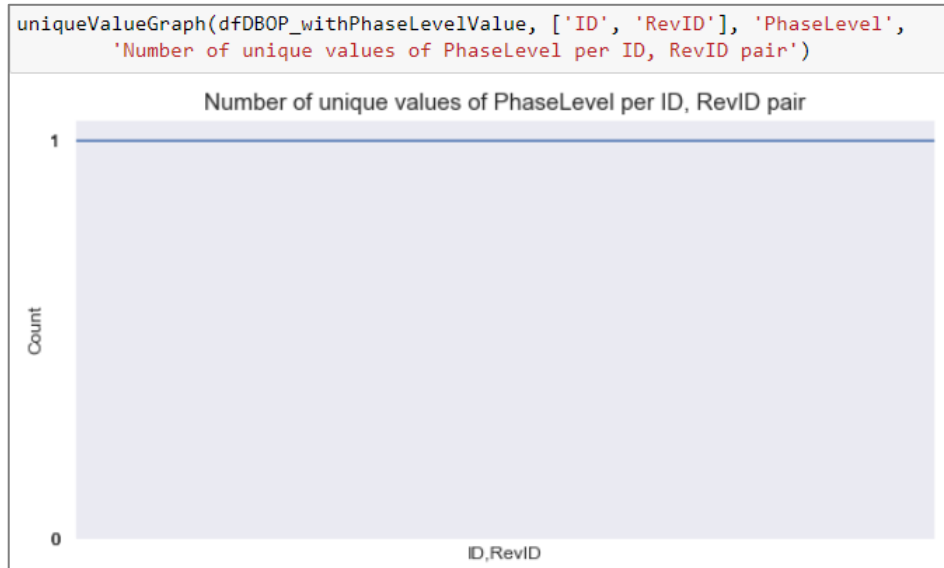


Figure 103. The number of unique PhaseLevel values per ID and RevID pair. (Author's code)

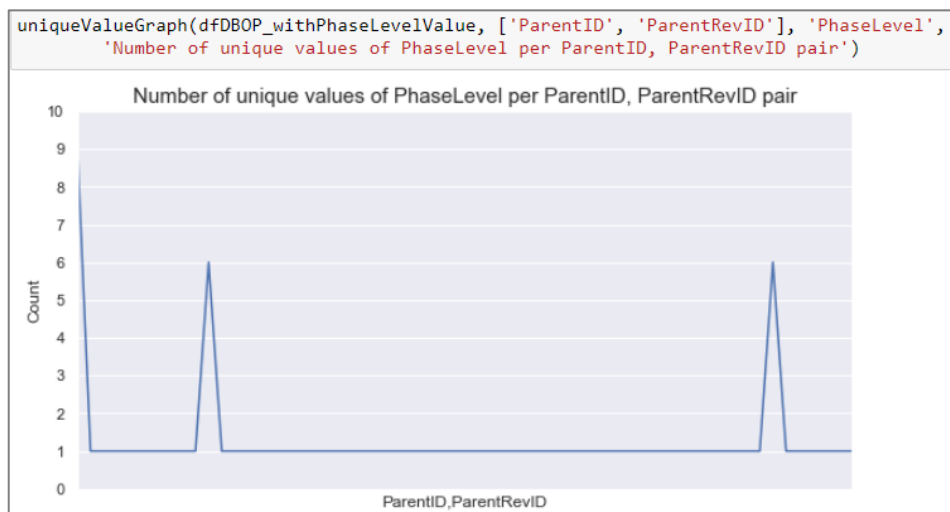


Figure 104. The number of unique PhaseLevel values per ParentID and ParentRevID pair. (Author's code)





Figure 105. The number of unique PhaseLevel values per ParentID and ParentRevID. StrLevel, and Seq\_Nr combination. (Author's code)

When examining the similar number of non-null values for the columns in the DBOP in Figure 31, no data groups are identified for *QualityKey*. Therefore, it was investigated if the *QualityKey* depends on the central elements in the DBOP hierarchical structure, or on some previously identified groups. A separate data frame containing the rows with a *QualityKey* value was created with the code in Figure 106. From this data frame, the columns with only null values were dropped. When looking at the result in Figure 106, it was decided to check the dependency on the following columns:

- *ID* and *RevID* pair
- *ParentID* and *ParentRevID* pair
- *ParentID*, *ParentRevID*, *StrLevel*, and *Seq\_Nr* combination
- *SortString*

The result in Figure 107, Figure 108, Figure 109, and Figure 110 reveals that the *QualityKey* is dependent on one of the following:

- *ID* and *RevID* pair
- *ParentID* and *ParentRevID*, *StrLevel*, and *Seq\_Nr* combination
- *SortString*

The dependency on *ParentID* and *ParentRevID* pairs does not exist, as a pair can return several unique *QualityKey* values.

```

dfDBOP_withQualityKeyValue = dfDBOP.loc[~dfDBOP["QualityKey"].isnull()]

dfDBOP_withQualityKeyValue = dfDBOP_withQualityKeyValue[dfDBOP_withQualityKeyValue.columns[
    ~dfDBOP_withQualityKeyValue.isnull().all()]]
dfDBOP_withQualityKeyValue.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 73 entries, 124 to 417
Data columns (total 20 columns):
ParentID          73 non-null object
ParentRevID       73 non-null object
StrLevel          73 non-null int64
Seq_Nr           73 non-null int64
Qty              73 non-null object
ID               73 non-null object
RevID            73 non-null object
Name             73 non-null object
TcType           73 non-null object
ReleaseDate      73 non-null object
Description       73 non-null object
PhaseLevel       73 non-null object
PlantLevel       73 non-null object
ProcessType      73 non-null object
QualityKey       73 non-null object
AlternateProcess  73 non-null object
OwningUser       73 non-null object
OwningGroup      73 non-null object
PSA              73 non-null object
SortString       73 non-null object
dtypes: int64(2), object(18)
memory usage: 12.0+ KB

```

Figure 106. The code for creating a data frame for the rows with a QualityKey value (Author's code)

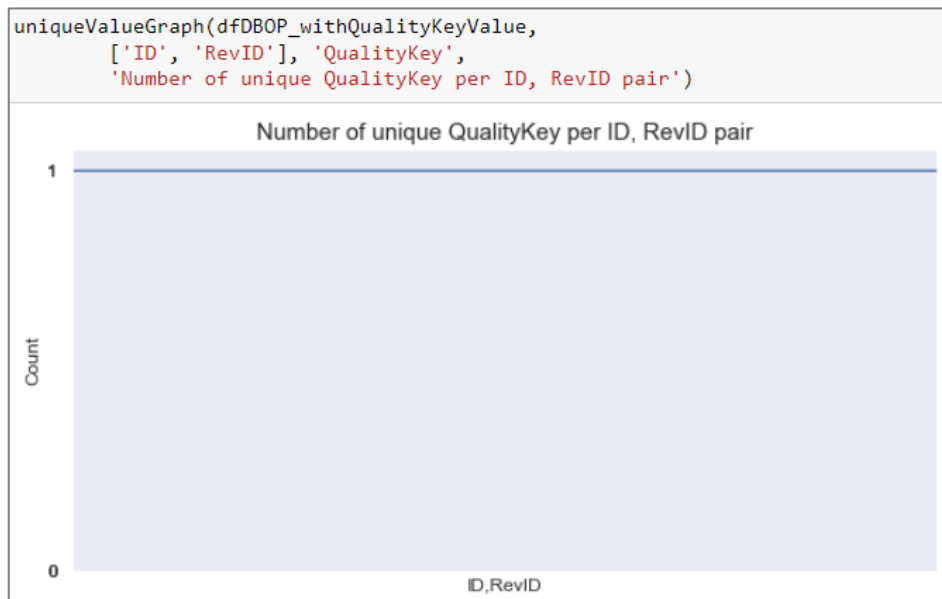


Figure 107 The number of unique QualityKey values per ID and ID pair (Author's code)

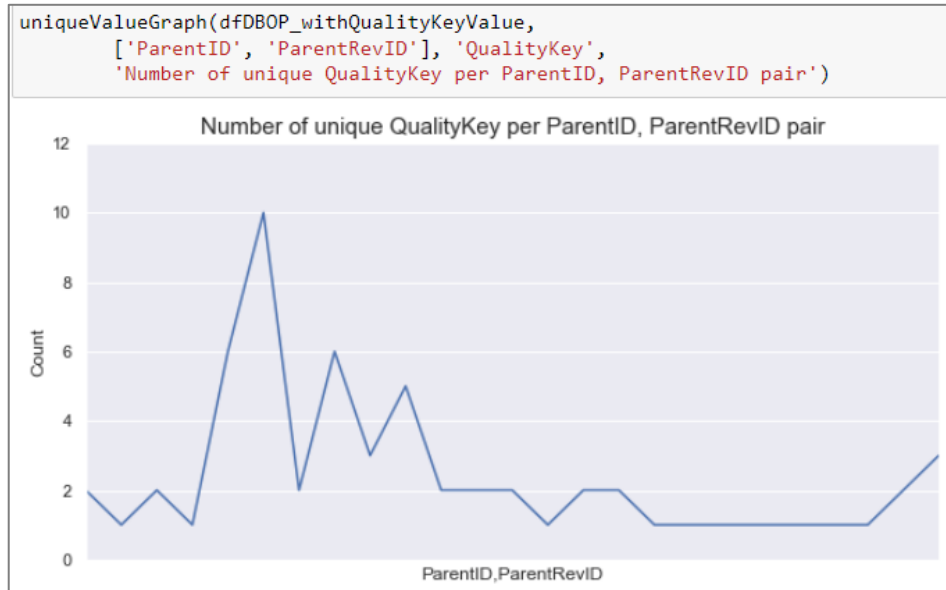


Figure 108. The number of unique QualityKey values per ParentID and ParentRevID pair. (Author's code)



Figure 109. The number of unique QualityKey values per ParentID and ParentRevID. StrLevel, and Seq\_Nr combination. (Author's code)



Figure 110. The number of unique QualityKey values per SortString. (Author's code)

When examining the similar number of non-null values for the columns in the DBOP in Figure 31, no data groups were identified for *ConsumedAssembly*. Therefore, it was investigated if the *ConsumedAssembly* depends on the central elements in the DBOP hierarchical structure, or on some previously identified groups. A separate data frame containing the rows with a *ConsumedAssembly* value was created in the code in Figure 111. From this data frame, the columns with only null values were dropped. When looking at the result in Figure 111, it was decided to check the dependency on the following columns:

- *ID* and *RevID* pair
- *ParentID* and *ParentRevID* pair
- *ParentID* and *ParentRevID*, *StrLevel*, and *Seq\_Nr* combination

The result in Figure 112, Figure 113, and Figure 114 indicates that *ConsumedAssembly* is dependent on one of the following:

- *ID* and *RevID* pair
- *ParentID* and *ParentRevID*, *StrLevel*, and *Seq\_Nr* combination

The dependency on *ParentID* and *ParentRevID* pairs does not exist, as pairs return several unique *ConsumedAssembly* values.

```

dfDBOP_withConsumedAssemblyValue = dfDBOP.loc[~dfDBOP["ConsumedAssembly"].isnull()]

dfDBOP_withConsumedAssemblyValue = dfDBOP_withConsumedAssemblyValue[
    dfDBOP_withConsumedAssemblyValue.columns[~dfDBOP_withConsumedAssemblyValue.isnull().all()]]

dfDBOP_withConsumedAssemblyValue.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 20 entries, 517 to 1315
Data columns (total 12 columns):
ParentID          20 non-null object
ParentRevID       20 non-null object
StrLevel          20 non-null int64
Seq_Nr           20 non-null int64
Qty               20 non-null object
ID                20 non-null object
RevID             20 non-null object
Name              20 non-null object
TcType           20 non-null object
ConsumedAssembly  20 non-null object
OwningUser        20 non-null object
OwningGroup       20 non-null object
dtypes: int64(2), object(10)
memory usage: 2.0+ KB

```

Figure 111. The code for creating a data frame for the rows with a ConsumedAssembly value. (Author's code)

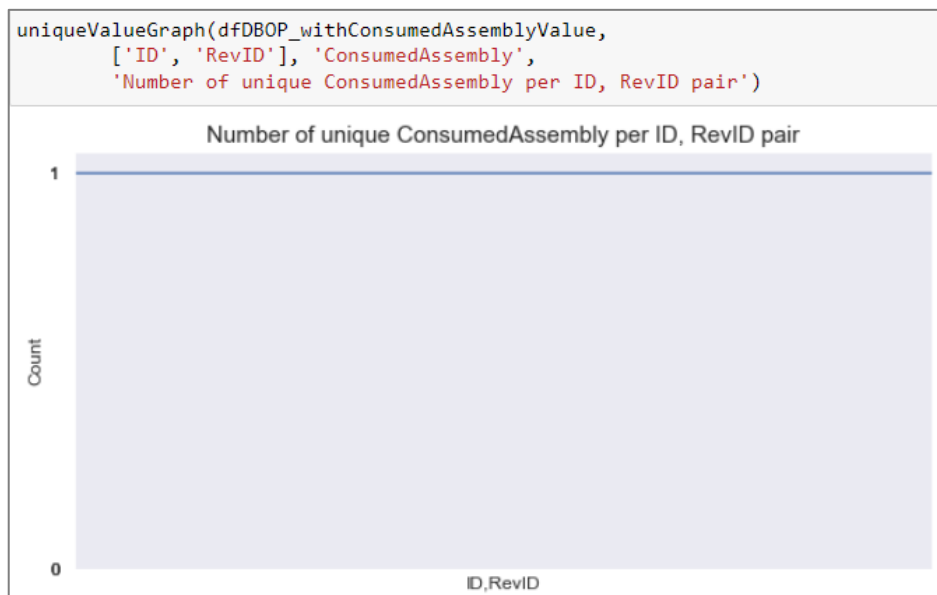


Figure 112. The number of unique ConsumedAssembly values per ID and RevID pairs. (Author's code)

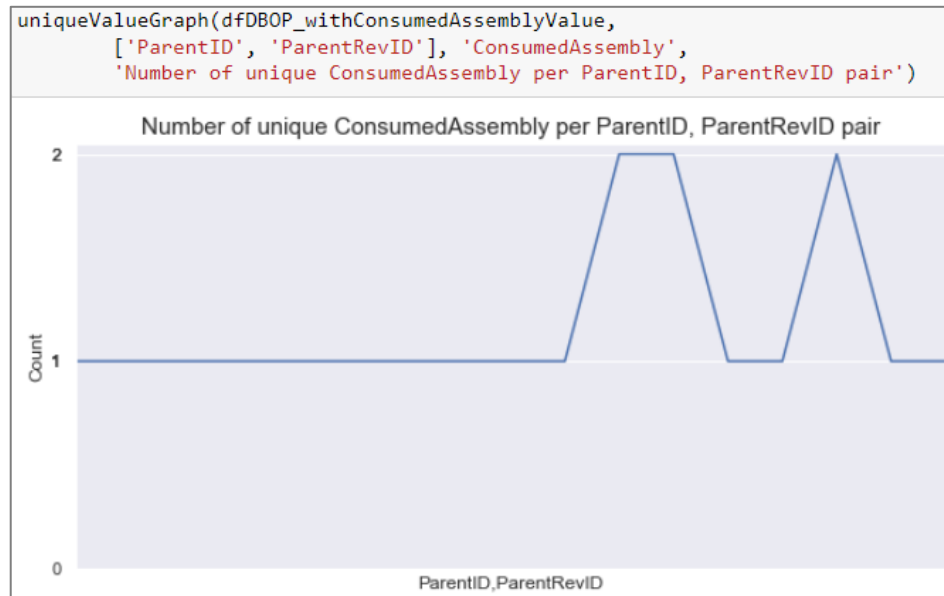


Figure 113. The number of unique ConsumedAssembly values per ParentID and ParentRevID pairs. (Author's code)



Figure 114. The number of unique ConsumedAssembly values per ParentID and ParentRevID, StrLevel, and Seq\_Nr combination. (Author's code)

When examining the similar number of non-null values for the columns in the DBOP in Figure 31, no data groups were identified for *PurchaseCode*. Therefore, it was investigated if the *PurchaseCode* depends on the central elements in the DBOP hierarchical structure, or on some previously identified groups. A separate data frame containing the rows with a *PurchaseCode* value was created in Figure 115. From this data frame, the columns with only null values were dropped. When looking at the result in Figure 115, it was decided to check the dependency on the following columns:

- *ID* and *RevID* pair

- *ParentID* and *ParentRevID* pair
- *ParentID* and *ParentRevID*, *StrLevel*, and *Seq\_Nr* combination

The results in Figure 112 and Figure 113 indicate that *PurchaseCode* is dependent on one of the following:

- *ID* and *RevID* pair
- *ParentID* and *ParentRevID*

The dependency on *ParentID* and *ParentRevID*, *StrLevel*, and *Seq\_Nr* combination exists, as a part of this column combination was already proved to have a dependence on *PurchaseCode* (see Figure 113). Extending an identified key to an additional column did not bring any benefits.

```
dfDBOP_withPurchaseCodeValue = dfDBOP.loc[~dfDBOP["PurchaseCode"].isnull()]

dfDBOP_withPurchaseCodeValue = dfDBOP_withPurchaseCodeValue[dfDBOP_withPurchaseCodeValue.columns[
    ~dfDBOP_withPurchaseCodeValue.isnull().all()]]

dfDBOP_withPurchaseCodeValue.info()

<<class 'pandas.core.frame.DataFrame'>
Int64Index: 46 entries, 1420 to 5237
Data columns (total 15 columns):
ParentID          46 non-null object
ParentRevID       46 non-null object
StrLevel          46 non-null int64
Seq_Nr            46 non-null int64
Qty               46 non-null object
ID                46 non-null object
RevID             46 non-null object
Name              46 non-null object
TcType           46 non-null object
ReleaseDate       46 non-null object
Description        42 non-null object
Ma9_ParentItem    42 non-null object
PurchaseCode      46 non-null object
OwningUser        46 non-null object
OwningGroup       46 non-null object
dtypes: int64(2), object(13)
memory usage: 5.8+ KB
```

Figure 115. The code for creating a data frame for the rows with a *PurchaseCode* value. (Author's code)

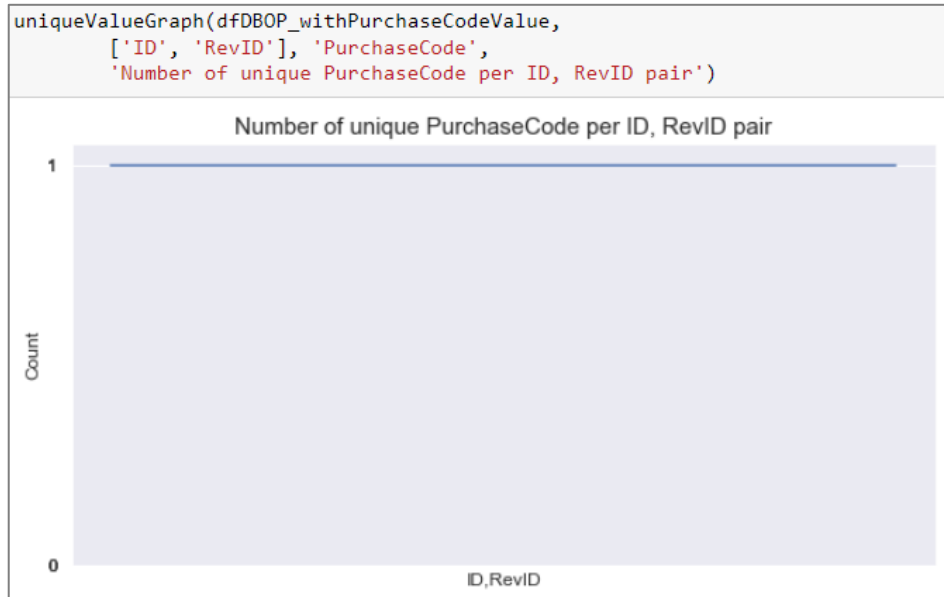


Figure 116. The number of unique PurchaseCode values per ID and RevID pair. (Author's code)

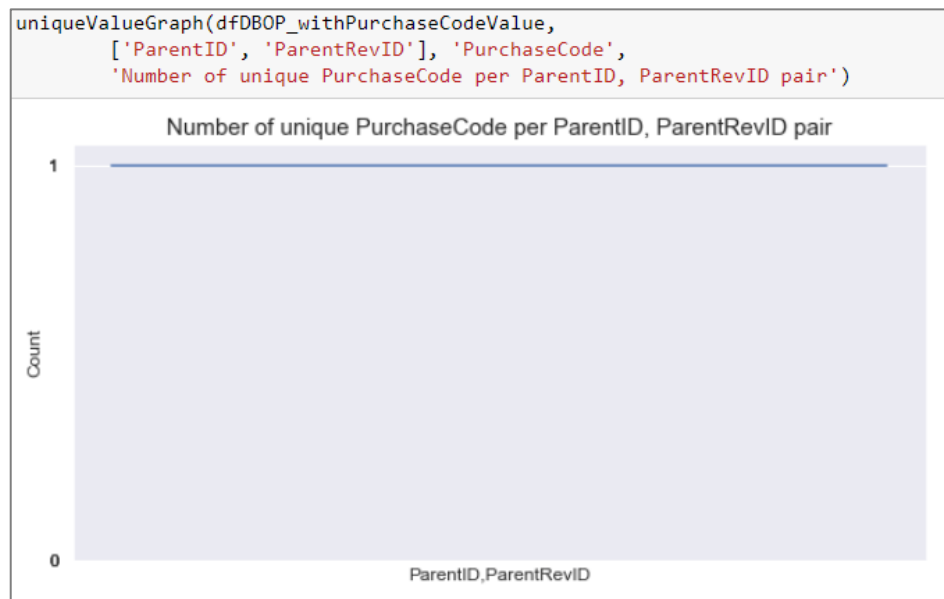


Figure 117. The number of unique ConsumedAssembly values per ParentID and ParentRevID pair. (Author's code)



### 4.2.1.1. Summary

The following is a summary of the findings from the investigation:

- There is no direct relation between *StrLevel* and *TcType*
- There is a group for engine-specific columns where the *EngineNumber* is the key, identifying the *EngineAbbreviation* and *EngineDescription* of this engine. A DBOP will always cover only one specific *EngineNumber*.
- There is no direct relation between *OwningUser* and *OwningGroup*. The graphs reveal that an *OwningUser* can have as many as 8 *OwningGroup*, and in an *OwningGroup*, there can be over 175 *OwningUsers*. These numbers will vary depending on the DBOP investigated.
- Each id type has its own revision.
  - *RevID* is the revision of the *ID*
  - *ParentRevID* is the revision of *ParentID*
  - *RealizationRevisionID* is the revision of *RealizationID*
- The *Ma9\_ParentItem* column also contains ids and revisions. The *Ma9\_ParentItem* is a multivalued attribute in which the id and revision pairs are separated with the | -character, and the revision for the id is split with the / -character.
- The *ID* and *RevID* pair's link to *ParentID* and *ParentRevID* pair does not exist throughout the DBOP structure. Also, the *Ma9\_ParentItem*'s link to the *ParentID* and *ParentRevID* pair needs to be considered.
- The *RealizationID* and *RealizationRevisionID* pairs depend on the *ID* and *RevID* combination. To understand if the *RealizationID* and *RealizationRevisionID* pairs are involved in forming the hierarchical structure, it was investigated if some of the *RealizationIDs* can be found in the *ID*, *ParentID*, or *Ma9\_ParentItem* columns. The *RealizationID* values were not found in any of the investigated columns. Hence, it was confirmed that it is not involved in creating the hierarchical structure.
- When investigating what *DrawingRelation* depends on, a possible data quality error was noticed for three *ID* and *RevID* pairs. The *ID* and *RevID* combination can identify a unique *DrawingRelation* value if the possible data quality error can be corrected.
- A group with *ID* and *RevID* pair as a key for columns with approx. 9000 non-null values were found. The following columns fit into that group: *ID*, *RevID*, *Name*, *TcType*, *ReleaseDate*, *Description*, *OwningUser*, and *OwningGroup*.

- *StrLevel*, *Seq\_Nr*, and *Qty* are not dependent on the *ID* and *RevID* pair. Instead, *StrLevel* and *Seq\_Nr* are assumed to form the DBOP process steps together with *ParentID* and *ParentRevID*. *Qty* is taken to indicate how many repetitions of a specific process step are needed.
- A group of *SortString*, *PlantLevel*, *ProcessType*, *AlternateProcess*, and *PSA* was identified. In this group, *SortString* is the key. *PhaseLevel* cannot be part of this group.
- *PhaseLevel* is dependent on *ID* and *RevID* pair or a combination of *ParentID*, *ParentRevID*, *StrLevel*, and *Seq\_Nr*.
- Three possible dependencies were recognized for *QualityKey*:
  - *ID* and *RevID* pair
  - *ParentID*, *ParentRevID*, *StrLevel* and *Seq\_Nr* combination
  - *SortString*
- *ConsumedAssembly* depends on the *ID* and *RevID* pair, or a combination of *ParentID*, *ParentRevID*, *StrLevel* and *Seq\_Nr*.
- *PurchaseCode* depends on the *ID* and *RevID* pair, or the *ParentID* and *ParentRevID* pair.

## 4.2.2. Logical data models

This chapter presents the design decisions made during the data model creation, and the data collected for the experiment. The logical data models were created based on the findings from the data analysis in chapter 4.2.1. The normalization guidelines up to the 3NF form are followed for the relational data model. The general principle of normalization is to reduce data redundancy [47]. Table 8 summarizes what was considered for the normalization. The graph type used for the graph data model is the labeled property graph. The labeled property graph was selected for the final aim of implementing the graph data model in Neo4j. The elements of the labeled property graph are explained in chapter 2.3.3. Technical and performance requirements are not considered in these logical data models.

Table 8. Guidelines to reach 3NF normalization for the relational data model

<b>1NF</b>	All attributes have a unique name. None of the attributes are composite or multivalued. [47]
<b>2NF</b>	No partial dependencies on a primary key exist in the tables. This means that in a table where the primary key consists of two or more attributes, the non-

	primary key attribute must depend on the complete primary key and not part of it. [48]
<b>3NF</b>	No transitive dependencies on the primary key exist in the tables. For example, if A is dependent on B, B is dependent on C. Hence, A is dependent on C, is a transitive dependency. [49]

In addition to data model creation, the experiment follows the experiment design in chapter 4.1. For the experiment, the data modeling time is measured in minutes. The relational data model is created first. After that, the same data modeler creates the graph data model. The reason for the selected data modeling order is elaborated in chapter 4.1.4.

### **4.2.2.1. Relational data model**

The creation of the relational data model took place on 24<sup>th</sup> of August 2022. The data modeling was carried out in a single session of 74 minutes without disruptions. The result is presented in Figure 118. This chapter explains how this result was reached.

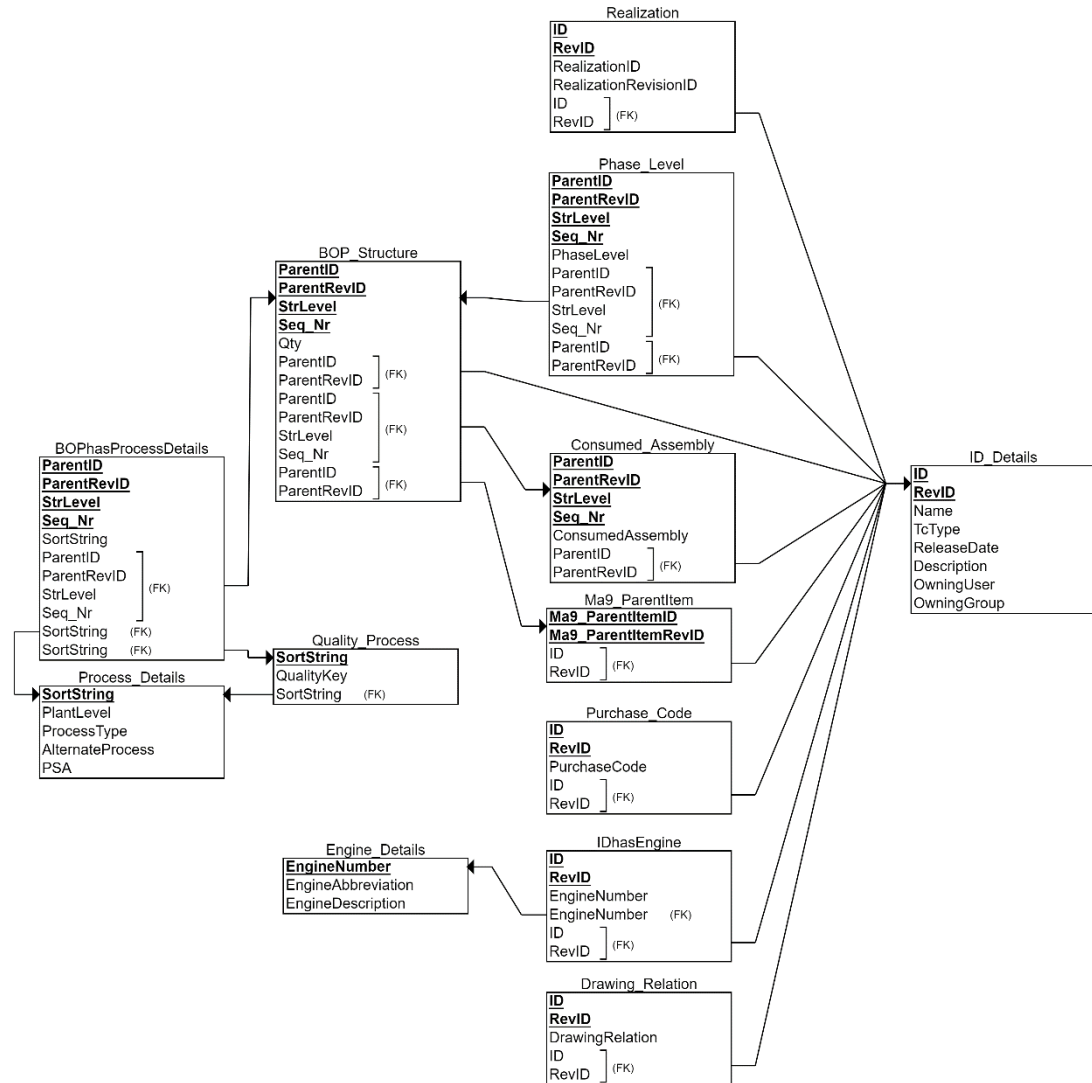


Figure 118. The DBOP relational data model (author's picture)

The data analysis and the summary presented in chapter 4.2.1.1 are an excellent bases for the relational data model. Data dependencies and possible data groups with primary keys are available in the analysis result. Building the relational data model starts with creating tables of the identified data groups. Table 9 summarizes how the data analysis finding is translated into data model design decisions. The intermediate result of the relational data model after the considerations in Table 9 is presented in Figure 119.

Table 9. How the identified data groups are utilized in the relational data model design

Data analysis finding	Data model design decision
<p><b>There is a group for engine-specific columns where the <i>EngineNumber</i> is the key, identifying the <i>EngineAbbreviation</i> and <i>EngineDescription</i> of this engine.</b></p>	<p>Implement an <i>Engine_Details</i> table with three attributes:</p> <ul style="list-style-type: none"> <li>• <i>EngineNumber</i></li> <li>• <i>EngineAbbreviation</i></li> <li>• <i>EngineDescription</i></li> </ul> <p>Of which <i>EngineNumber</i> is the primary key.</p>
<p><b>A group with <i>ID</i> and <i>RevID</i> pair as key, where the following columns are included: <i>Name</i>, <i>TcType</i>, <i>ReleaseDate</i>, <i>Description</i>, <i>OwningUser</i>, and <i>OwningGroup</i>.</b></p>	<p>Implement an <i>ID_Details</i> table with the following attributes:</p> <ul style="list-style-type: none"> <li>• <i>ID</i></li> <li>• <i>RevID</i></li> <li>• <i>Name</i></li> <li>• <i>TcType</i></li> <li>• <i>ReleaseDate</i></li> <li>• <i>Description</i></li> <li>• <i>OwningUser</i></li> <li>• <i>OwningGroup</i></li> </ul> <p>Of which the <i>ID</i> and <i>RevID</i> pair is the primary key.</p> <p>The <i>OwningUser</i> is a multivalued attribute. To fulfill 1NF, the multivalued attribute needs to be split. In the investigation in chapter 4.2.1, it was noticed that the <i>OwningUser</i> values represent the user identification in the <i>Surname</i>, <i>Forename</i> format, or a code like <i>grpadm</i>. As the <i>OwningUser</i> contains a diverse set of <i>Surname</i>, <i>Forename</i> values, and codes, it was decided to treat all the values as codes.</p> <p>An alternative solution that fulfills the 1NF is to create a <i>User_Details</i> table with <i>Surname</i> and <i>Forename</i> attributes. This table would use a synthetic primary key or the employee number if it could be included in the DBOP. The primary key would be named <i>UserID</i> and linked to the <i>OwningUser</i> attribute in the <i>ID_Details</i> table. With this approach, the <i>Surname</i>, <i>Forename</i> values in <i>OwningUser</i> is replaced by the <i>UserID</i> for that specific user.</p>
<p><b><i>StrLevel</i>, <i>Seq_Nr</i>, and <i>Qty</i> are not dependent on the <i>ID</i> and <i>RevID</i> pair. Instead, <i>StrLevel</i> and <i>Seq_Nr</i> are assumed to form the DBOP process steps together with <i>ParentID</i> and <i>ParentRevID</i>. <i>Qty</i> is assumed to indicate how many repetitions</b></p>	<p>Implement a <i>BOP_Structure</i> table with the following attributes:</p> <ul style="list-style-type: none"> <li>• <i>ParentID</i></li> <li>• <i>ParentRevID</i></li> <li>• <i>StrLevel</i></li> <li>• <i>Seq_Nr</i></li> <li>• <i>Qty</i></li> </ul> <p>The primary key is the combination of <i>ParentID</i>, <i>ParentRevID</i>, <i>StrLevel</i>, and <i>Seq_Nr</i>.</p>

<b>of a specific process step are needed.</b>	
<b>A group of <i>SortString</i>, <i>PlantLevel</i>, <i>ProcessType</i>, <i>AlternateProcess</i>, and <i>PSA</i> is identified. In this group, <i>SortString</i> is the key.</b>	Implement a <i>Process_Details</i> table with the following attributes: <ul style="list-style-type: none"> <li>• <i>SortString</i></li> <li>• <i>PlantLevel</i></li> <li>• <i>ProcessType</i></li> <li>• <i>AlternateProcess</i></li> <li>• <i>PSA</i></li> </ul> Of which the <i>SortString</i> is the primary key.

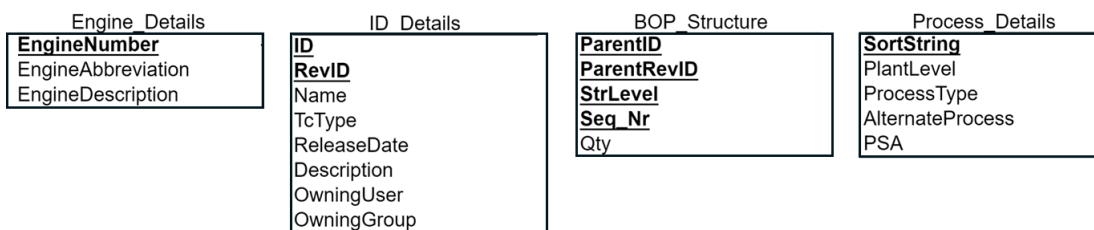


Figure 119. The first intermediate result of the DBOP relational data model (author's picture)

The data dependencies recognized in chapter 4.2.1 were utilized in the next step of the relational data model design. The identified data dependencies are summarized in Table 10. The intermediate result of the relational data model after the considerations in Table 9 and Table 10 is presented in Figure 120.

Table 10. How the identified data dependencies are utilized in the relational data model design

<b>Data analysis finding</b>	<b>Data model design decision</b>
<b>The <i>Ma9_ParentItem</i> is a multivalued attribute consisting of one or several ids and revisions.</b>	Implement a <i>Ma9_ParentItem</i> table with the following attributes: <ul style="list-style-type: none"> <li>• <i>Ma9_ParentItemID</i></li> <li>• <i>Ma9_ParentItemRevID</i></li> </ul> Of which the <i>Ma9_ParentItemID</i> and <i>Ma9_ParentItemRevID</i> pair is the primary key.  The decision to split the <i>Ma9_ParentItem</i> multivalued attribute into rows per id and rev pair and separate the id and revision into dedicated attributes supports the 1NF guideline. Additionally, it makes working with the data in the DBOP easier. For example, the need for string manipulation when linking the <i>Ma9_ParentItem</i> id and revision pair to the <i>ParentID</i> and <i>ParentRevID</i> pair is removed with this design decision.
<b><i>RealizationID</i> and <i>RealizationRevisionID</i></b>	Implement a <i>Realization</i> table with the following attributes:

<p>pairs are dependent on the <i>ID</i> and <i>RevID</i> combination.</p>	<ul style="list-style-type: none"> <li>• <i>ID</i></li> <li>• <i>RevID</i></li> <li>• <i>RealizationID</i></li> <li>• <i>RealizationRevisionID</i></li> </ul> <p>Of which the <i>ID</i> and <i>RevID</i> pair is the primary key.</p>
<p>A possible data quality error in the <i>DrawingRelation</i> attribute for three <i>ID</i> and <i>RevID</i> pairs was noticed in the data analysis. If this is confirmed to be an error, the <i>ID</i> and <i>RevID</i> combination can be used to identify a unique <i>DrawingRelation</i> value.</p>	<p>Implement a <i>Drawing_Relation</i> table with the following attributes:</p> <ul style="list-style-type: none"> <li>• <i>ID</i></li> <li>• <i>RevID</i></li> <li>• <i>DrawingRelation</i></li> </ul> <p>Of which the <i>ID</i> and <i>RevID</i> pair is the primary key.</p> <p>The suspected data quality error is confirmed to be an error, which means that a unique <i>ID</i> and <i>RevID</i> pair can only have one <i>DrawingRelation</i> value.</p>
<p><i>PhaseLevel</i> is dependent on the <i>ID</i> and <i>RevID</i> pair or a combination of <i>ParentID</i>, <i>ParentRevID</i>, <i>StrLevel</i>, and <i>Seq_Nr</i></p>	<p>Implement a <i>Phase_Level</i> table with the following attributes:</p> <ul style="list-style-type: none"> <li>• <i>ParentID</i></li> <li>• <i>ParentRevID</i></li> <li>• <i>StrLevel</i></li> <li>• <i>Seq_Nr</i></li> <li>• <i>PhaseLevel</i></li> </ul> <p><i>ParentID</i>, <i>ParentRevID</i>, <i>StrLevel</i>, and <i>Seq_Nr</i> combination is the primary key.</p> <p>This primary key selection over the <i>ID</i> and <i>RevID</i> pair assumes that the <i>PhaseLevel</i> attribute clarifies a specific process step.</p>
<p>Three possible dependencies are recognized for the <i>QualityKey</i>:</p> <ul style="list-style-type: none"> <li>• <i>ID</i> and <i>RevID</i> pair</li> <li>• <i>ParentID</i>, <i>ParentRevID</i>, <i>StrLevel</i>, and <i>Seq_Nr</i> combination</li> <li>• <i>SortString</i></li> </ul>	<p>Implement a <i>Quality_Process</i> table with the following attributes:</p> <ul style="list-style-type: none"> <li>• <i>SortString</i></li> <li>• <i>QualityKey</i></li> </ul> <p><i>SortString</i> is selected as the primary key, based on an explanation from business stakeholders that the <i>SortString</i> is a unique identifier for a process step that requires a quality check or some other quality measure.</p>
<p><i>ConsumedAssembly</i> depends on the <i>ID</i> and <i>RevID</i> pair or a combination of</p>	<p>Implement a <i>Consumed_Assembly</i> table with the following attributes:</p> <ul style="list-style-type: none"> <li>• <i>ParentID</i></li> <li>• <i>ParentRevID</i></li> </ul>

<p><b>ParentID, ParentRevID, StrLevel, and Seq_Nr.</b></p>	<ul style="list-style-type: none"> <li>• StrLevel</li> <li>• Seq_Nr</li> <li>• ConsumedAssembly</li> </ul> <p>ParentID, ParentRevID, StrLevel, and Seq_Nr combination is the primary key. Any strong reasoning behind designing a table with the ParentID, ParentRevID, StrLevel, and Seq_Nr attributes instead of ID and RevID attributes does not exist.</p>
<p><b>PurchaseCode is dependent on the ID and RevID pair or ParentID and ParentRevID pair</b></p>	<p>Implement a Purchase_Code table with the following attributes:</p> <ul style="list-style-type: none"> <li>• ID</li> <li>• RevID</li> <li>• PurchaseCode</li> </ul> <p>For which the ID and RevID pair is the primary key.</p> <p>The primary key selection assumes that a purchase is related to an activity (ID_Details table) and not to a particular DBOP step (BOP_Structure table).</p>

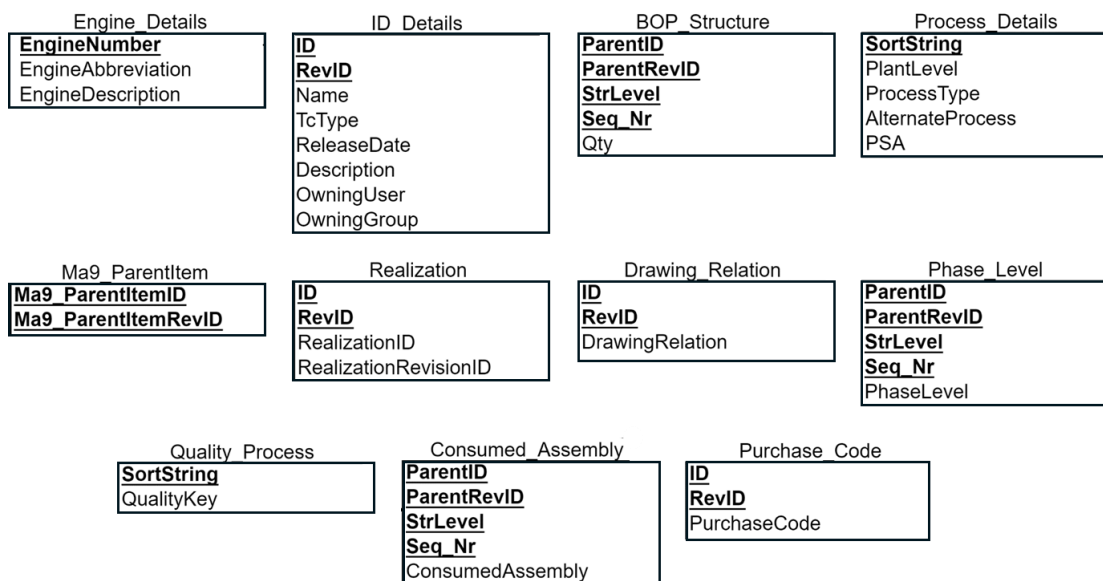


Figure 120. The second intermediate result of the DBOP relational data model (author's picture)

The tables in the relational data model, Figure 120, cover a storage location for each of the DBOP attributes in focus. The relationships between the tables are still missing when compared to the result presented in Figure 118. The relational data model identifies relationships with foreign keys (chapter 2.3.2). Different relationship notations exist depending on the tool used when creating the model.



ERDPlus specifically lists the attributes of the foreign key. The list is automatically created when dragging a relationship line between two tables. If the attributes forming the foreign key already exist in the table with the foreign key, it can be incorrectly assumed that the attributes are duplicated. The *Purchase\_Code* table in Figure 118 is an excellent example of this. When foreign key attributes are not previously identified as attributes in a table, ERDPlus indicates the need when dragging the relationship line between tables. The *MA9\_ParentItem* table in Figure 118 is an excellent example of this.

When creating the relationships in the DBOP model, the *ID\_Details* table and the *BOP\_Structure* table were considered to be the core tables. All other tables in the DBOP are connected to one or both. Based on the data analysis in chapter 4.2.1, it is understood that the *BOP\_Structure* table describes the process structure, and the *ID\_Details* table describes the activity performed at that specific process step. An action can be completed in multiple DBOP process steps.

The *Ma9\_ParentItem* table is another core table. *Ma9\_ParentItem* is directly involved in forming the DBOP hierarchical structure. The most important relationship this table has is to the *BOP\_Structure*. To understand the details of the activity performed, a relationship to the *ID\_Details* is also needed.

Based on the data analysis result in chapter 4.2.1, it is known that:

- A *RealizationID* and *RealizationRevisionID* pair only depend on the *ID* and *RevID* pair. Hence, the only relationship to the *Realization* table is from the *ID\_Details* table.
- *DrawingRelation* is only dependent on the *ID* and *RevID* pair. Hence, the only relationship to the *DrawingRelation* table is from the *ID\_Details* table.

The data analysis results in chapter 4.2.1 reveal that the *PhaseLevel* and the *ConsumedAssembly* have a data dependency on both the *ParentID*, *ParentRevID*, *StrLevel*, and *Seq\_Nr* combination and the *ID* and *RevID* pair. It is assumed that these dependencies are essential and create relationships for the *Phase\_Level* and *Consumed\_Assembly* tables to *BOP\_Structure* and *ID\_Details* tables.

The *PurchaseCode* attribute is identified to have a data dependency on the *ID* and *RevID* pair and the *ParentID* and *ParentRevID* pair. Table 10 elaborates that a purchase is related to a specific activity (*ID\_Details* table) and not to a particular

DBOP step (*BOP\_Structure* table). Based on this, only a relationship from the *ID\_Details* table to the *Purchase\_Code* table is created.

In Figure 120, two tables with *SortString* as the primary key can be identified:

- *Process\_Details* table
- *Quality\_Process* table

The *SortString* attribute is used when creating a relationship between the data in these two tables. The data in these tables are assumed to give details to the DBOP step, and relation to the *BOP\_Structure* is needed. Considering the possible future query needs on the data in the data model, a separate join table is created. The join table is named *BOPhasProcessDetails*, and it has relationships to both the *Process\_Details* table and the *Quality\_Process* table. With this join table, the *SortString* attribute is not needed in the *BOP\_Structure* table. Including the *SortString* attribute in the *BOP\_Structure* table would cause only 333 of the 9210 rows to have a *SortString* value. The number of rows is derived from the number of non-null values per column in Figure 31.

The *Engine\_Details* table is the final table without a relationship. This table contains the data for the engine owning the DBOP. The data analysis result, discussed in chapter 4.2.1, revealed that the same engine information is repeated on three rows in the DBOP. To reduce the data redundancy, a join table named *IDhasEngine* is created between the *ID\_Details* and *Engine\_Details* tables. This table contains three attributes: *ID*, *RevID*, and *EngineNumber*. As the DBOP covers only one *EngineNumber*, the same *EngineNumber* is repeated as many times as there are *ID* and *RevID* pairs relating to the engine data. The *Engine\_Details* table will only have one row. This row specifies the *EngineAbbreviation* and *EngineDescription* for a specific *EngineNumber*.

All the design decisions made when creating the DBOP relational data model are described in Figure 118. The aim was to create a data model that reaches 3NF. Based on the data analysis, discussed in chapter 4.2.1, it is understood which tables are needed in the data model to reduce data redundancy, to avoid partial dependencies, and not to have any transitive dependencies in the data tables. With one exception, the guidelines to reach 3NF were fulfilled, as listed in Table 8. The exception made is for the partly multivalued attribute: *OwningUser*. The reasoning behind this decision is elaborated in Table 9. In short, all the data in *OwningUser* is considered as a code.

### 4.2.2.2. Graph data model

The graph data model is created as the second step. The inputs used for the graph data model are the result of the data analysis, described in chapter 4.2.1, and the DBOP relational data model, presented in Figure 118. Discussions with business stakeholders or data modeling experts were not held between the data modeling sessions for the relational and graph data models.

The creation of the **graph data model** took place on 30<sup>th</sup> of August 2022. The data modeling was carried out in a single session of 48 minutes. The resulting graph data model is presented in Figure 121. This DBOP graph data model and the process used to reach the result was very much disliked by the graph data model expert at Wärtsilä. The business stakeholders did not give any comments on the model.

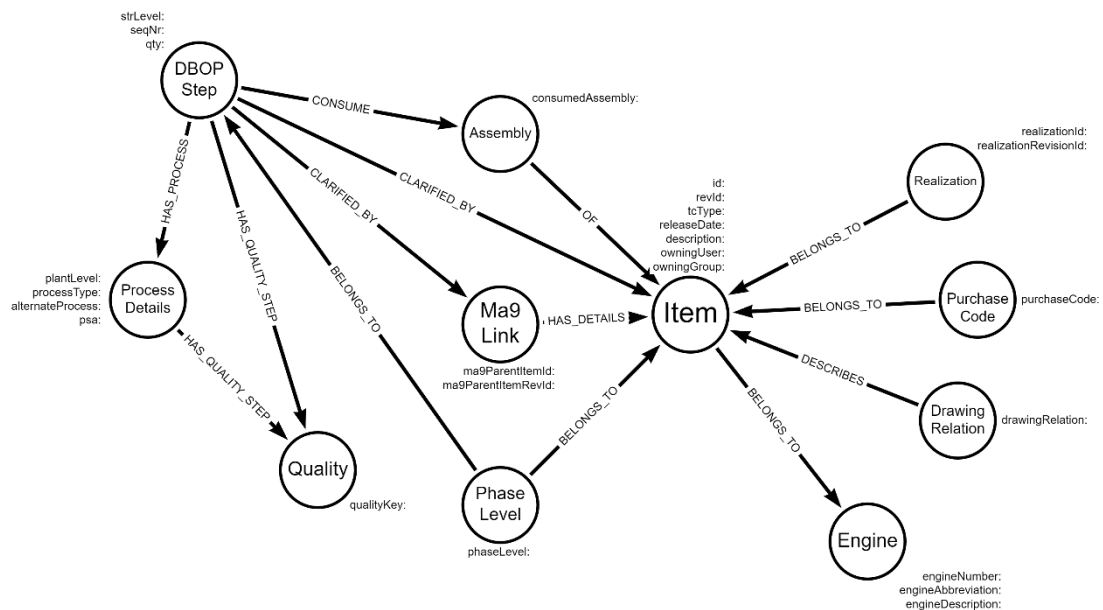


Figure 121. The first DBOP graph data model. The graph data model expert did not accept this version  
(Author's picture)

The process followed to reach the result in Figure 121 is the Neo4j advice on how to move from a relational data model to a graph data model [50]. The guidance suggests analyzing the relational data model with the following sequential steps:

1. Locate all foreign keys
2. Drop all foreign keys
3. Name relations
4. Locate join tables

## 5. Change join tables to relations

Figure 122 presents the result of the analysis made on the DBOP relational data model. This result is then translated into the graph presented in Figure 121. In the graph data model, the tables are nodes. The attributes in the tables are properties on the node. The relationships are named as planned in Figure 122. All the table attributes were defined as node properties for the simplicity of the first model. If this approach would have been continued, keeping the properties on the nodes, moving some of them to the edges or even separating them as separate nodes would have been considered.

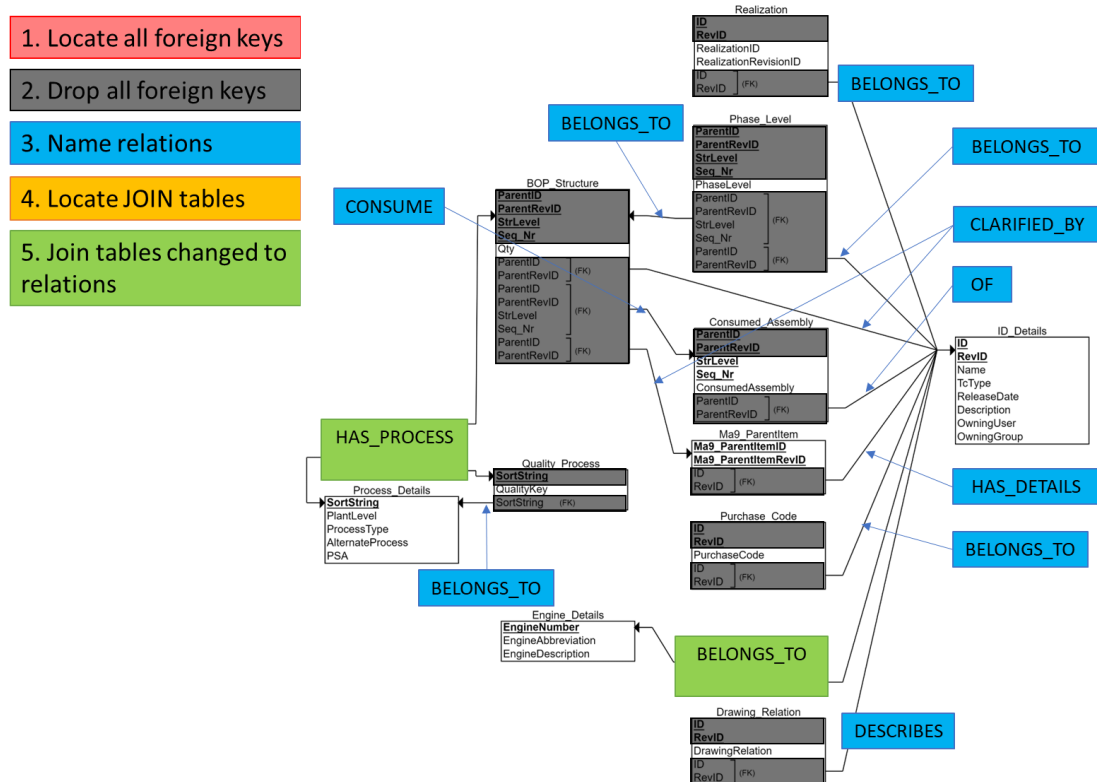
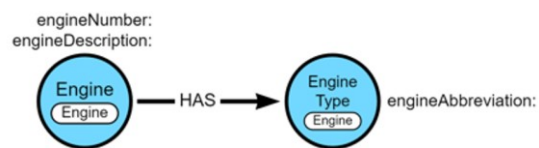


Figure 122. The result of the relational data model analysis (Author's picture)

The graph data model expert at Wärtsilä commented that carrying out the systematic relational data model analysis gives some insight into the data. However, it is not enough to make a well-functioning graph data model. Aligned with the suggestions in chapter 2.4, he marks the importance of discussing with business stakeholders to understand the actual business process and needs. He also emphasizes the art of designing an intuitive graph data model for the people using it. And that the DBOP model should withstand changes in the

manufacturing process and have the flexibility to cover the DBOP for different products.

To reach a better DBOP graph data model, the graph data model expert suggests focusing on *TcType*. The *TcType* values give an indication of which nodes are needed. The suggestion of creating separate nodes for properties assumed to be good link points between different engine DBOPs was immediately turned down. A concrete example of this is presented in Figure 123, which suggests creating a node for the engine-type data. The idea is that this node links all DBOPs with a specific engine type. Based on the data graph data modeling expert's experience, this causes so-called "super nodes" with millions of edges that reduce the performance of the graph database. A better approach is to keep the attributes from the relational data model as properties on nodes or edges.



*Figure 123. The suggestion of breaking out properties assumed to be good linking points between different engine's DBOP is not a good idea (Author's picture)*

The second graph data model was created on the 1<sup>st</sup> of September 2022, with a total modeling time of 180 minutes from two separate sessions. The inputs used for the modeling are the feedback and advice from the graph data modeling expert and the DBOP Excel file. The only information taken from the data analysis in chapter 4.2.1 is the knowledge of how the hierarchical structure is built with the principle in Figure 60. The resulting graph data model is presented in Figure 124.

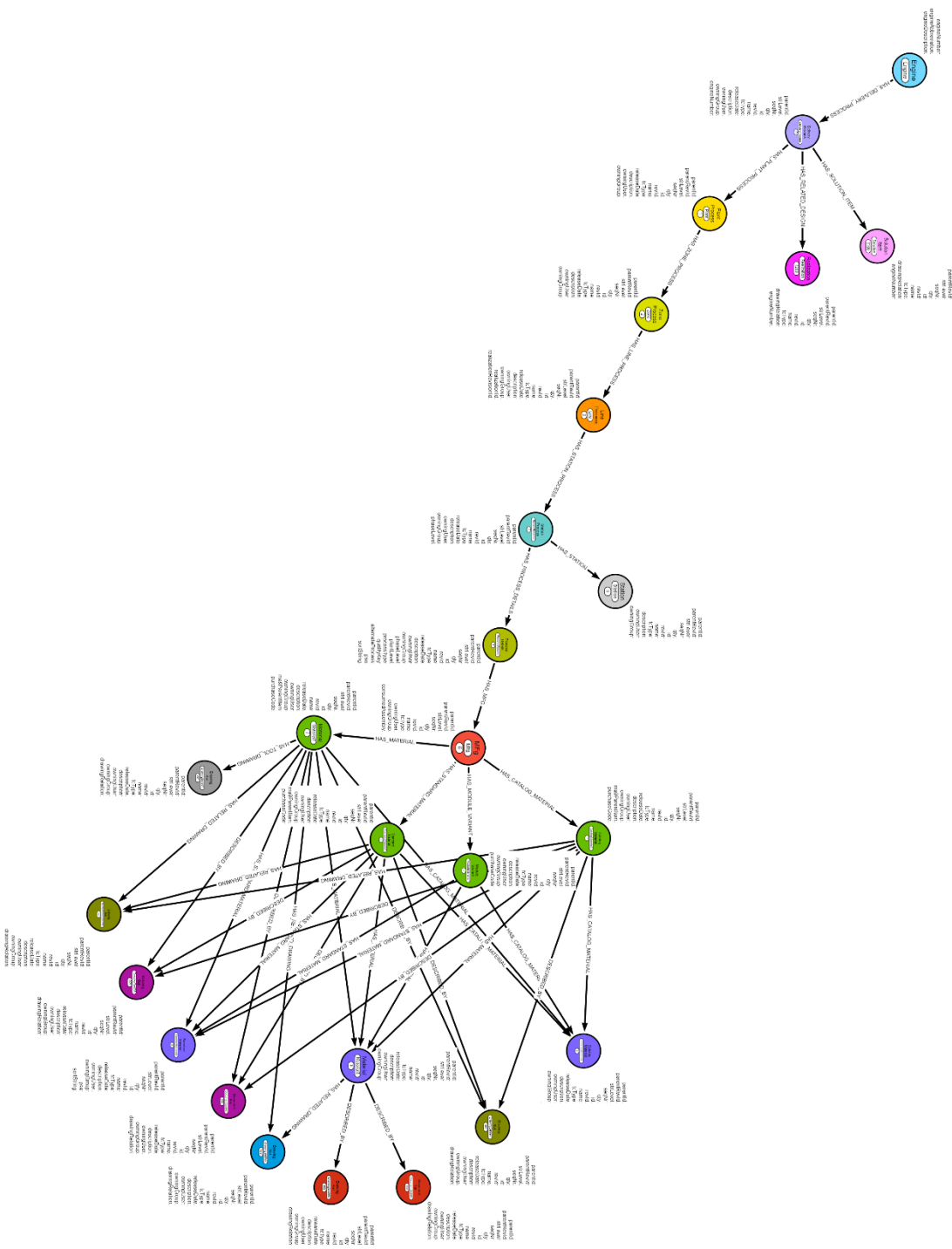


Figure 124. The second DBOP graph data model. The graph data model expert did not accept this version (Author's picture)

The node name is a simplification of the *TcType* value. The properties on the nodes are the column names of the columns with non-null values for the specific *TcType*. The labels on the nodes are a further simplification of the node name and the *StrLevel* value the specific *TcType* value has. The edges are named utilizing the

node name, the direction to which the arrow points to. An alternative naming approach would have been simply to use “HAS” for each edge. A more complicated naming approach was selected to later distinguish the edges in Neo4j.

Figure 125 presents the first part of Figure 124. This part of the created graph data model is aligned with how the graph data model expert and business stakeholder see the DBOP graph data model. The second part, Figure 126, was disliked by both parties. The second part is too complicated and focuses too much on the DBOP data structure.

The simplification of the second part was done in two steps. First, the different material type nodes at *StrLevel 7* were combined into one *Material* node. This also reduces the number of needed edges. The result of the first simplification step is presented in Figure 125. Next, the *Material* node at *StrLevel 7* and the *Material* node at *StrLevel 8* are combined, and the duplicates of *DocumentSet* nodes and *DrawingSet* nodes are removed. The result of the second simplification step is presented in Figure 128. The simplification of the graph data model took place on the 2<sup>nd</sup> of September 2022, with a modeling time of 25 minutes. The complete final DBOP graph data model is presented in Figure 129. By simplifying the graph data model from 26 to 16 nodes and from 41 to 17 edges, a result aligned with the expectations of the business stakeholder and graph data model expert was reached. Twenty minutes was included to the *AnalysisTime(graph)* variable from the additional discussions with the graph data modeling expert and business stakeholder

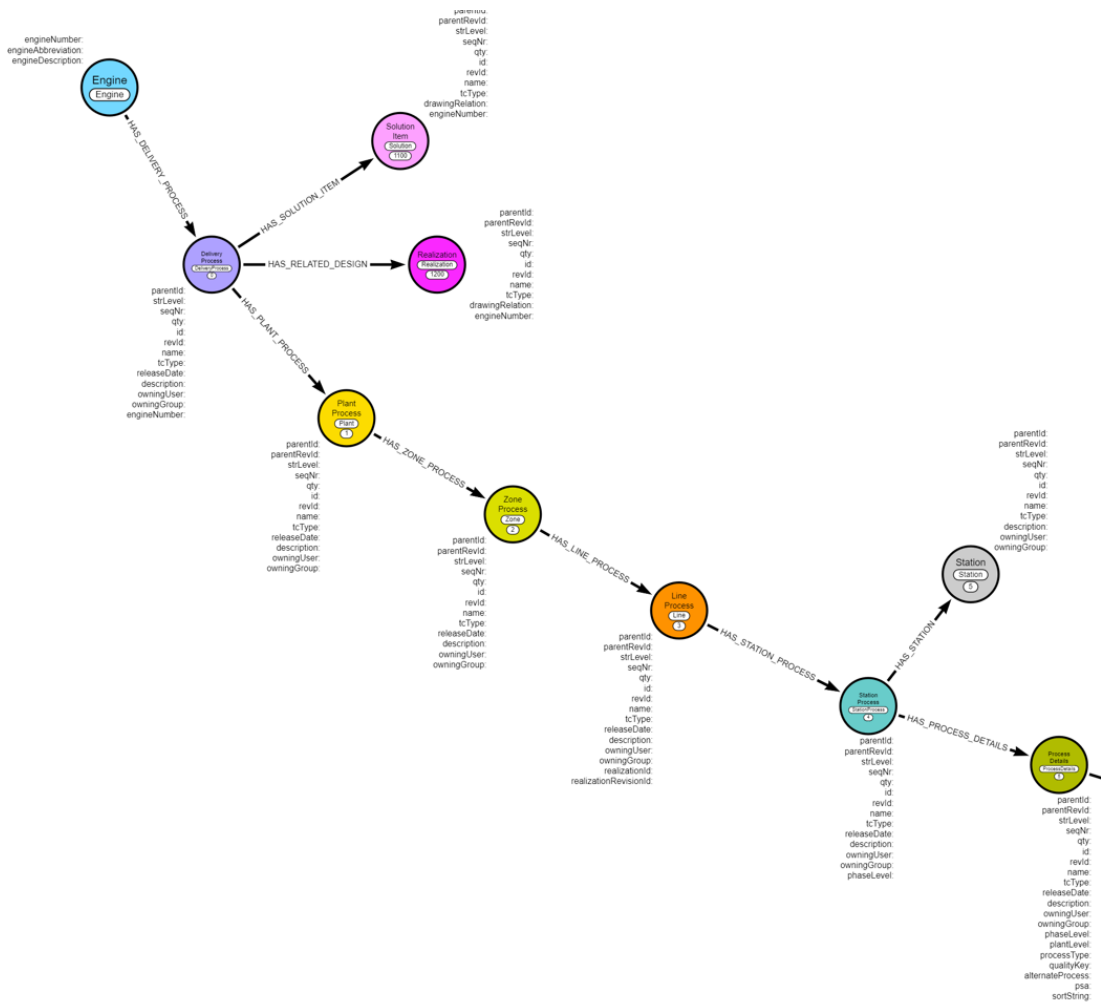


Figure 125. The first part of the second DBOP graph data model. This part was accepted by the graph data model expert and the business stakeholder (Author's picture)





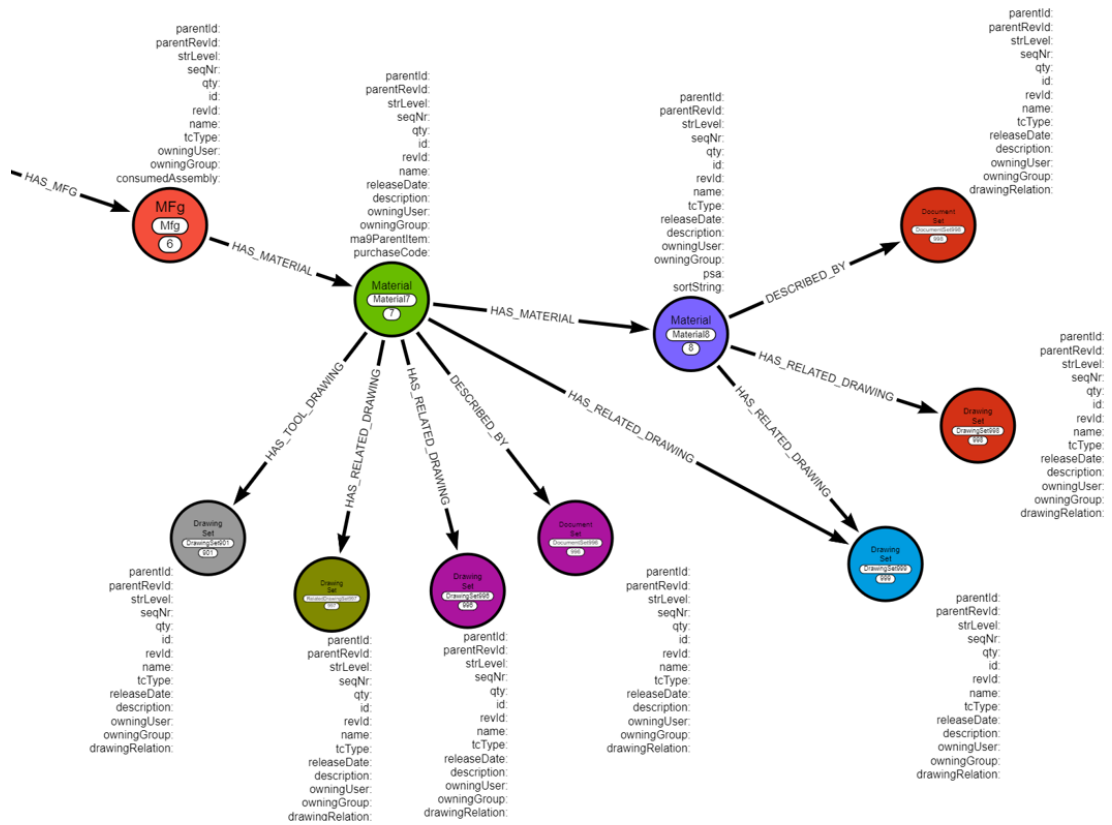


Figure 127. Different material nodes are combined in the second part of the second DBOP graph data model. (Author's picture)

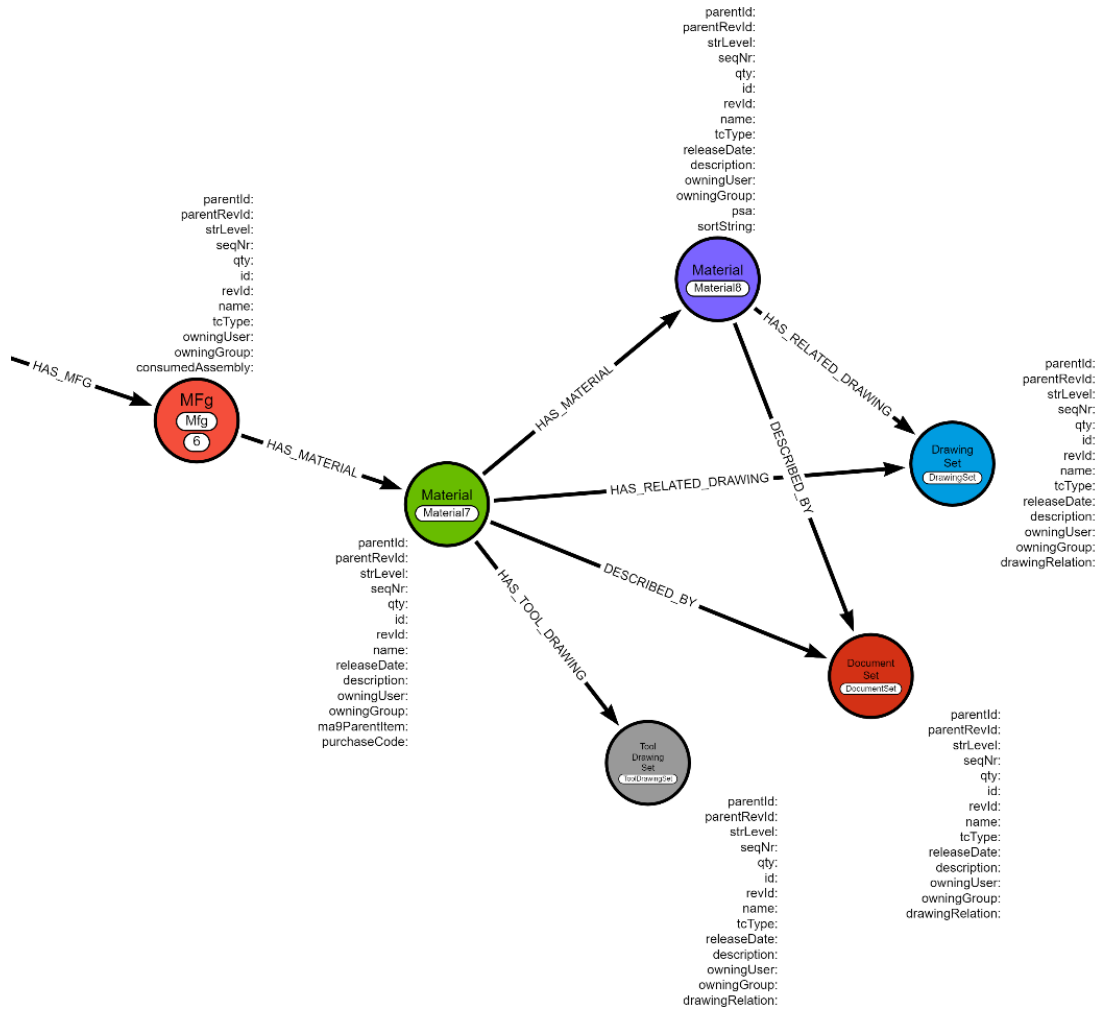


Figure 128. Material nodes are combined and duplicates of Drawing Set nodes and Document Set nodes are removed. (Author's picture)

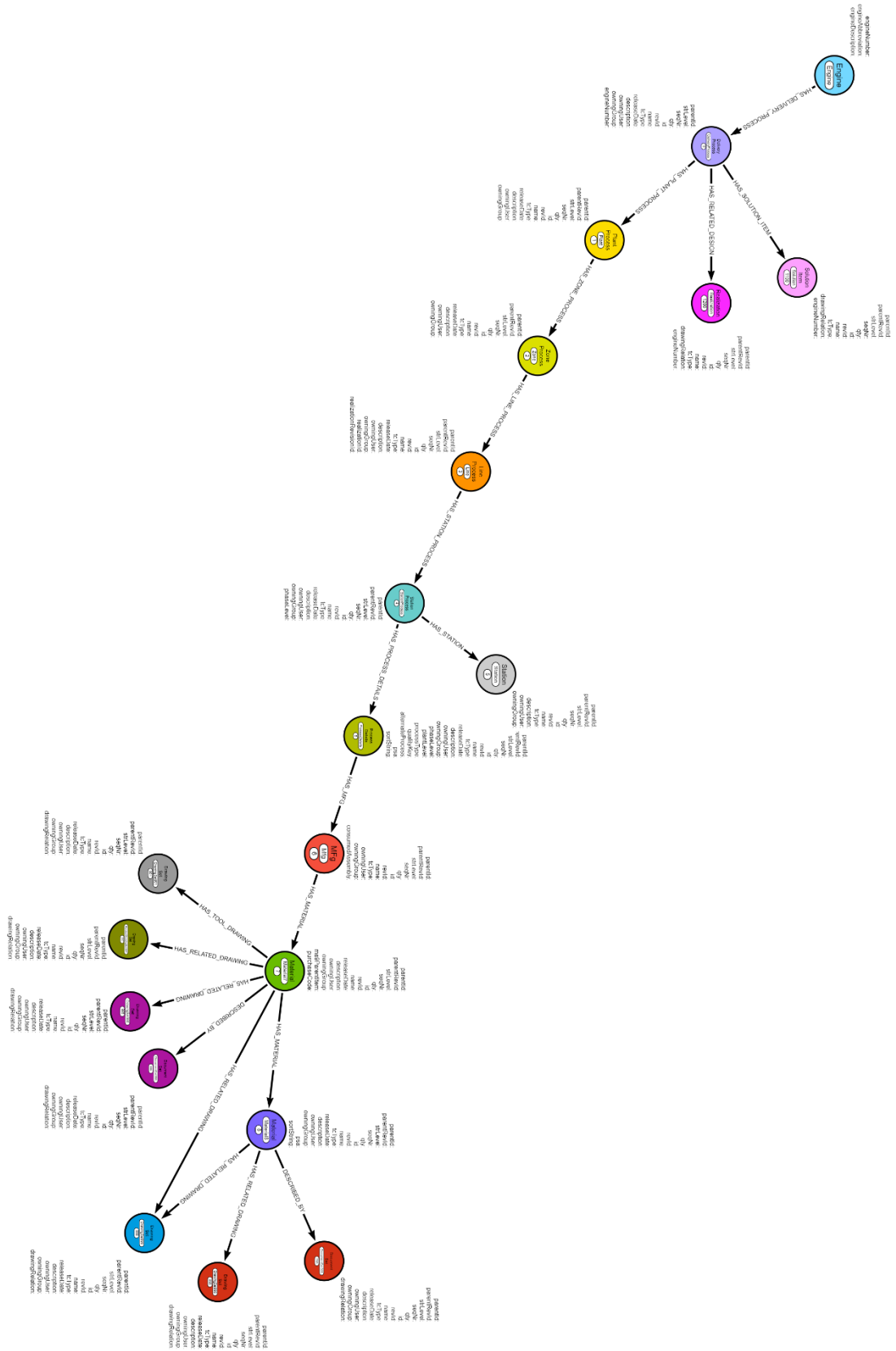


Figure 129. The final DBOP graph data model (Author's picture)

### 4.3. Graph data model implementation in Neo4j

This chapter presents the steps to bring a graph data model created in **arrows.app** [43] to the Neo4j desktop version.

A new project is created in Neo4j, and within this project a new DBMS is added. The project is named as *DBOP\_experiment*, and the DBMS is *DBOP Graph DBMS*, as presented in Figure 130. The DBMS is started and opened in the Neo4j Browser.

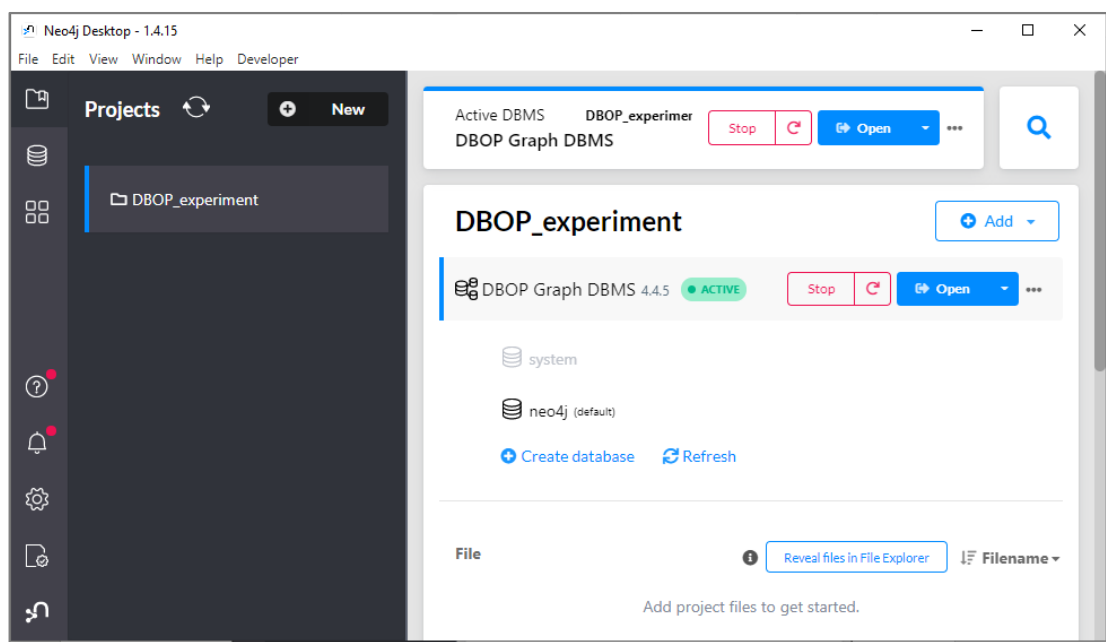


Figure 130. The Neo4j Desktop with the DBOP Graph DBMS running (Author's picture)

The arrows.app export functionality is utilized to copy the Cypher CREATE statement (Figure 132), from the arrows.app to the Neo4j.

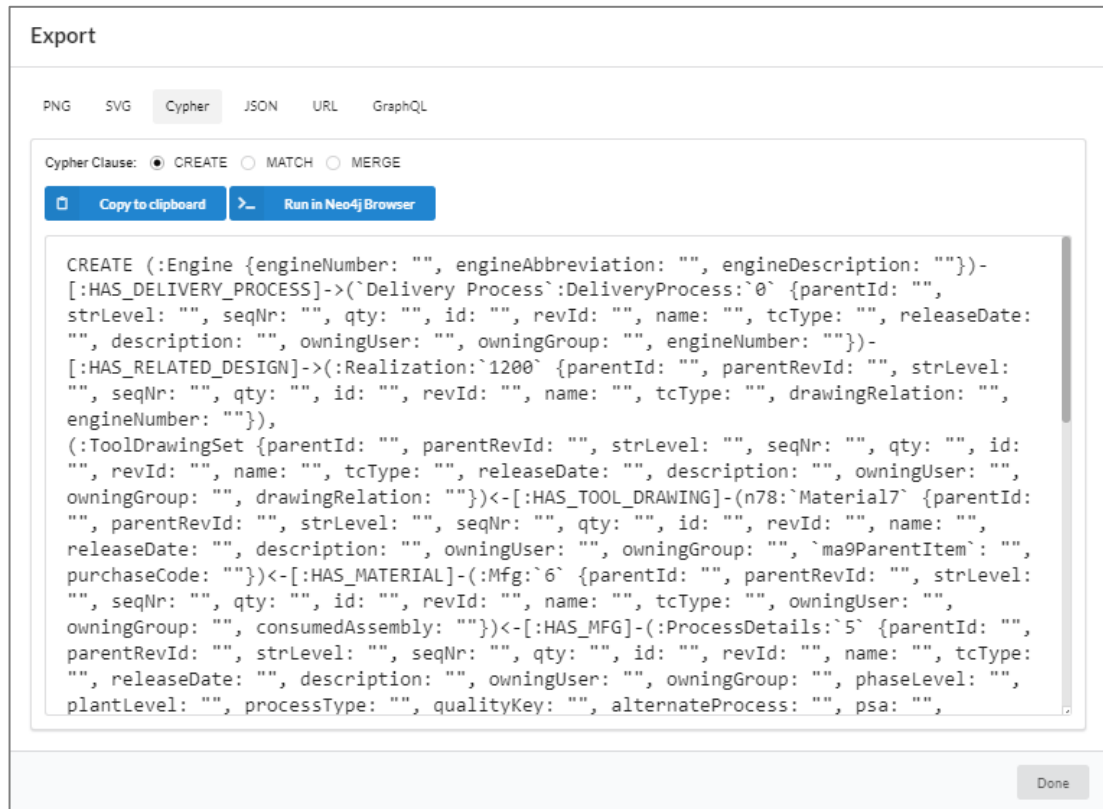


Figure 131. Using the arrows.app export functionality to copy the Cypher CREATE statement (Author's picture)

The Neo4j Browser is opened from the Neo4j Desktop and pasted into the Cypher CREATE statement, as presented in Figure 132. After pressing the play icon in blue, the statement is executed, and the graph is created in 261 ms, as presented in Figure 133.

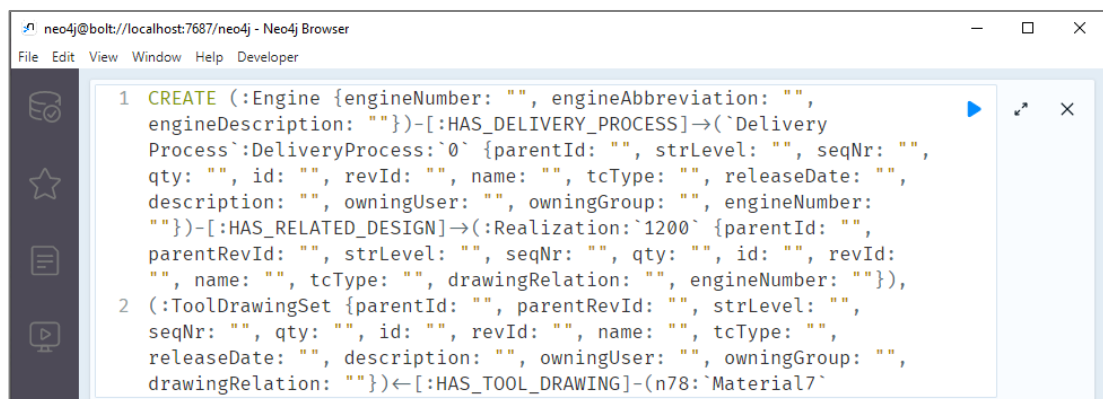


Figure 132. The Cypher CREATE statement is pasted to the Neo4j Browser window (Author's picture)

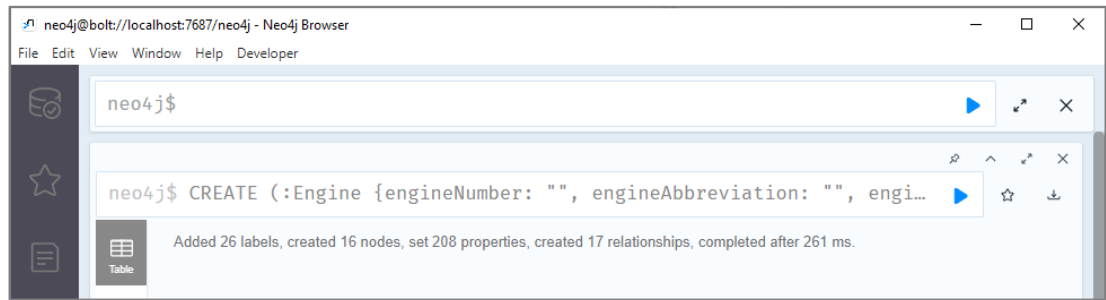


Figure 133. The DBOP graph created in Neo4j in 261 ms (Author's picture)

Figure 134 demonstrates that the graph that was created in the arrows.app is now available in Neo4j. Properties and labels are visible by pressing a specific node, as presented in Figure 135.

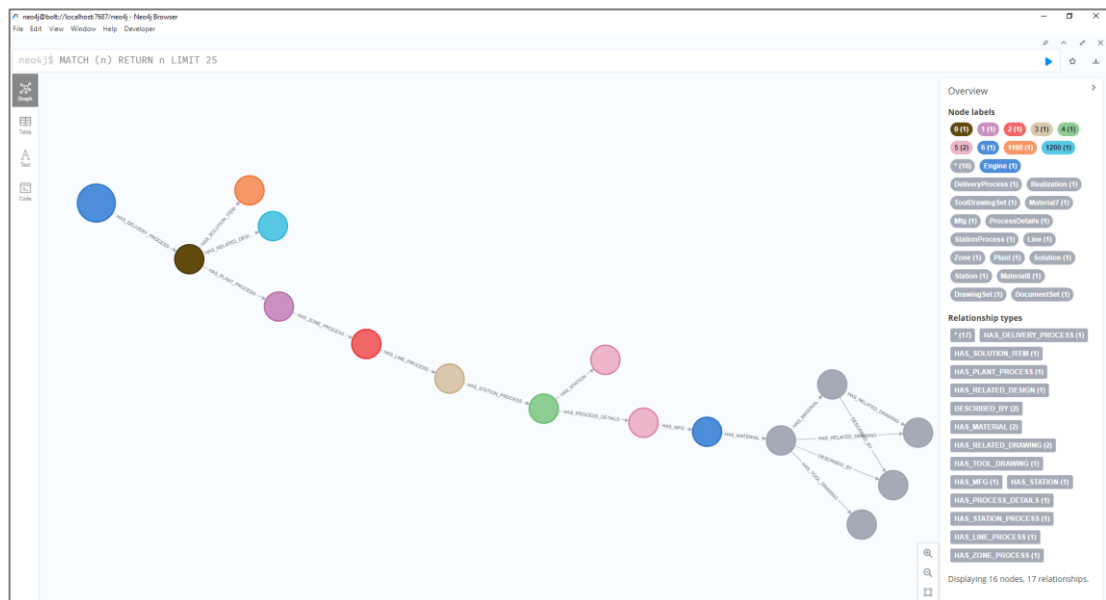


Figure 134. The DBOP graph model in Neo4j (Author's picture)

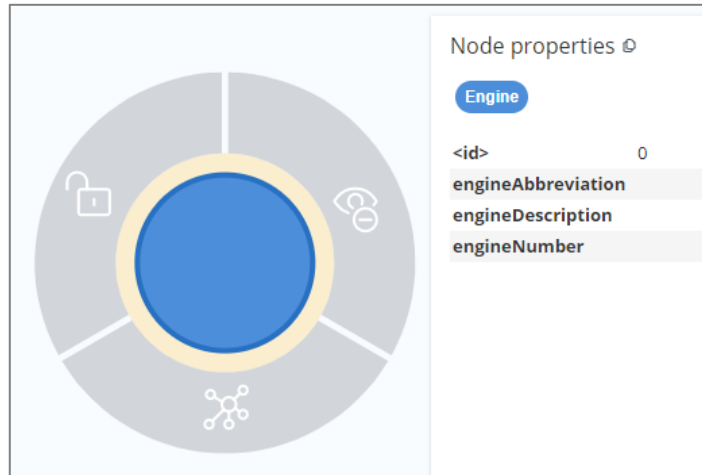


Figure 135. Viewing the label and properties of the Engine node in Neo4j (Author's picture)

With these few simple steps, the graph data model created in the arrows.app can be imported to Neo4j.



## 5. ANALYSIS

This chapter analyses the lessons and results of the research. Competencies and knowledge needed for graph data modeling in Neo4j are also suggested.

### 5.1. Case study result

In the experiment design phase, discussed in chapter 4.1, the alternative hypothesis defines the graph data model as more dynamic than the relational data model. The case study indicates the graph data model to be more dynamic than the relational data model. The availability of only one subject for the case study restrict us to give a statistically relevant result. Despite this fact the result of the case study is calculated according to the planned experiment analysis procedure. This provides an example for how we suggest the result of an future experiment to be analyzed.

The data collected in the experiment are summarized in Table 11. The alternative hypothesis,  $H_{1\_create}: CreateEff(graph) > CreateEff(relational)$ , is formed by inserting the values:  $21.4 > 10.4$ .

Table 11. The experiment result

Name	Value	Description
<i>GraphExp</i>	2	Followed a course or read a book
<i>RelationalExp</i>	4	More than six months of industrial experience.
<i>AnalysisTime(relational)</i>	547 min	Total time from the initial analysis. No additional research is needed during data model implementation.
<i>AnalysisTime(graph)</i>	90 min	70 minutes of initial analysis + 20 minutes during data modeling.
<i>CreateTime(relational)</i>	74 min	Figure 118 was created in one single session.
<i>CreateTime(graph)</i>	253 min	48 minutes for the first version, Figure 121. 180 minutes for the second version, Figure 124. 25 minutes of simplification to reach the final model in Figure 129.
<i>Elements(relational)</i>	29	13 tables

		16 relations
$Elements(graph)$	33	16 nodes 17 edges
$CreateEff(graph)$	21.4	$(547 + 74) / 29 = 21.4137931$
$CreateEff(graph)$	10.4	$(90 + 253) / 33 = 10.394$

The **Wilcoxon** test was used for analysis. The significance level of 0.05 was used to consider the result significant. **R-Studio** [51], the open-source and professional software for data analysis, is used for the **Wilcoxon** test. The code created in R-Studio is presented in Figure 136, and the result is given in Figure 137.

```
##Inserting values to variables
createEff_graph = 21.4137931
createEff_relational = 10.394

##using the test "wilcoxon_test" in "coin" library
#to compute the Z statistic and the p-value
install.packages("coin")
library(coin)

##Hypothesis testing
testResult= wilcoxon_test(createEff_graph ~createEff_relational)
testResult
```

Figure 136. The code in R-Studio for Wilcoxon (Author's code)

```
> ##Hypothesis testing
> testResult= wilcoxon_test(createEff_graph ~createEff_relational)
> testResult

Asymptotic Wilcoxon-Pratt Signed-Rank Test

data: y by x (pos, neg)
Z = 1, p-value = 0.3173
alternative hypothesis: true mu is not equal to 0
```

Figure 137. The result of Wilcoxon analysis in R-Studio (Author's picture)

A key finding in the case study was that a change of mindset of the data modeler is needed when moving from the relational domain to the graph domain. As suggested in chapter 2.4.1, the questions to ask from the model need to be understood already when forming the understanding of the data model needs. It is not enough to use an analytical approach to translate a relational data model to a graph data model. Instead, the policy must be to hold discussions and to align with business stakeholders. Hence, the approach used in this experiment was wrong. It was too focused on the data and lacked the understanding of the conceptual world of the data at hand.

The analysis time needed for the relational data model versus the graph data model indicates the effort required when change needs arise due to changing business needs. The relational data model analysis took six times longer than the graph data model analysis.

When designing the experiment, it was expected that the ratio of the dynamic capabilities of the relational data model versus the graph data model would remain the same no matter the data in the scope and the involvement of the business stakeholders. After the experiment, this assumption was recognized to be wrong. Producing a 3NF relational data model based on a data analysis result is relatively easy. The data analysis did not benefit the graph data model creation greatly. Creating a well-functioning and intuitive graph data model requires more time sorting out the problem with the business stakeholders.

When discussing the data models with the business stakeholder and the data modeling expert, it was noted that neither of them provided any comments regarding the relational model result. For the graph data model, criticism that the first attempts were not aligned with their expectations was received immediately. This indicates that the graph data model is easier to discuss and align with business stakeholders and other data modelers. This confirms that the graph data model is more intuitive than the relational data model and reduces the gap between the conceptual world and the model implemented in a DB. This gap was discussed in chapter 3.6 and visualized in Figure 22.

## 5.2. Competencies and knowledge

This chapter answers **RQ4**:

- What knowledge and competencies are needed for graph data modeling and implementation in Neo4j?

The question is answered based on experience gained through the literature review and case study.

The literature review revealed that the graph data model is often described as more dynamic and intuitive than the relational data model. From the experiment described in chapter 4.2.2, it can be noticed that an engineer from the relational

domain focuses too much on data structures and specific details, which makes the graph data model too complex and difficult to understand.

Creating a simple and intuitive graph data model requires discussions with business stakeholders over extensive data analysis. This can be a challenge from two different angles:

- Time and commitment from business stakeholders
- Social skills and attitude of the data modeler

From the experience gained in this study, it was noticed that a 30-minute session with business stakeholders could be enough to draw the graph whiteboard model that can be further enhanced by either the data modeler alone or with business stakeholders. This study's almost 10-hour long data analysis benefits the relational data model but not the graph data model. Hence, the attitude of making it alone needs to be forgotten, and the engineer needs to engage in discussions with business stakeholders.

The engineer starting graph data modeling needs a basic understanding of graph data modeling, tools, and practices, and a willingness to continuously study and learn more. The engineer should also have enough experience to determine if there will be a benefit in moving from the existing DB to the graph DB. This decision should not be made without analysis and careful consideration.

## 6. CONCLUSION AND FUTURE RESEARCH

This chapter concludes the thesis by revisiting the research questions to understand if the research objective is met and provides suggestions for future research.

### 6.1. Answer to research questions

The research was initiated by Wärtsilä's desire to understand if its experienced success with graph data models and Neo4j GDBMS could be extended to the manufacturing process data and later to value creation in internal and partner networks. The literature review revealed that the graph DB seems to be a good choice for manufacturing collaboration in internal and partner networks where the relations will play a vital part and frequent evolvments in the systems can be expected. However, there are no clear indications of immediate and remarkable practical benefits in areas like query performance, flexibility, and agility. The suggestion from Robinson et al. [5] is to sort out what the specific problems to solve is, before deciding to move from a familiar DB type to the graph DB.

Modeling the DBOP as a graph data model shows that the data structure has eleven dept levels, Figure 129. The literature review revealed an indication that when there are more than four dept levels, the graph DB will show remarkably better query performance than the relational DB, as presented in Figure 19. Based on this, it is understood that if the DBOP is the core data in the manufacturing collaboration in the internal and partner network, the graph DB is a suitable choice.

From the case study of implementing the DBOP relational and graph data model it was noticed that the graph data model seems more dynamic than the relational data model. It is also more intuitive and easier to align with business stakeholders and other data modelers. A good measure of intuitiveness is how easy the model is to discuss with other stakeholders. From the two alternative models created, there was no feedback on the relational data model. At the same time, the business stakeholder and the graph data modeling expert could give their opinions on the graph data model. It was also noticed that **the business stakeholder cannot comment on a graph data model that is too data-focused and complex.**

This research focused on finding solutions and answers to the following research questions (RQ):

- RQ1.** How do data modeling for a relational DB differ from data modeling for a graph DB?
- RQ2.** Can an experiment where the manufacturing process is modeled as a relational model versus a graph data model prove that the graph data model is more dynamic than the relational data model?
- RQ3.** How to present the manufacturing process data in Neo4j?
- RQ4.** What knowledge and competencies are needed for graph data modeling and implementation in Neo4j?

RQ1 is answered based on a literature review and the practical implementation of relational and graph data models. Chapter 2 and chapter 3 collect the findings from the literature review on a level that anyone interested in data modeling will find easy to understand. Especially business stakeholders without experience in data modeling and data modelers moving from the relational domain to the graph domain will benefit from reading these chapters.

A key finding in the literature review is that **a company considering a switch to the graph domain should not only follow the hype of moving to graphs**. A careful analysis of their specific needs is recommended. In case a company faces the challenge of choosing between a relational DB or graph DB, it is suggested to use the recommendations in chapter 3.5, where the Bechberger and Perryman decision tree is presented in Figure 21.

For RQ2 an experiment has been designed. The experiment is not carried out due to resource issues. A case study to evaluate the experiment design however indicates that the graph data model is more dynamic than the relational data model. The data being modeled in the case study is the DBOP of an engine manufactured in Wärtsilä STH. Chapter 5 provides an analysis of the experiment result together with an answer to RQ4. The result shows a clear difference in the mindset needed from the data modeler when focusing on a relational DB versus a graph DB. According to Fernigrini [36], the data structure is essential in the relational DB design. When modeling a NoSQL DB, **the type of queries to be executed on the data is the focal point**. Robinson et al. agree that a graph data model not only shows how we consider things to be related but also clearly communicates the kinds of questions that are important in the modeled domain [5].

The input for the data modeling was mainly an Excel file consisting of a table of 9210 rows and 38 columns. To understand the data, an almost 10-hour long data analysis was performed. The strategy set up for the analysis was later in the modeling phase recognized to be a heavy focus on the needs for the relational database. **The data relationships and groups identified through the data analysis focused on data structures and not the questions that are important in the DBOP domain.**

Figure 118 presents the resulting relational data model. It is in 3NF, and it manages the current DBOP data scope. It is recognized that if the model needs to be kept in 3NF, a change request would require data analysis to be performed again. When looking at the relational data model, a set of connected tables can be identified. The visibility of the process flow is missing, and no comments were received when showing it to the business stakeholder. **The silence is interpreted to mean that the model is not intuitive.**

Neither did the business stakeholder comment on the first graph data model, presented in Figure 121. This also indicates that if business needs were changed, the business stakeholders could not explain how this change affects the data model. The graph data modeling expert disliked this result and gave the impression that **the art of graph data modeling is not only to translate tables from a relational data model into nodes and attributes into properties. The graph data model should be intuitive and flexible to withstand changes in the business.**

It was recognized to be a challenge to drop the detailed data focus and create a graph data model that is simple and intuitive for the business stakeholder. There were no questions to ask about the data and no discussions with the business stakeholders. It was recognized that a 30-minute-long session with business stakeholders describing the DBOP and listing the questions they are interested in would have produced a better graph data model. After several modifications, a graph data model was formed, which was aligned with how the business stakeholder and the graph data modeling expert understands the DBOP, presented in Figure 129.

**The key finding is that a graph data model is not a translation of relational model tables and attributes into graph nodes and properties. Discussions with business stakeholders produce an intuitive and dynamic graph data**

**model. Therefore, the data modelers need to understand the importance of conversations and drop the attitude of managing alone.**

The graph data model was implemented in Neo4j to answer RQ3. This is recognized to be an easy step thanks to the arrows.app that provided a Cypher statement to be run in Neo4j. The decision to use a modeling tool over the possibility of creating the graph data model directly in Neo4j did not create additional work. Importing the data to Neo4j is outside the scope of this study. Importing the data and performing queries on the data would indicate how efficient the graph data model is.

Based on this study, a possibility of extending Wärtsilä's success with graph data models and Neo4j GDBMS to the manufacturing process data and later to value creation in internal and partner networks is recognized. However, a clear yes or no answer cannot be given based on this study.

## **6.2. Recommendations for future research**

The logical data models created in the experimental setup would require further testing and discussions with business stakeholders before being used.

There is an indication that the graph data model would be suited for DBOP. Before Wärtsilä decides on a graph DB implementation for the DBOP, importing data to the Neo4j implementation and testing how this implementation performs compared to the current DB implementation is recommended. The performance can be measured by performing queries designed based on the questions business stakeholders define to be necessary.

An interesting future research would be an experiment where the DBOP graph data model created from the data structure perspective, presented in Figure 121, is compared to the graph data model aligned with business stakeholder understanding, presented in Figure 129. The test could be performed by implementing both in Neo4j, importing the DBOP data, and running performance tests on questions that are interesting to the business stakeholders. This would give an interesting result of how the difference in viewpoint and data



understanding affects the quality of the data model and, hence, the DB's performance.

From the literature review was found indications that the relational data model's response to change is weak and requires expensive configurations to reflect changes in business needs [4] [5]. In contrast, the graph data model is described to be dynamic. It is easy to add new data elements when adapting to new business requirements in the graph data model [13]. Despite the dynamic capability being highlighted as a benefit of the graph data model, we note that research comparing the relational domain to the graph domain mainly focuses on DB query execution times and handling relationships between data elements. The literature review did not find any experiments investigating the difference in effort and time needed to implement and modify a relational data model versus a graph data model.

We designed an experiment to get statistical fact that the graph data model is more dynamic than the relational model. With limit the of only one subject participating in our experiment, we cannot say that our result is statistically relevant. We however, recognized the experiment design and analysis planned to be valid. To get a statistically relevant result we recommend the experiment to be carried out with ten to twenty subjects.

## SWEDISH SUMMARY

### En fallstudie om en övergång från en relationsdatamodell till en grafdatamodell i ett industriellt sammanhang

Den populära relationsdatamodellen är i det här arbetet utmanad av grafdatamodellen. Relationsdatamodellen är en beskrivning av datastrukturen för en relationsdatabas och grafdatamodellen är en beskrivning av datastrukturen för en grafdatabas. Arbetet tar upp grundbegrepp i datamodelleringsprocessen samt beskriver skillnader mellan relations- och grafdatamodellen. Huvudfokusen i litteraturgranskningen är att få en förståelse av när och hur det lönar sig för en firma att ta steget från relations- till grafdatabas. Wärtsilä, som är uppdragsgivare till arbetet är speciellt intresserat av den dynamiska egenskapen av grafdatamodellen. I litteraturen hittas inget bevis på att grafdatamodellen är mer dynamisk än relationsdatamodellen. Det här arbetet innehåller därför ett experiment där den dynamiska egenskapen av en relations- och grafdatamodell mäts. Eftersom endast en person deltog i experimentet kan resultatet dock inte tolkas som statistiskt relevant.

Forskningsfrågorna i fokus i detta arbete är:

- RQ1.** Hur skiljer sig datamodellering för en grafdatabasimplementering jämfört med en relationsdatabasimplementering?
- RQ2.** Kan man genom ett experiment bevisa att grafdatamodellen är mer dynamisk än relationsdatamodellen?
- RQ3.** Hur kan en motors tillverkningsprocess modelleras i Neo4j?
- RQ4.** Vilka kunskaper och kompetenser behövs för grafdatamodellering och vidare implementation i Neo4j?

En grundförståelse för datamodellering och databaser, samt får förståelse när och hur en firma bör byta från en relationsdatabas till en grafdatabas bildas genom att studera virtuella böcker, fallstudier, forskningsresultat och rapporter. Källmaterialet som används är till största del mellan noll till fem år gammalt och hittas genom Google, Google Scholar och Åbo Akademis virtuella biblioteks-databas.

Relationsdatamodellen som används för att beskriva strukturen i en relationsdatabas beskriver både data och kopplingar mellan data i tabellformat.

Grafdatamodellen beskriver datastrukturen som en graf. Denna fundamentala skillnad gör att det är lättare att göra ändringar i grafdatamodellen än i relationsdatamodellen.

Datamodelleringsprocessen för en relations- samt grafdatamodell utgår från att förstå problemet som ska modelleras. Detta arbete sker i samförståelse med affärsintressenter. Enligt Fernigrini [36] är det viktigt att få en förståelse för datastrukturen om man modellerar för en relationsdatabas. Däremot är det viktigt att skapa en förståelse för vilken typ av frågor man är intresserat av att få svar på utifrån data om man designar en grafdatabas [36]. Robinson med flera anser också att grafdatamodellen beskriver hur saker är relaterade och vilka de essentiella frågorna är [5]. Då problemet är förstått och dokumenterat i textformat övergår man till det konceptuella modelleringsskedet.

Tekniska eller systemdetaljer ingår inte i denna datamodell. En ofta använd teknik i relations data modelleringen är att skapa en ER-modell. En ER-modell kan antingen uttryckas som graf- eller textformat. I grafmodellering skapar man också en graf som kan vara så enkel som en skiss på en whiteboardtavla framtagen i ett möte med affärsintressenter. Viktigt i detta skede är att förstå att den konceptuella modellen beskriver problemet från affärsintressenternas synvinkel [7]. Databasutvecklare ska därför vara försiktiga med att inte redan i detta skede lösa problemet och tänka på den verkliga databasimplementeringen [7].

Efter den konceptuella datamodelleringen följer det logiska datamodelleringsskedet. I detta skede definieras hur databasen ska implementeras. Man tar fortfarande inte in detaljer från specifika databassystemleverantörer. För en relationsdatabas översätts ER-modellen till en relationsdatamodell, vilket innebär att en grafrepresentation översätts till tabellformat. Datatabellerna normaliseras ofta till tredje grad. För en grafdatabas förblir modellen i grafformat. Skillnader man kan se då man övergår från konceptuell till logisk datamodell för en grafdatamodell är att sådant som var en entitet blir en egenskap för en nod i den raffinerade modellen [7].

I grafdatabasdesign är den logiska datamodellen den sista modellen som skapas före databasimplementeringen. För en relationsdatabas skapas en fysisk datamodell som går in på detaljer och krav från en specifik databasleverantör.

Datastrukturen kan även modifieras och det som man normaliserat i det föregående skedet kan de-normaliseras för bättre frågeprestanda.

Relationsdatabasen är trots sin robusta tabellstruktur fortfarande populär och passar utmärkt för dataaggregation. Relationsdatabasen med hög dataintegritet och konsistens används ofta i användarfall som kräver hög garanti för datatransaktioner. Ett exempel är banktransaktioner. Relationsdatabasen har dock sina nackdelar. En som lyfts fram i litteraturen är dess **höga underhållskostnad**. Den höga underhållskostnaden är direkt beroende av databasens tabellstruktur med fördefinierade kolumner och krav på att varje rad i tabellen ska vara unik. Dessutom är **mappningen av data i tabellformat inte hur data existerar i verkligheten**. I verkligheten existerar data som objekt och relationer mellan dessa objekt.

En relationsdatabas **sparar inte relationer**, utan dessa kalkyleras vid behov med hjälp av kopplingsförfrågningar mellan tabeller. Dessa kalkyler är **kostsamma**, eftersom relationsmodellen först tar fram en mängd möjliga svar och från dessa sedan filtrerar ut det rätta svaret. Med dagens extensiva datakopplingar kan relationsdatabasen orsaka situationer där ett företag går miste om värdefull förståelse av data och dess kopplingar på grund av att en relationsdatabas inte klarar av att leverera svar på frågor som går djupare än fyra hierarkiska nivåer.

Grafdatabasen kan i motsats till relationsdatabasen prestera väl i situationer där antal attribut, data samt kopplingar mellan data är stora, det finns höga krav på affärsflexibilitet och hur snabbt man får tillgång till data. I motsats till relationsdatabasen är grafdatabasen direkt framtagen för att spara data och kopplingar mellan data. I en grafdatabas **är kopplingen mellan data lika viktig som dataelementet, om inte ännu viktigare**. Grafdatabasens struktur som utgörs av noder samt kopplingar mellan dessa noder gör **det lätt att utöka strukturen för att svara på ändrade affärsbehov**. Exempel på världsledande företag som skapat sitt värde utgående från datakopplingar är *Facebook, Google, LinkedIn* samt *Paypal*. Alla dessa är tidiga adoptanter av grafdatabasen.

Grafdatabasen har visat sig förträfflig i situationer som:

- Bedrägeriupptäckt i realtid
- Realtidsrekommendationer till användare
- Masterdata
- Nätverks- och informationsteknikverksamhet
- Identitets- och åtkomsthantering

- Uppfyllande av regelverk
- Analyser
- Digital tillgångshantering
- Kontextmedvetna tjänster
- Semantisk sökning
- Situationsmedvetenhet

Trots att man på internet kan läsa om många förträffliga implementeringar av grafdatabasen i olika typer av företag, ska man inte bli förbryllad och välja databastyp utgående från något man läst om. Viktigt är i stället att förstå sitt eget problem och använda sig av ett analytiskt förfarande då typ av databas väljs. En tydlig indikering på bättre svarsprestanda, flexibilitet och smidighet behövs för att överväga ett byte från en väletablerad och lättförstådd databas till en grafdatabas.

I ett analytiskt förfarande för att välja typ av databas utgår man från vilket typ av problem man försöker lösa. Man bildar en förståelse av vilken typ av data man kommer lagra samt hur data ska hämtas. På en generell nivå anses alla problem passa i någon av dessa kategorier:

1. Urval/sökning
2. Aggregation
3. Relaterade eller rekursiva data
4. Mönstermatchning
5. Centralitet, bildning av kluster och inflytande

Kategori ett och två anses vara bättre ämnade för en relationsdatabas, medan kategori tre till fem är lämpade för en grafdatabas. Om man efter kategoriseringen fortfarande känner osäkerhet kan man använda sig av beslutsträdet i Figure 21.

Uppdragsgivaren, Wärtsilä, var speciellt intresserat av den dynamiska kapabiliteten av grafdatabasen. I litteraturen fanns inget bevis på att grafdatamodellen är mer dynamisk än relationsdatamodellen. För att få ett svar gjordes en fallstudie där den dynamiska egenskapen av en relations- och grafdatamodell mäts. En dynamisk egenskap anses i detta arbete vara detsamma som effektiviteten av att skapa och därefter modifiera datamodellen. I experimentet modelleras en logisk relationsdatamodell samt en grafdatamodell. Data som modelleras är DBOP för en motor som produceras vid Wärtsilä STH i Vaasa, Finland.

Trots utmaningar att bygga en grafmodell som godkändes av en grafmodelleringsexpert på Wärtsilä, visar fallstudien att grafdatamodellen visar indikation på att vara mer dynamisk än relationsdatamodellen. Resultatet kan dock inte tolkas som statistiskt relevant på grund av att endast en person utförde fallstudien. Största tiden av fallstudien gick åt till att analysera och förstå data som skulle modelleras. Utgående från analysen var det med tidigare erfarenheter av en relationsdatabas relativt enkelt att bygga en relationsdatamodell enligt tredje gradens normalisering.

Att bygga grafmodellen utgående från samma dataanalys och översätta tabeller och attribut i relationsdatamodellen till noder och egenskaper i grafdatamodellen visade sig vara ett dåligt val. Att vara för analytisk i grafdatamodellering är en nackdel. I stället för att datamodelleraren ensam analyserar data, bör hen uppsöka de som förstår sig på problemet i fråga och tillsammans med dem ta fram en konceptuell lösning till problemområdet. Endast genom diskussion kan konsten med att skapa en intuitiv, välfungerande och dynamisk grafdatamodell uppnås. Är grafdatamodellen en översättning från relationsdatamodellen saknas intuitivitet och affärsintressenter kommer inte förstå modellen, vilket i sin tur leder till problematik i diskussioner då en eventuell ändring ska överenskommas och implementeras. Vilket verktyg som används i datamodelleringsskedet är också viktigt att tänka på. I det här arbetet användes *arrows.app*. *Arrows.app* genererar ett Cypher-skript som kan köras i Neo4j vid en databasimplementering. Detta sparar tid och dubbelarbete undviks.

Genom fallstudien bildades även förståelsen av att ett eventuellt modifieringsbehov av datamodellen skulle vara mer tidskrävande för relationsdatamodellen än för grafdatamodellen. Detta baserar sig delvis på att dataanalysen för att skapa relationsmodellen var sex gånger längre jämfört med grafdatamodellen. En annan orsak är att affärsintressenterna hade lättare att kommentera grafdatamodellen jämfört med relationsdatamodellen. Flera kommentarer visar att grafdatamodellen är lättare att förstå och det är därmed lättare att diskutera och uppnå konsensus om vilka ändringar som behövs när affärskraven ändras.

Genom en litteraturgranskning samt en fallstudiehar i detta arbete uppnåtts en förståelse av datamodelleringsprocessen för en relationsdatamodell samt grafdatamodell. Relationsmodellen med sin robusta tabellstruktur kan anses vara mindre dynamisk än grafdatamodellen i en värld med ständiga förändringar.

Dock ska beslutet av att övergå till en grafdatamodell inte fattas okritiskt eller utgående från trender. I stället ska en noggrann analys på basen av det egna problemområdet göras. Problemområdet i fokus i detta arbete var en motors tillverkningsprocess och en eventuell utvidgning till partnernätverk. Från dataanalysen som visar att DBOP har elva hierarkiska nivåer kan man anta att grafdatabasen skulle uppvisa sin fördel i frågeprestanda jämfört med relationsdatabasen. Detta understöds av resultatet av litteraturgranskningen som visar att en relationsdatabas har svårigheter att leverera resultat i frågor som sträcker sig djupare än fyra nivåer. För Wärtsilä rekommenderas dock fortsatta studier i problemområdet innan ett slutgiltigt beslut görs.

## REFERENCES

- [1] Wärtsilä, "New Open Smart Manufacturing Ecosystem aims at transforming manufacturing collaboration," Wärtsilä, 26 1 2022. [Online]. Available: <https://www.smarttechnologyhub.com/new-open-smart-manufacturing-ecosystem-aims-at-transforming-manufacturing-collaboration/>. [Accessed 16 5 2022].
- [2] "Cambridge Dictionary," [Online]. Available: <https://dictionary.cambridge.org/dictionary/english/dynamic>. [Accessed 24 5 2022].
- [3] R. Kumar, "Flexible Data Modeling is Key for Product Information Management Strategy," 4 2 2020. [Online]. Available: [https://pimcore.com/en/resources/blog/flexible-data-modeling-is-key-for-product-information-management-strategy\\_a44814](https://pimcore.com/en/resources/blog/flexible-data-modeling-is-key-for-product-information-management-strategy_a44814). [Accessed 23 5 2022].
- [4] M. Kunkel, "The Rise of the Flexible Data Model," 6 5 2019. [Online]. Available: <https://www.logicgate.com/blog/the-rise-of-the-flexible-data-model/>. [Accessed 23 5 2022].
- [5] I. Robinson, J. Webber and E. Eifrem, Graph Database New Opportunities for Connected Data, Sebastopol, USA: O'Reilly Media Inc, 2015.
- [6] A. e. a. Silberschatz, Database System Concepts, McGraw-Hill Companies, 2010.
- [7] D. Bechberger and J. Perryman, Graph Databases in Action, Shelter Island: Manning, 2020.
- [8] H. Piili, "Comparing EDW and PLMGraph (Wärtsilä internal documentation)," 13 10 2021. [Online]. Available: <https://confluence.devops.wartsila.com/display/PLMB/Comparing+EDW+and+PLMGraph>. [Accessed 7 6 2022].
- [9] M. Hunger, R. Boyd and W. Lyon, The Definitive Guide to Graph Databases, Neo Technology, 2021.
- [10] H. Piili, "Graph Database Selection (Wärtsilä internal documentation)," 14 5 2022. [Online]. Available:



- <https://confluence.devops.wartsila.com/display/PLMB/Graph+Database+Selection>.  
[Accessed 7 6 2022].
- [11] "DB-Engines Ranking," DB-Engines, 6 2022. [Online]. Available: <https://db-engines.com/en/ranking>. [Accessed 8 6 2022].
- [12] "DB-Engines Ranking - Trend Popularity," DB-Engines Ranking, 6 2022. [Online]. Available: [https://db-engines.com/en/ranking\\_trend](https://db-engines.com/en/ranking_trend). [Accessed 8 6 2022].
- [13] J. Webber and I. Robinson, "The Top 5 Use Cases of Graph Databases," 8 5 2017. [Online]. Available: [https://go.neo4j.com/rs/710-RRC-335/images/Neo4j\\_Top5\\_UseCases\\_Graph%20Databases.pdf?\\_gl=1\\*1n48pdm\\*\\_ga\\*NDM2NDcxODk3LjE2NDc1OTY2NDY.\\*\\_ga\\_DL38Q8KGQC\\*MTY1MjMzOTM0Ni43LjEuMTY1MjMzOTQ4MC4w&\\_ga=2.100916361.965507018.1652339346-436471897.1647596646&\\_gac=1.586691](https://go.neo4j.com/rs/710-RRC-335/images/Neo4j_Top5_UseCases_Graph%20Databases.pdf?_gl=1*1n48pdm*_ga*NDM2NDcxODk3LjE2NDc1OTY2NDY.*_ga_DL38Q8KGQC*MTY1MjMzOTM0Ni43LjEuMTY1MjMzOTQ4MC4w&_ga=2.100916361.965507018.1652339346-436471897.1647596646&_gac=1.586691). [Accessed 23 5 2022].
- [14] "datum/data," The Mayfield Handbook of Technical & Scientific Writing, [Online]. Available: <https://web.mit.edu/course/21/21.guide/data.htm#:~:text=Datum%20is%20singular%2C%20meaning%20%22one,used%20as%20a%20singular%20noun..> [Accessed 23 6 2022].
- [15] "Database," javatpoint, [Online]. Available: <https://www.javatpoint.com/what-is-database>. [Accessed 10 6 2022].
- [16] "What is a Database Model," Lucidchart, [Online]. Available: <https://www.lucidchart.com/pages/database-diagram/database-models>. [Accessed 9 6 2022].
- [17] K. Sahatqija, J. Ajdari, X. Zenuni, B. Raufi and F. Ismaili, "Comparison between relational and NoSQL databases," *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018.
- [18] J. Bhogal and I. Choksi, "Handling Big Data Using NoSQL," *International Conference on Advanced Information Networking and Applications Workshops*, 2015.
- [19] "Types of Databases," Javatpoint, [Online]. Available: <https://www.javatpoint.com/types-of-databases>. [Accessed 10 6 2022].

- [20] "Neo4j – The Leader in Graph Technology," Neo4j, 2022. [Online]. Available: <https://neo4j.com/company/>. [Accessed 26 2022].
- [21] "The Graph Technology Buyer's Guide - What You Should Know Before Selecting a Graph Technology Solution," 28 10 2021. [Online]. Available: <https://neo4j.com/whitepapers/graph-database-buyers-guide/>. [Accessed 27 5 2022].
- [22] "Neo4j Licensing," Neo4j, 2022. [Online]. Available: <https://neo4j.com/licensing/>. [Accessed 27 6 2022].
- [23] J. Depeau, "Graphs in Automotive and Manufacturing: Unlock New Value from Your Data," neo4j, 27 5 2020. [Online]. Available: <https://neo4j.com/blog/graphs-in-automotive-and-manufacturing/>. [Accessed 31 5 2022].
- [24] A. Vucotic, N. Watt, T. D. F. Avedrabbo and J. Partner, Neo4j in Action, Shelter Island, NY: Manning Publications Co., 2014.
- [25] B. M. Sasaki, "Graph Databases for Beginners: ACID vs. BASE Explained," Neo4j, 13 11 2018. [Online]. Available: <https://neo4j.com/blog/acid-vs-base-consistency-models-explained/?ref=blog>. [Accessed 26 2022].
- [26] "Neo4j Developer Guide - Cypher Query Language," Neo4j, [Online]. Available: <https://neo4j.com/developer/cypher/>. [Accessed 16 2022].
- [27] "The Neo4j Cypher Manual v4.4," Neo4j, 2022. [Online]. Available: <https://neo4j.com/docs/cypher-manual/current/>. [Accessed 26 2022].
- [28] "Neo4j Cypher Manual - Cypher styleguide," Neo4j, [Online]. Available: <https://neo4j.com/docs/cypher-manual/current/styleguide/>. [Accessed 26 2022].
- [29] "Neo4j Cypher Manual - Naming rules and recommendations," Neo4j, [Online]. Available: <https://neo4j.com/docs/cypher-manual/current/syntax/naming/>. [Accessed 26 2022].
- [30] "Neo4j Cypher Manual - Query tuning," Neo4j, [Online]. Available: <https://neo4j.com/docs/cypher-manual/current/query-tuning/>. [Accessed 26 2022].

- [31] M. West, *Developing high quality data models*, Burlington, USA: Elsevier Inc., 2011.
- [32] G. F. Hurlburt, G. K. Thiruvathukal and M. R. Lee, "The Graph Database Jack of All Trades or Just Not SQL?," *IT Pro / IEEE Computer Society*, no. November/December 2017, 2017.
- [33] D. Taylor, "Data Modelling: Conceptual, Logical, Physical Data Model Types," 23 4 2022. [Online]. Available: <https://www.guru99.com/data-modelling-conceptual-logical.html>. [Accessed 25 5 2022].
- [34] "Ultimate Entity Relationship Diagram Tutorial (ER Diagrams)," Creately, 25 4 2022. [Online]. Available: <https://creately.com/blog/diagrams/er-diagrams-tutorial/>. [Accessed 10 6 2022].
- [35] N. Roy-Hubara, L. Rokach, B. Shapira and P. Shoval, "Modeling Graph Database Schema," *IT Professional*, vol. vol. 19, no. no. 6, pp. pp. 34-43, 2017.
- [36] K. Kaur and R. Rani, "Modeling and Querying Data in NoSQL Databases," *IEEE International Conference on Big Data*, 2013.
- [37] "Neo4j Developer Guide - Graph Data Modeling," Neo4j, [Online]. Available: <https://neo4j.com/developer/data-modeling/>. [Accessed 16 6 2022].
- [38] L. Fernigrini, "What Are Conceptual, Logical, and Physical Data Models?," Vertabelo, 9 2 2021. [Online]. Available: <https://vertabelo.com/blog/conceptual-logical-physical-data-model/>. [Accessed 11 6 2022].
- [39] J. Cao, D. F. Bucher, D. M. Hall and M. Eggers, "A graph-based approach for module library development in industrialized construction," *Elsevier*, 15 3 2022.
- [40] J. Saarela, "Graph Database Use Cases (10 examples)," 7 2 2020. [Online]. Available: <https://www.profium.com/en/blog/graph-database-use-cases/>. [Accessed 24 5 2022].
- [41] ActiveWizard, "Graph Databases Use Cases," [Online]. Available: <https://activewizards.com/blog/graph-databases-use-cases/>. [Accessed 24 5 2022].

- [42] "ERDPlus," [Online]. Available: <https://erdplus.com/>. [Accessed 10 8 2022].
- [43] "Arrows.app," [Online]. Available: <https://arrows.app/>. [Accessed 10 8 2022].
- [44] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, R. B. and A. Wesslén, Experimentation in Software Engineering, Springer, 2012.
- [45] "Pandas," [Online]. Available: <https://pandas.pydata.org/>. [Accessed 11 9 2022].
- [46] "NumPy Introduction," w3schools, [Online]. Available: [https://www.w3schools.com/python/numpy/numpy\\_intro.asp#:~:text=NumPy%20aims%20to%20provide%20an,and%20resources%20are%20very%20important..](https://www.w3schools.com/python/numpy/numpy_intro.asp#:~:text=NumPy%20aims%20to%20provide%20an,and%20resources%20are%20very%20important..) [Accessed 11 9 2022].
- [47] "First Normal Form (1NF)," Geeks for Geeks, 15 7 2022. [Online]. Available: <https://www.geeksforgeeks.org/first-normal-form-1nf/?ref=lbp>. [Accessed 1 10 2022].
- [48] "Second Normal Form (2NF)," Geeks for Geeks, 25 11 2019. [Online]. Available: <https://www.geeksforgeeks.org/second-normal-form-2nf/?ref=lbp>. [Accessed 1 10 2022].
- [49] "Third Normal Form (3NF)," Geeks for Geeks, 31 7 2019. [Online]. Available: <https://www.geeksforgeeks.org/third-normal-form-3nf/?ref=lbp>. [Accessed 1 10 2022].
- [50] "Intro to Graph Databases Episode #4 - (RDBMS+SQL) to (Graphs+Cypher)," Neo4j, 22 3 2016. [Online]. Available: <https://www.youtube.com/watch?v=NO3C-CWykkY&t=294s>. [Accessed 2022 8 1].
- [51] "R-Studio," [Online]. Available: <https://www.rstudio.com/>. [Accessed 10 14 2022].