FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Comparison of storing data flows in graph and relational database
Student:	Valeriy Lyalin
Supervisor:	Ing Michal Peroutka
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

The aim of the work is to compare the performance of the classic relation and graph database on the specified use cases and on data given by the client. The motivation is the question whether it would be worthy to transfer or replicate part of the data into the graph database in the client's product.

Follow these steps:
1. Get to know MMP product and its current way of storing data for data flow analysis.
2. Learn typical use cases for data flow analysis.
3. Search for potential graph database engines that could be used.
4. In agreement with the supervisor, select one database engine, implement the relevant part of the database you will get from the supervisor.
5. Describe and implement use cases in both databases as a benchmark.
6. Compare the response in both systems, discuss the results.
Make recommendations whether to go to the graph repository with part of the database.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 28, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Comparison of Storing Data Flows in Graph and Relational Database

Valeriy Lyalin

Department of Software Engineering
Supervisor: Ing. Michal Peroutka

May 26, 2020

Acknowledgements

I would like to thank everybody who directly or indirectly contributed to the bachelor thesis. My special thanks go to Ing. Michal Peroutka, the supervisor of the thesis, who mentally and physically supported during the whole process of working on the thesis, and who shared his rich experience and always was there when I needed.

Declaration

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

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 26, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Lyalin, Valeriy. *Comparison of Storing Data Flows in Graph and Relational Database*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Bakalářská práce se zabývá porovnáním ukládání a dotazování datových toků v grafovém a relačním systému správ databází na základě určitých případů užití. První část práce nastiňuje zvlaštností systémů pro správu grafových databází a architekturu podnikových datových skladů. Poté na základě požadavků je provedeno srovnání existujících grafových databází, na konci kterého se vybere ArangoDB pro nasledující implementaci a porovnávání. Dále se popisuje návrh optimálního mapovacího modelu v grafové databázi na základě určitých požadavků a na základě existujícího relačního modelu. Dále se nastiňuje implementace operací nad datovými toky na základě případů užití. Pak se provádí řadu testů, díky kterým se učiní závěr, že pro konkretné případy užití se vyplatí používat systém pro správu grafových databází.

Klíčová slova grafová databáze, porovnávání databází, datový model, ArangoDB, NoSQL, .NET Core

Abstract

The bachelor thesis compares storing and querying data flows in a graph and a relational database management system based on specific use cases. The

first part of the thesis outlines the Graph Database Management Systems and Enterprise Data Warehouse Architecture. Then, concerning the requirements, conducts a comparison of existing graph databases, based on which chooses ArangoDB for the following implementation and comparison. Then describes the designing optimal graph mapping model based on requirements and an existing relational model. Then outlines an implementation of the data flow operations based on use cases and conducts a series of tests to conclude, if it is worthy to represent the mapping model in a graph database management system. Finally finds that on the given use cases, it is worth using a graph database management system.

Keywords graph database, database comparison, data model, ArangoDB, NoSQL, .NET Core

Contents

List of Figures

List of Tables

Introduction

The limitations of traditional relational databases to cover the requirements of application domains have lead to the development of new alternative technologies. These technologies are called NoSQL, which stands for "not only SQL". All the NoSQL technologies could be categorized as key-value stores, document stores, column-oriented databases, and graph databases. Each type of NoSQL database has its own strengths and weaknesses. Key-value stores are more flexible and could have better performance in read and write operations. Document stores are highly transient and have more flexible schema as well. Column-oriented databases are more efficient in hard-disk access for a given workload. Graph databases are more efficient in querying highly connected data and provide Graph Processing Engine (i.e. "Index Free Adjacency").

NoSQL technologies are not new. They have existed from the late 1960s, but relational databases were the only standard for all types of applications simply because the relation models cover all the requirements of application domains. It is still the standard for almost all domains, but the recent growth in social network analysis and overall data analyses have lead developers to extend their horizons and use NoSQL technologies. Some companies stated to enlarge (or even replace) their existing relational databases by NoSQL technologies. Among these companies belongs a company that owns the Metadata Management Platform (MMP).

The MMP is a model-oriented project, which among other things provides a Data Dictionary tool. The customer's data warehouse metadata, which is shown in the tool, usually comes from different resources, then stores at the stage area and goes through a series of transformations in a Data Warehouse, Data Marts, and eventually is shown to users. All of the data flow transformations are shown in the Data Dictionary tool. For each object, the application shows source and destination objects. The MMP project stores all the data flow transformations in a relational database. Relational Database Management Systems (RDBMS) can efficiently handle the task, but the project is interested in extending an existing Data Dictionary tool by adding additional

data linage functionality. Relational databases are not so effective in the area. A typical use case that metadata-oriented project is intended to add is finding the shortest path between two objects in transformations.

The aim of this thesis

The bachelor thesis aims to answer a question if it is worth implementing a relative part of a mapping model using an existing relational mapping model or current relational database is capable of effectively performing the data flow operations. The aim is considered to be achieved by the following smaller steps.

1. Get to know MMP product and its current way of storing data for data flow analysis.
2. Learn typical use cases for data flow analysis.
3. Search for potential graph database engines that could be used.
4. In agreement with the supervisor, select one database engine and implement the relevant part in the database.
5. Describe and implement use cases in both databases as a benchmark.
6. Compare the response in both systems, discuss the results.

Chapter 1

Graph database management system

A Graph Database Management System (GDBMS) is a database management system with Create, Read, Update, Delete methods that expose a graph data model. Graph systems are generally optimized for performance, and engineered with integrity and operational availability in mind [1]. The structure of graph databases is generally contained with nodes and edges. Nodes represent an object and edges represent the connection (usually directed) between two objects. The fundamental concept of the system is the graph (or relationships). Graph databases keep relationships between nodes as a priority. When relational systems using joins to connect tables, graph databases usually use pointers or indices to connect documents. According to the people from Neo4j: "*Accessing nodes and relationships in a native graph database is efficient, constant-time operation and allows you to quickly traverse millions of connections per second per core.*" [2]

1.1 The storage types of graph databases

Graph databases differ in underlying storage. There are two main storage types: native and non-native. Native graph storage is storage specifically designed for graph database requirements. Graph storage is considered to be non-native when it comes from relational, wide-column or other NoSQL database. Databases with native storage offer index-free adjacency. This means that each node has a collection of pointers of its edges. In contrast, the idea of index adjacency is that each node has an index to an item in a big hash table. The table is usually a hash table with double-linked-lists in each cell. This linked-lists contains edges.

The graph databases with index-free adjacency present this feature as a huge benefit over graph databases that have a non-native data storage [3]. It

may be true in cases where a database located only on one server (i.e., runs in a standalone mode) and nodes have a relatively small amount of edge pointers. To be able to traverse edges in both directions, nodes have to store edges on both directions. To delete a bidirectional edge in index-free adjacency, the edge has to be deleted in both nodes. But this is not the only problem. The main problem is that index-free adjacency is worthless when it comes to distributed world [4], simply because pointers works only on one machine.

1.2 Graph models

Graph models can be categorized as label-property graphs (LPG), Resource Description Frameworks (RDF), and Hypergraphs.

A label property model is realized by vertexes, edges, properties, and labels. Both vertexes and connections store properties, which are represented by key-value pairs. Labels are used when defining constraints, adding indexes for properties and grouping them. An object can have multiple labels. Edges and nodes are stored in edge and node collections respectively. Each collection is usually a set of documents in JavaScript Object Notation (JSON) format with unique property identifiers. Each document in edge collection additionally has source and destination vertexes.

Resource description framework also knows as Triplestores, is a triple of a subject, a predicate and an object. Subjects and objects are vertexes and predicates are edges. RDF has two types of vertexes: resources and attribute values, and one type of edges: relationships, which connect resources. Resources and relationships have a unique uniform identifier (URI). An attribute value contains literal values. Nodes or edges have no internal structure. RDF databases, such as GraphDB [5], AlegroGraph [6] claim that this technology over a relational system gives the benefit of better discoverability of content assets, rich semantic context, easier knowledge exploration, and navigation [7]. But the structure of RDF has some negative aspects. The first aspect is that edges do not have attributes. To add properties to the edge in the RDF graph, we have to add an intermediate object with attributes and connect all of these three objects by edges. Another aspect is that all of the edges are unique. Consider modelling the fact that Dan has liked Alice three times. In a label-property graph, it can be simply represented as three edges from Dan to Alice with a label “likes”. In a Triplestores, it must be stored in a different scheme. That scheme is outlined in figure 1.1.

Another type of database is a hypergraph. The key feature of this database over a property graph database are hyperedges. A hyperedge is an edge that connects any number of edges. One of the representants of this technology is HypergraphDB [8], which describe itself as a general-purpose, open-source data storage mechanism based on a robust knowledge management formalism known as directed hypergraphs. According to the database vendor, the hyper-

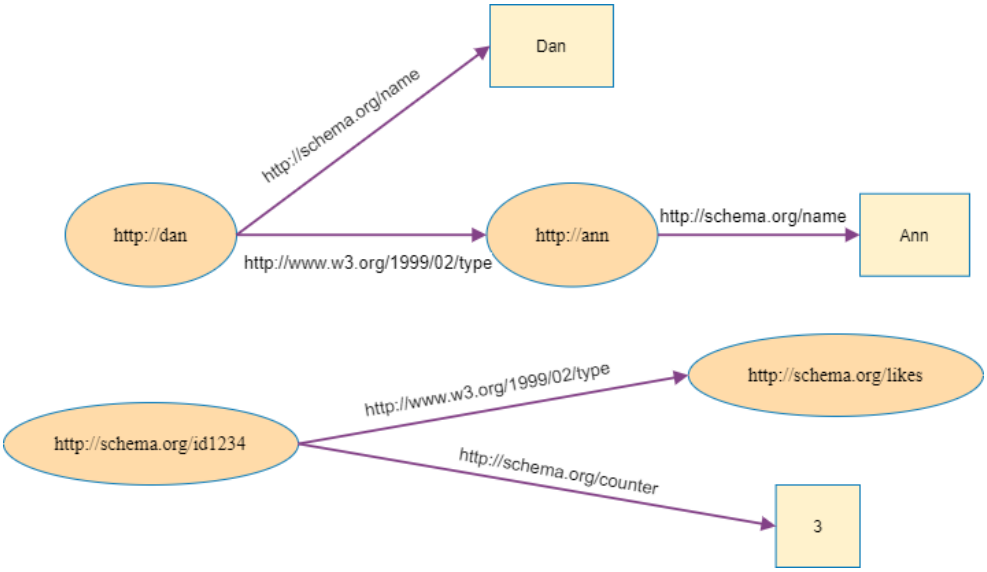

Figure 1.1: An RDF model of the fact that Dan has liked Ann three times

graph model is designed mostly for knowledge management, AI and semantic web projects.

1.3 Multi-model graph databases

A multi-model database is designed to support multiple data models against a single, integrated backend. The type of database can store, index and query data in more than one model. The most significant advantages of such a system are that it takes useful features from multiple database systems, and it has one API for querying. ArangoDB and OrientDB are hybrid databases that combine document, graph and key-value models. A document store is suitable for querying some of its fields. A graph store is useful for performing graph operations. The mentioned databases combine these two benefits. For example, product registration and customer information in a grocery store can be modelled in a document store for better querying of a field that describes product or customer. However, finding correlations between customers buying certain products would be better performed on adequately modelled graph system. Usually, customers want both, so a multi-model data store is a solution to the problem.

The other side of the approach is that due to different model combination, it is not suitable for all use cases of each model. For example, ArangoDB can be used as a key-value store. However, due to additional overheads (such as adding identifiers to every document), it is not recommended to use ArangoDB for use cases which require hyper-scale [9].

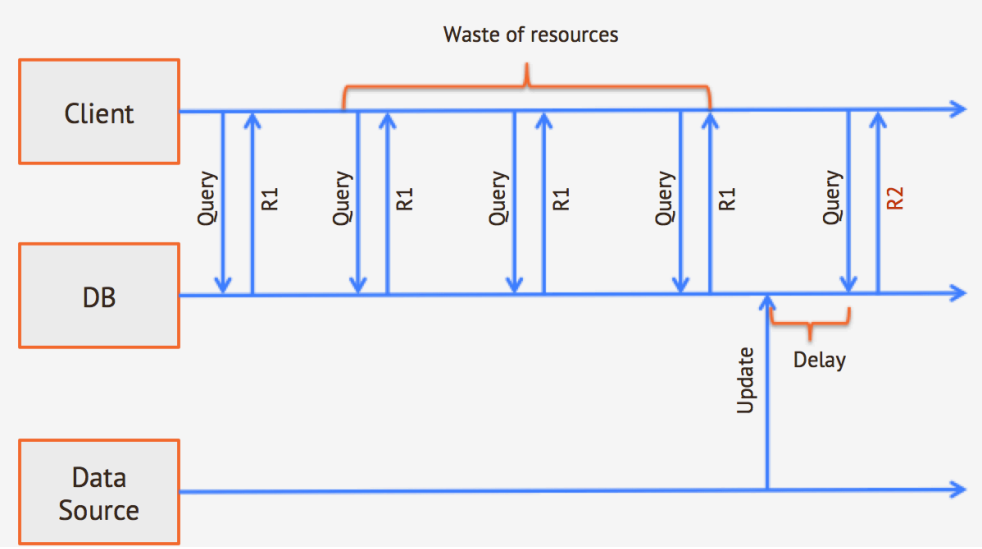

Figure 1.2: A comparison of a regular query and a live query. Source: [10]

1.4 Live query

Live query is a subscription on changes on a particular object in a database. This future supports only a small part of graph databases such as OrientDB. The feature is useful when it comes to realtime, reactive applications. Mainly when a backend and a database server are located on separate servers and communicate via the internet, which is quite a popular scenario. Instead of continually querying a database for updates, pulling and comparing the data, an application can subscribe on updates on a particular object in a database (as shown on the figure 1.2).

1.5 Querying

In contrast with a relational world, where almost every relational database supports one querying language, which is called SQL, as of 2020, there is no such language in NoSQL systems. This can be solved by releasing GQL language. However, the language is still in progress. There were some attempts of creating a general-purpose language for all the graph databases, or at least for label-property graph databases. However, it all ended up with the fact that almost every database has its querying language. Although there are some languages, which can be considered as relatively popular among others. These languages are Cypher, SPARQL, Gremlin and Graph Query Language(GQL).

Cypher is a query language of the most popular graph database – Neo4j [11]. The language is designed for a Property Graph Model, specifically for Neo4j. Syntax of the Cypher is unique and does not SQL-like language.

SPARQL is a querying language for RDF. SPARQL supports Amazon Neptune, AllegroGraph, Apache Marmotta and others. SPARQL consists of triple patterns, conjunctions, disjunctions, and optional patterns.

Gremlin a graph query language of Apache TinkerPop. This language supports OrientDb, TitanDb, Apache Spark and others. Queries in the language could be written in an imperative or declarative way. Gremlin has been made with the idea to be a universal language for property-label databases. The query transforms to a set of traversal strategies, which do their best to determine the most optimal execution plan based on an understanding of graph data access costs as well as the underlying data systems' unique capabilities [12].

GQL is a new language for property graph databases. This language is the first new ISO database language since SQL. As of 2020, this language is still in progress. The language will extend SQL by combining some ideas from open Cypher, Oracle PGQL and other query languages. GQL is intended to be a declarative database language, like SQL. This means that the GQL would express the logic of a computation without describing its control flow.

1.6 ACID and BASE

ACID is an acronym, which stands for atomicity, consistency, isolation and durability. These properties provide a mechanism to ensure correctness and consistency of a database. Atomicity means that a transaction is completed or is aborted as a whole. Consistency ensures that a database transfers from one consistent state to another. Isolation or independence ensures that result of concurrent execution of a transaction is the same as sequential execution. Durability guarantees that once a transaction is committed, it will remain committed even in case of a system failure.

The BASE is an acronym, which breaks down as basic availability, soft-state and eventual consistency. Basic availability means that a database appears to work most of the time. Soft state implies that a database does not guarantee to be consistent. Eventual consistency ensures consistency at some later point. This model values availability over consistency. The BASE consistency model is used by column, key-value and document stores.

When it comes to a distributed world, database systems have to face with CAP theorem. CAP theorem is a concept that a distributed database can have at most two of the following properties: consistency, availability and partition tolerance. A system is partition tolerant when it can sustain any number of network failure between nodes that do not fail an entire network. Availability means an ability to access a cluster even if a node in the cluster is down. Consistency means that data is the same across a cluster. Partition tolerance cannot be left, because network faults happen quite often. So systems are

choosing between consistency and availability. If a database is intended to support ACID properties, it has to give up with availability.

1.7 Graph searching algorithms

It is always better to perform complex graph operations on a database server since transferring data over network significantly slows down the calculation. In a relational world, these partially solved by using procedures. But procedures do not return data. There are some workarounds how to get data from a procedure execution (for example storing it in a table and then retrieve data). But this is quite complicated. Developers of most graph databases thought about the fact that performing graph searching algorithms on a graph database is a common scenario, so most of the graph databases support the most used graph algorithms. These algorithms are finding the shortest path between two nodes (OrientDb, ArangoDb, Neo4j, etc.), getting distances between two nodes (ArangoDb, OrientDb, Neo4j), node similarity calculation (Neo4j), PageRank algorithm (Neo4j) and others.

1.8 Graph databases and scaling

Scaling is a necessity for larger application domains. Most of the graph databases offer vertical and horizontal scaling. In a database world, vertical scaling means improving an existing server by adding more or improving CPU, RAM and memory. Horizontal scaling is merely adding more machines. The vertical direction is more expensive, and costs grow exponentially, whereas horizontal improvement is cheaper and costs grow linearly. Horizontal scaling though involves more software efforts to take advantage of scale. The process of combining more than one servers or instances connecting a single database is called clustering. Clustering is closely associated with replication, partitioning, load balancing, automation and other things.

1.8.1 Replication

Replication is a strategy of duplicating data across servers (nodes). Replication in ArangoDB can be done using Master-Slave or Active Failover technique.

In a Master-Slave setup, more slaves replicate from a master. The replication can be synchronous or asynchronous. In the case of synchronous replication, when performing update operations, data is locked until all the slaves have updated their data. In this approach, we gaining consistency, but have to trade off availability. If asynchronous mode is turned on, data modification is firstly performed on a master. All the changes are logged in the write-ahead log. Using the log, a replication applier reads data from a master database's

log and applies changes locally. This leads to eventual consistency because there is a replication lag, which can be described as the time between applying modifications on a master and a moment when all the slaves are synchronized (i.e. have the same data). Every slave has to fetch the write operation's data from the master's log, then parse and apply it locally. The duration of the lag depends on network capabilities, amount of data and frequency in which slaves are checking updates. When availability plays a crucial role and more important than consistency, this can be a solution.

Active Failover technique is a slightly better version of a Master-Slave setup. It consists of a leader, a follower(s) and an agency. The role of a leader and a follower is the same as in Master-Slave setup. An agency determines which server becomes a leader in a failure situation. Moreover, an agency observes and supervises all server processes.

1.8.2 Partitioning

Partitioning is distributing data between nodes. Partitioning can be vertical and horizontal. Vertical partitioning is creating a collection or a table with fewer properties or columns and using additional collections or tables to store the remaining data. Horizontal partitioning involves splitting data using a particular property or column. For example, in reporting data can be partitioned by a creation month.

1.8.3 The ArangoDB cluster structure

In ArangoDB, the structure of a cluster includes agents, coordinators and database servers (shown in figure 1.3).

Agency is the central place to store the configuration in a cluster. Without the Agency none of the other components can operate. It performs leader elections and provides other synchronization services for the whole cluster. All the agency decisions are replicated in a configuration tree. It supports transnational reads and writes, which among other things means that a cluster can contain multiple agencies. Agency is not visible from the outside, but other servers inside a cluster subscribe for all the changes in the configuration tree.

Coordinators are the ones the client(s) can talk to. They know where data is stored and parse queries to find out what data to look for. Based on the query, coordinators either send a query to a particular database server or send it to all servers. Then obtain data from the database server(s) and return the data to the client. Coordinators are stateless, which means they can be easily restarted as needed.

A database server stores data. It can be either a leader or a follower. Data modifications firstly applied to a leader and secondly to followers. It uses an active failover technique, meaning that database servers are subscribed for all

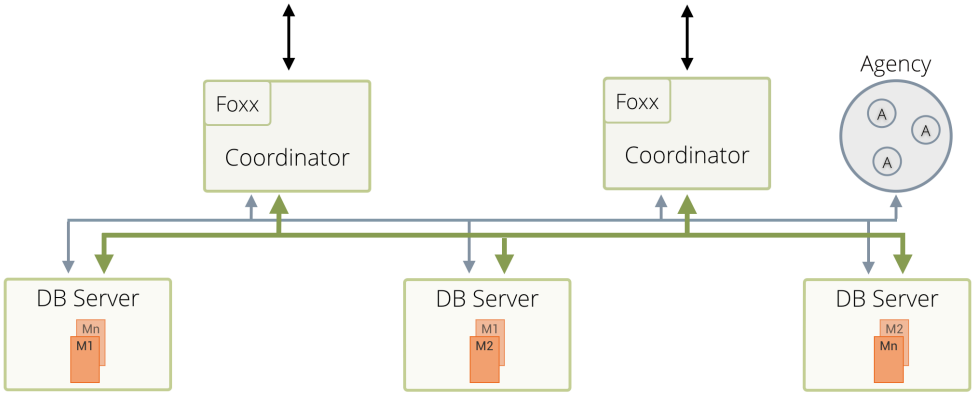

Figure 1.3: Structure of an ArangoDB Cluster. Source: [13]

the changes in the configuration tree. They can execute queries, but database services are not accessible directly, only indirectly through coordinators.

1.8.4 ArangoDB cluster recomendations

Each item in the ArangoDB structure can be multiplied. ArangoDB recommends placing an agency on a separate, less powerful server. Then place coordinators on machines, where database servers run. The configuration reduces latency of transferring data from a database server to a coordinator.

1.8.5 Datacenters

At some point in the growth of the database, there is a need for replicating data across multiple clusters. By doing that we can reduce the possibility of failure, increase availability, etc. In ArangoDB terminology, it is called datacenter to datacenter replication (DC2DC), but it is possible to replicate multiple datacenters. Replication between datacenters is asynchronous. It is not synchronous, because in that case availability decreasing significantly. Replication can be done only in one direction and cannot be synchronous (as of ArangoDB 3.6). Also, only one datacenter can be primary, and others are replicas. All the clients communicate only with a central cluster. Data modifications are made to a primary datacenter, then are made asynchronously to replicas using a changelog. In case of failure, user intervention is required to decide either to bring a master backup or choose a slave to become a master. If the second decision has been made, data consistency is not guaranteed. According to ArangoDB, slaves will typically be behind the master by a couple of seconds or minutes.

1.8.6 Conclusion

In conclusion, it is essential to point out that clustering by itself does not solve the problem of speeding up queries, improving availability, etc. It solves the problem only if data is effectively partitioned and intelligently replicated across the whole cluster. In a graph world, compared to a relation one, it is even harder, because we need to keep in mind that most commonly used edges and vertexes in the graph should be one the same database server. Otherwise, all the graph features, like finding the shortest path between two nodes by edges, will be significantly slowed down by the enormous amount of network communication and the superiority of the graph database over the relational one will be negligible.

Chapter 2

Enterprise data warehouse architecture

A data warehouse is a type of data management system that is designed to support business intelligence activities, especially analytics [14]. The architecture of a data warehouse, like house architecture, is different and depends on business use cases. Inmon is a data warehouse approach that is used by one of the project MMP customers. According to the approach data warehouses have following layers: Data Source Layer, Data Staging layer, Data Storage Layer and Data Presentation Layer. An example of a Data Warehouse architecture is shown in figure 2.1.

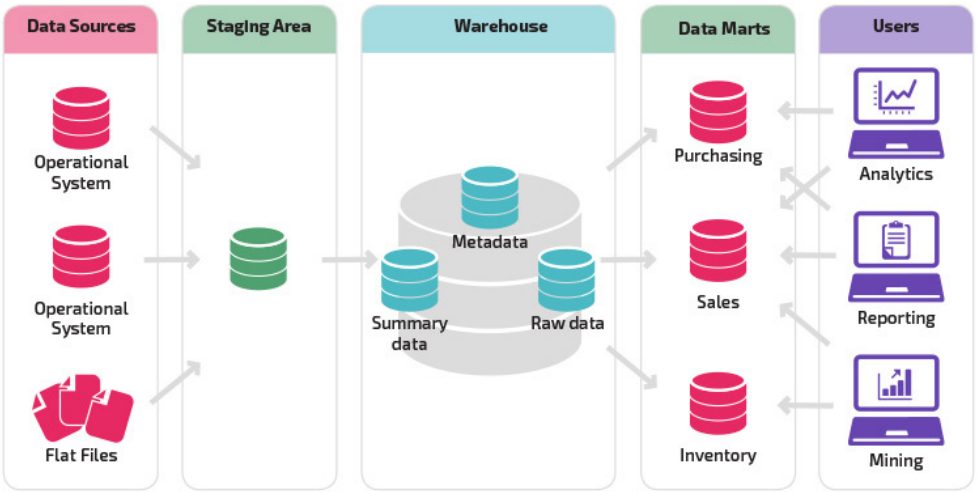

Figure 2.1: An example of a Data Warehouse architecture. Source: [15]

2.1 Data source layer

A data source layer is a layer where data is collected. Data can be structured (database), semi-structured (XML files, JSON) and unstructured (images, voice recordings, videos and so on). The data can be categorized as operational data, social media data and third-party data. Operational data includes product data, marketing data, HR data and other data. Social media, among other things, includes content popularity, web site hits, content page completion. Third-party data includes demographic information, survey data, etc.

2.2 Data staging layer

A data staging layer is responsible for extracting data from different external and internal sources, performing a high-level check on the data and storing all the information into a single landing database storage. The final result of staging will be clean and organized data that can be loaded into a data warehouse. This layer uses extract, transform and load (ETL) tools. An example of the instrument, which supports ETL is Informatica.

2.3 Data storage layer

A data storage layer is a single central repository in a data warehouse architecture. The design of the storage can be different depending on the business use cases. The data storage layer can be represented as a data warehouse, data marts or an Operational Data Store. Data marts can also be placed along with a data warehouse database. A data mart is a subset of the database, which stores the information of a particular function of an organization, such as sales, inventory, purchasing.

2.4 Data presentation layer

A data presentation layer is where a user interacts with structured data. This layer provides an ability to query, analyze the information and develop reports. Usually, a presentation layer includes a Graphical User interface.

2.5 Metadata

Metadata is an essential part of every data warehouse. Metadata is a set of data that describes information about the stored data. Without them, it would not be possible to determine the meaning of stored data. The metadata in a data warehouse can be categorized as technical metadata and business

metadata. The first one includes information about processes, repositories, a physical layer of data. In other words, it includes information about indexes, mappings, relationships, indexes, constraints, etc. Business metadata includes definitions of functionality, elements, how data is used in business and so on.

2.6 Mapping in a data warehouse

Mapping is an essential process of every data warehouse architecture. Passing through from one layer to anther data is being transformed into different structures. Without mapping metadata, the information about data origins will be lost. An analytical department uses information about data transformation to figure out firstly the sources of data and secondly data targets.

Chapter 3

Analysis and design

3.1 MMP DD Tool

The MMP Data Dictionary (DD) tool is an application for a data warehouse that stores and shows metadata of a part of a data warehouse. The metadata usually is loaded from `.pdm` files of SAP PowerDesigner, but also can be imported from other sources. The primary item of DD is a model. The model describes the internal structure of a particular aspect of a data warehouse. A model can contain an annotation, a description, tables, views, diagrams, source and target models (mapping), source and target tables (mapping as well), procedures and so on. Almost every object can have a description and annotation, which describes an object. A model can contain any number of tables. Each table includes columns, indexes, source and target tables and columns, etc. Views contain its SQL query and columns. Diagrams are graphical objects of models that are shown in PowerDesigner. Some object, such as models, table, columns can have source and target transformations. These transformations help to answer questions, such as where data came from and where it goes. Also, transformations are the focal point of the bachelor thesis. Transformations are going to be exported from the Physical Data Model and imported to a new model in a graph database. The Physical Data Model of the Data Dictionary Tool is a relational model that is designed to store the content of metamodels.

3.2 Analyzing requirements

The thesis aims to answer a question if it is worth implementing a relative part of mapping model using an existing relational mapping model to get better performance of some specific data flow operations or a current relational database is capable of effectively performing the data flow operations.

3.2.1 Use cases for comparison

As mentioned in the chapter of a data warehouse architecture, mapping is an essential part of capturing the process of transferring data between layers of a data warehouse. Sometimes data is transferred without changes, but mostly information changes. The amount of changes and transformations is not constant. It depends on data source and data itself. Moreover, there is usually more than one transformation between two objects. As a result, it is important to have a tool, which will support:

- finding the shortest path between two objects in transformations
- data source analysis
- impact analysis of object deletion

3.2.2 Additional requirements and subtasks

The process of building an application for data flow comparison on the previously specified use cases is following. The first step is to search for potential graph database management system. The limitation for the system is that it has to be open-source with a possibility for commercial use. The second step is to adequately design a mapping model in a graph database. The model should be scalable and extendable. Moreover, the model should also support multitenancy. The third step is to build an application for comparison. The only limitation here is to use a `.NET Core` environment. The limitation may affect choosing a graph database because it has to have a driver for `.NET` environment. The fourth step is to transfer data from a relational database to a graph one. Then data should be enhanced by tenant information. In the step, it is essential to consider how to automatize the process of transferring data. Last but not least, is to conduct a series of tests on both relational and graph database and write down a conclusion if it is worthy of using a graph database management system to store and to use it on the describe use cases.

3.3 Comparision of graph database management systems

The era of social networks has led to the active development of alternative database technologies. In 2010 ACID graph databases that supports horizontal scaling became available. Also in 2010, multi-model databases became available (such as ArangoDB and OrientDB). It increased their attractiveness for use in large projects. Ten years passed by, a lot of commercial and open-source graph databases come in sight, a lot of efforts has been made to improve already existing databases. Let's take a look at the most popular ones.

3.3.1 Neo4j

Neo4j is an open-source label property graph database with a GPLv3 license and a commercial license. According to DB-Engines.com ranking [16], it is the most popular graph database and it is the 21th most popular database [17]. Neo4j supports transaction data access and claims to have ACID properties in a stand-alone mode. The database can be scaled in both directions. In a cluster setup, it offers casual and eventual consistency. Casual consistency means that a client application is guaranteed to read at least its own writes [18]. The database supports a range of programming languages and frameworks: .Net, Java, JavaScript, Scala, Go, Python, etc. It has its Cypher query language and RESTful API for querying. Neo4j has native graph storage and offers index-free adjacency. As mentioned in the chapter of GDBMS, the benefits of this feature are questionable, especially in a cluster setup.

3.3.2 Microsoft Azure CosmosDB

Microsoft CosmosDB is a multi-model database service, which considers itself as a document, key-value, wide column store and a graph database. The database is under a commercial license and only hosted on Azure Services. CosmosDB is the second most popular graph database and placed on the third position in a document, key-value, wide column store rating [17] among other big players like Redis (a key-value store and a graph), Cassandra (a wide column store) and Neo4j(a graph database). Almost all of these competitors focus on a single thing, whereas CosmosDB focuses on all the things. In general, it stores data as a JSON and uses SQL as a query language with some extensions for a graph database. The database can be scaled vertically and horizontally. Azure CosmosDB allows developers to choose among five consistency methods: strong, bounded staleness, session, consistent prefix and eventual. Session consistency is recommended and set by default. The consistency method is placed right in the middle between durable consistency with latency and eventual consistency with the lowest latency. The service even allows developers to create their consistency level. The service offers migrations from MongoDB, HBase, Amazon DynamoDB, SQL Server, etc. CosmosDB supports Java, Javascript, Python, .Net environment, etc.

3.3.3 ArangoDB

ArangoDB is a document, key-value, graph DBMS with support of text-search. It has two versions: a community, which is under Apache V2 and an enterprise, which is a commercial one. The commercial version includes more encryption capabilities, claims to have better clustering mode with SmartJoins, SmartCollections and SmartGraphs. ArangoDB can be scaled in both directions. The database is placed on the third position in the DB-Engines graph ranking [17]. The DBMS can be turned to a set of data centres where there is the main

one, and others are asynchronous copies the primary datacenter. In a regular cluster mode, it offers either eventual or immediate consistency with support of ACID properties. ArangoDB supports full-text-search, which uses the view concept. Developers can create any number of these views. It is possible to perform complex searches across the whole graph, even in a cluster mode. The query language of ArangoDB is called AQL. The syntax is close to SQL with some extensions and small differences. ArangoDB supports a variety of programming languages: .Net, Java, JavaScript, Python, Go, etc.

3.3.4 OrientDB

OrientDB is a combination of a graph DBMS, a document and a key-value store. It also has two editions: community and enterprise. The community edition is under `Apache V2`. Enterprise edition includes replication and Multi data center support, non-stop backup, etc [19]. OrientDB can be scaled vertically and horizontally. Horizontal scaling includes the Multi-Master replication, sharding and multiple datacenter architecture. Replication is configurable. Users are allowed to set up a write quorum, which is recommended to be $N/2 + 1$, where N is a number of nodes. The database uses OrientDB SQL dialect. The dialect mainly differs in joins and some extensions for a graph database (shortest path, traverse, match, et cetera). OrientDB supports .Net, Java, JavaScript, Python, Scala and other programming languages.

3.3.5 Virtuoso

Virtuoso is a multi-model hybrid that supports management of data that is represented as relational tables and/or RDF graphs. The database is available with `GPLv2` license in a community edition and commercial licenses. The database supports scaling in both directions. Horizontal scaling options are only available with the enterprise license. Virtuoso has multiple replication methods: master-slave, master-master, chain, star, bi-directional. This database supports ACID for relational and RDF data, even in a cluster mode. The multi-model hybrid uses SQL extended by Federated SPARQL as a query language. Virtuoso has drivers for .Net, Java, JavaScript, Python and others.

3.3.6 Amazon Neptune

Amazon Neptune is a combination of a graph database and an RDF store. The system is offered as a cloud-based only Amazon service. As others already mentioned database management systems, this one supports vertical and horizontal scaling. Resources of each machine can be configured. In a cluster mode, it offers asynchronous replication for up to 15 devices. It uses active-failover replication technique. Amazon Neptune uses SPARQL and Gremlin as a query language. It supports .Net, Go, Java, JavaScript, Scala, Python and other languages.

3.3.7 Conclusion

Neo4j is an excellent database with a big community of developers using it. However, it is under the GPLv3 license, which means that the system cannot be chosen. CosmosDB is strongly coupled with Azure and works only as a service, which force to use Azure Services. Amazon Neptune, like CosmosDB, is cloud-based only commercial service. Virtuoso uses an RDF and a relational model, and it is under GPLv2 license. A label property graph model comparing to RDF-Triplestore model seems to be more flexible and intuitive; it has more intuitive query languages. OrientDB has an attractive Apache V2 license in a community edition. However, according to DB-Rankings, which constantly tracking the popularity of databases, OrientDB loses to its competitor (ArangoDB) in gaining popularity. Also, it has a questionable multi-master replication architecture, compare to active failover architecture in ArangoDB. ArangoDB is under Apache V2 license in a community edition. It has excellent support for the .NET environment, including drivers, which implements LINQ querying. In the case of database growth, it can be horizontally and vertically scaled, even with a community license. In conclusion, the ArangoDB had been chosen as a graph database for storing and analyzing data flow operations.

3.4 Design of a mapping model in RDBMS

Project MMP in the Data Dictionary tool stores mapping information in `Model_Mapping`, `Table_Mapping` and `Column_Mapping` tables. All of these tables can be potentially stored in just one table, but one of the main reasons for that implementation is optimization. All of these mappings contain information about the source and target objects. In case of `Model_Mapping` these are `Source_Model` and `Target_Model`. `Table_Mapping` additionally has the source and target tables. Finally, `Column_Mapping` has the source and target models, tables and columns. These source and target objects are subsets of existing Models, Tables and Columns. Although mapping tables do not contain integer keys, instead, it contains names of the objects, which usually are `nvarchar(254)`. Having fixed length is important for indexes because as of SQL Server 2016, the maximum allowable size of the combined index for a non-clustered index is 1700 bytes [20]. This denormalization is added to improve the performance of querying. Otherwise, almost every query will have multiple joins, which will significantly affect the querying speed.

3.5 ArangoDB liminations

ArangoDB is a schemaless graph database management system. Data in the system is stored in JSON documents. Schema is defined per document, not

per collection. Among other things, it means that two documents can have different properties. In reality, ArangoDB groups documents by its schema to save storage. But it is important to keep in mind that indexes are defined per collection. In case if every document in a collection will have different schema, it is worthless to define indexes.

Even though ArangoDB is a schemaless database, an edge collection can join up to two types of objects. The relation can be created either within the same data collection or between two collections. It is impossible to define an edge collection, which, for example, connects a document from A collection and a document from B collection, and at the same time connects objects from collections B and C.

3.6 Design of a mapping model in LPG GDBMS

There are multiple ways how to represent mapping in a graph database. Each representation has its cons and pros, but before going into details of each representation, it is essential to point out some MMP requirements for representation. Main MMP demands of representation are an abstraction, scalability and speed. The first two are essential since every IT product has a rapidly growing number of functional requirements. A graph representation should be abstract and scalable in a way that instead of adding representation for a similar demand, it would be easier and faster to extend an existing representation.

In the following models, some properties are omitted for brevity, such as a tenant. The information can be either defined in a separate collection and connected using edges or can be stored in the objects itself. In a relational world, to reduce numbers of anomalies, and speed up an update on tables, the data is usually transformed into the third normal form. In case of frequent reading, data is denormalized, because joins are expensive. In the graph world, there are no joins. This leads to a conclusion that theoretically in a graph database it is better to have normalized collections (at least in a standalone mode). Although having tenant information in a separate collection does not improve the speed of deletion, because during removal all the edges that connect object vertexes with a tenant vertex have to be deleted as well. Moreover, having the information separately may take more space than having it right in an object node itself.

3.6.1 Column mapping graph

The first model is represented by two data collections. The first one is `ColumnVertex`, which is a vertex collection and the second one is the edge `TransformsToEdge` collection. The representation is shown in figure 3.1. Each node is a combination of a model, table and column code. Nodes are connected using transformation edges. The benefit of the representation is performance.

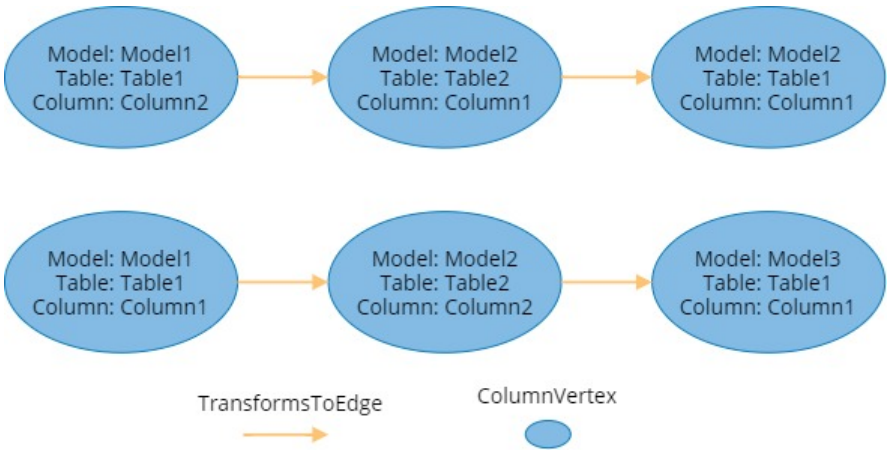

Figure 3.1: Column Mapping graph

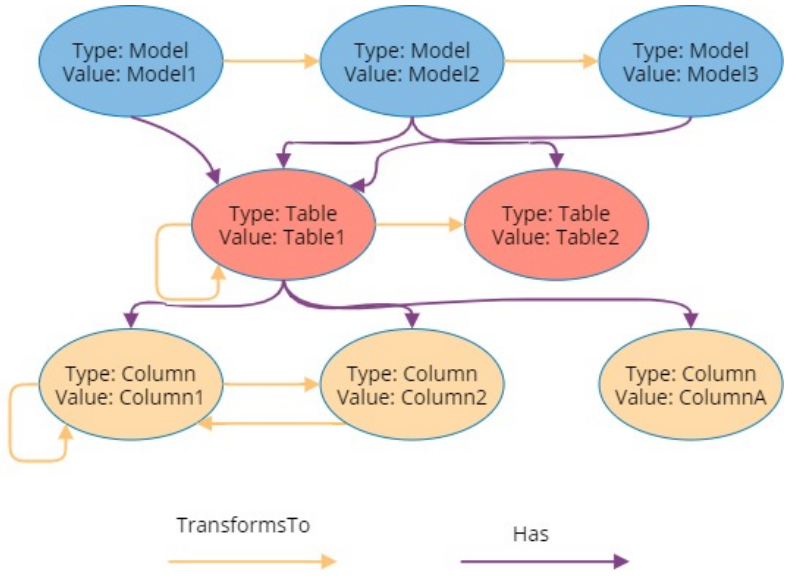

Figure 3.2: Metadata graph representation with unique vertexes

An index can be defined on these three properties, and only one lookup is required to find a particular object. The representation though has a huge drawback of poor scalability. Since column mapping is not the only type of mapping, it needs to define separate collections for model mapping, table mapping, etc.

3.6.2 Metadata graph representation with unique vertexes

The second model is called the metadata model. The representation is more generic compare to the first one. Since it is impossible to have an edge collection that connects more than two different types of vertex collections, an object type is defined as a separate property of the `ObjectVertex` collection. The `Has` edge is added to capture hierarchy, i.e. a table belongs to a model, and a column belongs to a table. The `TransformsTo` is an edge collection that connects `ObjectVertices`. With this concept, there is a possibility of unexpected relations. For example, columns can have models and models can transform into tables. The problem can be solved by triggered data checking.

The model is contained with unique objects. Having unique objects reduces the size of the collections and improves the speed of querying. Although it has a vast problem. The problem is that different models can potentially have tables that have the same name, but have different columns. For example `Model1` has `Table1` that contains `Column1` and `Column2` and `Model3` also has `Table1`, but with `ColumnA`. As a result both of these models will share the table `Table1` that has `Column1`, `Column2` and `ColumnA`. This case is also shown in figure 3.2. That is why the model cannot be used to represent mapping.

3.6.3 Meta model representation with non-unique vertices

The representation is similar to the previous but without unique objects. This solves the problem with losing hierarchy. Having a generic model for different types of objects makes it scalable. The representation contains not only information about column mapping, but also table and model mapping. Moreover, specific attributes based on the Type attribute can be stored either in the objects itself or in different collections and connected using edges.

The model has some disadvantages. The first one is that most of the data is stored in a single `ObjectVertex` collection, which may lead to slower querying. Though this can be improved by adequately using different types of indexes. Choosing indexes on the model is described in the following section. The second negative aspect is to find a particular column; it needs to find a model, which has a specific table, and a column that belongs to the table. But ArangoDB, like other graph databases, promises fast traversing (at least at a standalone mode), which should solve the more complex searching.

3.6.4 Conclusion

As mentioned above, the main demands are querying speed (read operations are more critical than update and delete) and scalability. The first model might be potentially fast, but it cannot be chosen since the representation has poor scalability. The second one also cannot be selected because it loses hierarchy information after a transformation. As a result, the third metamodel is selected for comparison with a relational model.

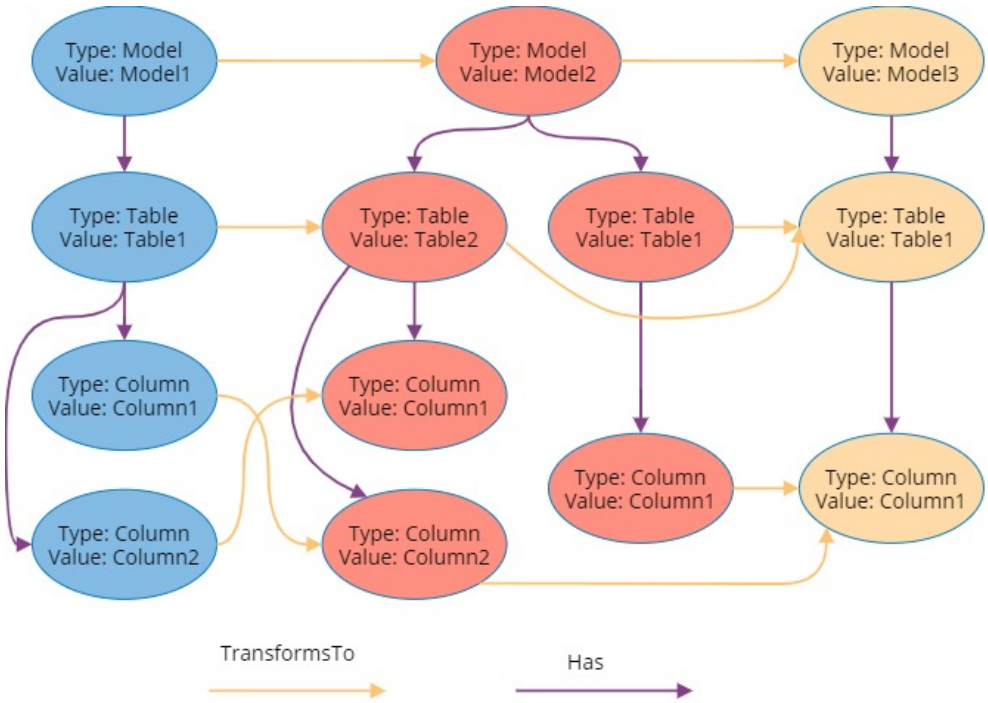

Figure 3.3: Meta model representation with non-unique vertices

3.7 Indexes on the Meta Model

By default, ArangoDB adds indexes to the document identifiers. Additionally, it adds indexes to source and target vertexes to every document in edge collections. All of these was done to improve the performance of traversing operations. However, performing traversing or finding the shortest path requires object identifiers. As mentioned in the previous section, to find a particular column, it needs to find a model that has a particular table which has a particular column. Performance of the operation definitely can be improved by adding combined indexes that includes `Repository_Code`, `Object_Type` and `Object_Name`.

Performance of the data flow operations in the relational mapping model strongly depends on indexes, since finding object neighbours involves searching for rows that have a particular source or target columns, such as `Model_Code`, `Table_Code` and `Column_Code` in case of looking for all the incoming column neighbours, and `Source_Model`, `Source_Table`, `Source_Column` in case of finding all the outgoing neighbours of a particular column. This leads to a conclusion, that should be defined two combined non-clustered indexes for every mapping table, which will include either source or target columns and also each index should include a tenant name because each query will eventually have a tenant cut.

3.8 Choosing a programming language

The Metadata Management Platform uses `.NET Environment`, to be precise .NET Core Environment. This is a relatively new framework, the first version of which was released by Microsoft on June 27, 2016. Benefits of this framework are that it is under the MIT License and it is a cross-platform successor of .NET Framework. Since Oracle had changed the licensing of Java language, it helped the new framework from Microsoft gain popularity even more. As of 2019, according to StackOverflow, it is the most loved framework [21]. Other features of the Environment are strong support from Microsoft, performance, built-in dependency injection and Language Integrated Query (LINQ).

The output of this thesis is a comparison of a relational and a graph database, and also an application that will perform operations and tests on these databases. The application may be modified and used for the data flow operation as a separate microservice application. This means that it can be written in any language. But since the project MMP uses .NET Core Environment, it is not wise to have a different environment, because it is harder to maintain, requires installing different packages for each customer. To sum up, that were the main reasons why the application for comparison is written in C# on .NET Core Platform.

Chapter 4

Realization

4.1 The application overview

The result of the thesis is a backend application that exposes REST API for data flow querying. Each endpoint of the REST API is designed for comparison. When it gets a request, it stores the request time, and when it is done with calculations, it returns a JSON that additionally contains server calculation time. This time includes only calculations and does not include time for accepting a request and sending a response.

The application exposes two endpoints for data flow operations. One for getting the shortest path and one for traversing either all the incoming or all the outgoing related objects. Database technology is sent as a parameter. Based on a request, the application connects to a relational or a graph database management system. In case of a graph DBMS, all the graph operations are performed right in the database. In other words, the application sends a request, for example, for finding the shortest path between two columns in transformations, and then gets a result. Unconditionally it is a significant benefit of a graph database management system since in production; it is not a common scenario when a database and an application runs on the same machine. In case of a relation database, these operations are implemented in the application. There is a lot of communications happening between the application and an RDBMS. Moreover, an object-relational mapping may also add additional latency to performing the operations. To perform fair comparison were implemented stored procedures for the specified use cases. These procedures stores result in temporary tables. After procedure execution, the application reads data from these tables and returns results from the tables to clients.

The data flow API is built for different types of objects. Instead of having separate endpoints for column, table and model mapping, it combines all of them in two endpoints. Additionally, the application exposes endpoints for

getting column, table and model description that also contains its identifiers. These identifiers are strings. Since ArangoDB uses string keys, sending strings using `GET` method is not wise. The key should be firstly encoded and then decoded, because it may contain occupied symbols, such as backslash. It was the reason why `POST` has been chosen over `GET` method.

The application endpoints are following:

- `POST /api/data-flow/technology/shortest-path`
- `POST /api/data-flow/technology/traversal`
- `POST /api/object/technology/model`
- `POST /api/object/technology/table`
- `POST /api/object/technology/column`
- `POST /api/object/export-to-graph-model`

The application also provides an API for exporting mapping data from SQL Server. The endpoint transforms data from a relational database into graph collections and stores data using a specified path.

Additionally, the application has a presentation layer, which is automatically generated from code. Swagger is an open-source framework, which generates beautiful `HTML` pages that contain a list of exposed endpoints. The tool generates it from the code during the build time. So backend developers do not have to constantly update an API documentation and frontend developers always uses actual endpoint information for building frontend applications. Moreover, the graphical user interface of the tool supports querying, including `GET`, `POST`, `PUT` and `DELETE` methods.

4.2 Architecture of the application

The application is built on Microsoft .NET Core Web SDK and consist of five layers: External Data, Data, Service, Controller and Presentation layers. The presentations layer is automatically generated during build time using Swagger. A package diagram of the application is shown in figure 4.1.

External Data Layer is consists of database management systems. These are the ArangoDB graph database and Microsoft SQL Server object-relational database.

Data layer represents objects that exist in an RDBMS and GDBMS. Each entity in the layer can have mapping attributes, constraints, related objects, foreign keys, etc. In other words, this layer contains definitions for object-relational and graph mapping.

The service layer is a logic layer of the application. It consists of Repositories, Services, Utilities and Models. A repository implements create, read,

update, delete (CRUD) operations using entities from the Data Layer and Language Integrated Queries (LINQ). Additionally, it implements operations on top of CRUD methods, such as finding the shortest path between objects. The only limitation is that it cannot use another repository. Usually, each class in Data Layer has its repository. Although, sometimes it needs to use a combination of different repositories to handle a request. It was the reason why Services were added to the application. The services are usually defined per controller. Utilities consist of methods that do not use database entities. An example can be export to file in `CSV` format. Models, like Data Entities, have properties and are used for storing and exchanging data, but models do not take part in an object-relational or graph mapping. There are three types of models: Settings, Models and View Models. Settings are used for accessing the application settings. Models are the ones that are parsed from a `HTTP` request. View Models used for sending data back to a client.

Controller layer defines API, checks authorization and roles, gets requests, parses request parameters, invokes a service with the parameters and returns a response (usually a View Model). According to best practices, controllers should be as thin as possible to enforce separation of concepts and to make code more testable [22].

A class diagram of the application is divided into three parts and shown in figure 4.2, figure 4.3 and figure 4.4. The method parameters are omitted for brevity. The `ControllerBase` and `DbContext` methods and properties are also omitted for brevity and because these abstract classes are parts of Entity Framework Core library.

4.3 Adding multi-tenancy to application

A Cloud Applications refers to an application that is deployed in a cloud environment rather than being hosted on a local server or machine. Instead of maintaining each component of the application separately for each customer, it manages to serve different customers in one global environment. To handle the demand, an application should be able to determine which customer it belongs to. One way of doing this is by adding tenancy. For the database world, it may mean adding a tenant column to every single table. In the application, each table and each document has a repository code. The code is a string, which is unique for each customer. For now, the information is stored in appsettings.json, but it also can be stored in a database. Each database request in the application has a tenant `WHERE` cut to ensure that data is taken from the correct sources.

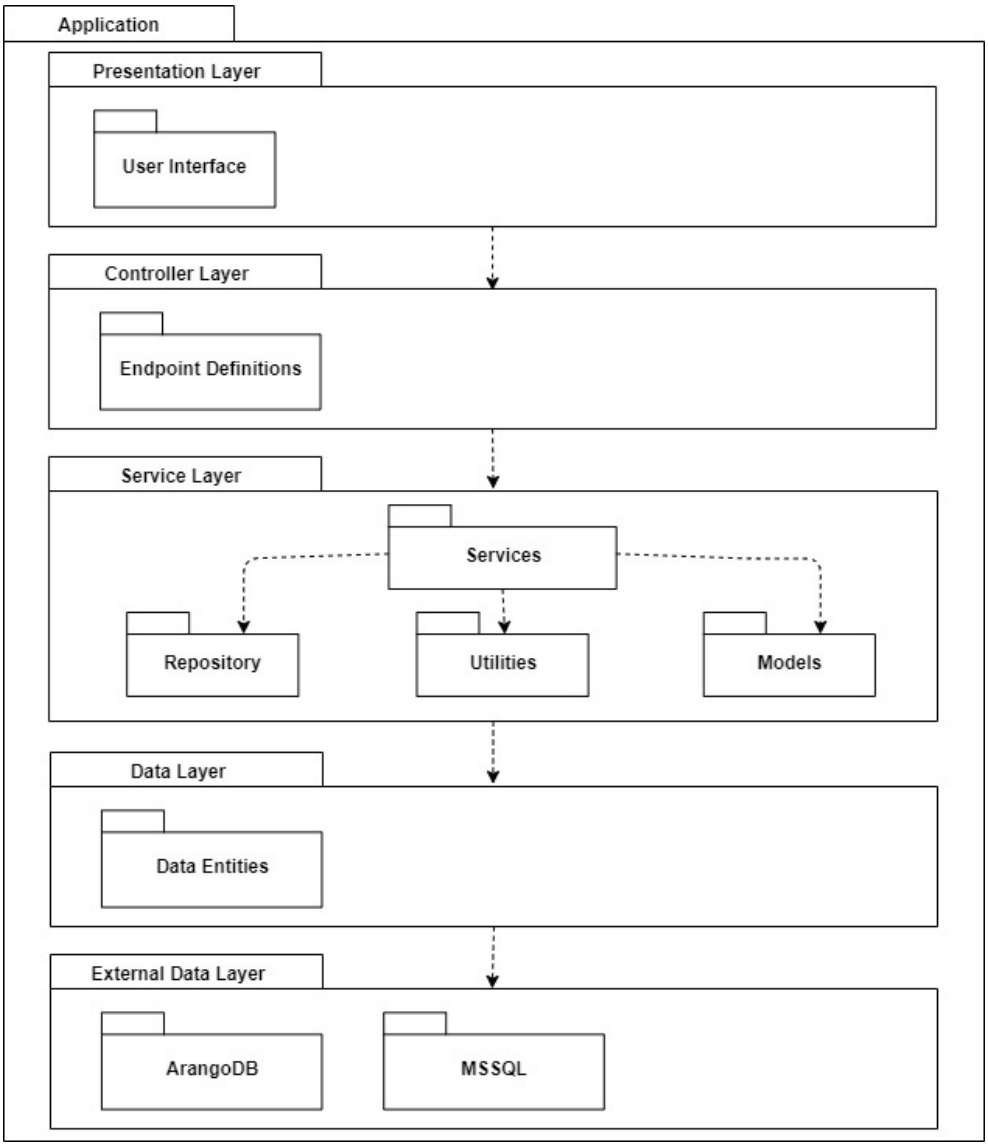

Figure 4.1: A package diagram of the Database Comparison Application

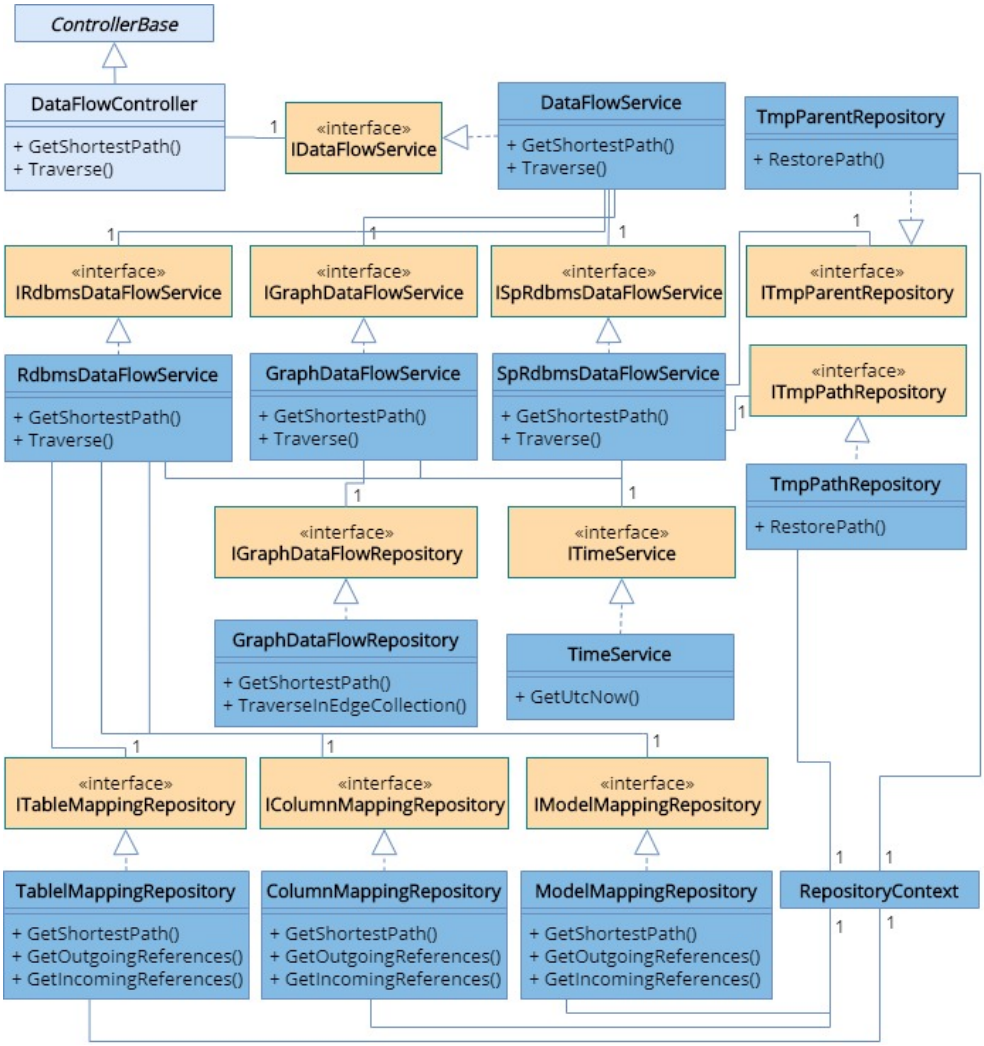

Figure 4.2: The `DataFlowController` class diagram

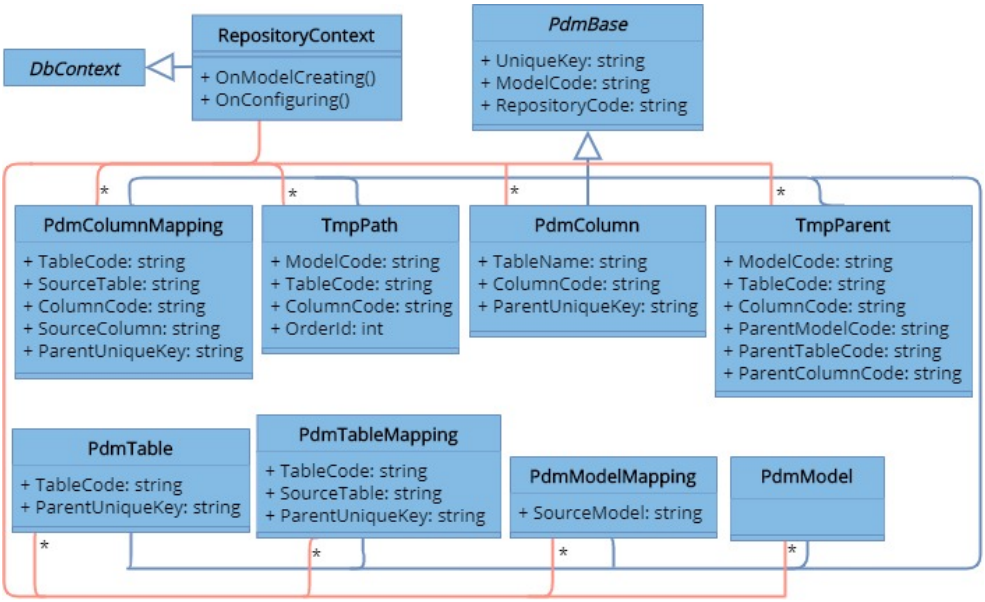

Figure 4.3: The `RepositoryContext` class diagram

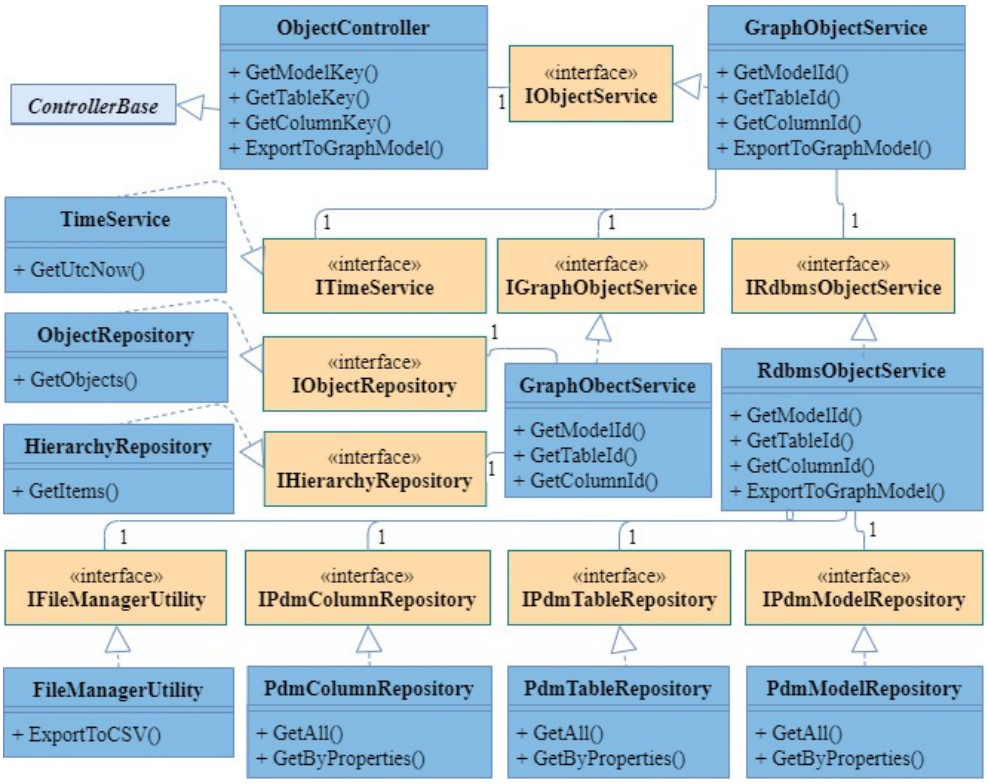

Figure 4.4: The `ObjectController` class diagram

4.4 Graph operations in a RDBMS

Mapping in a relational database defined in model, table and column mapping tables. For each of them are implemented data flow operations, such as finding the shortest path between two objects, getting all outgoing and incoming objects. Although, an API is designed to be general for all the types of mappings in a way that it takes only identifiers. Then, based on the identifiers, an application can determine what type of mapping should be used.

4.4.1 The shortest path algorithm

Finding the shortest path between two objects in transformations is implemented using a Breadth-First search algorithm (BFS). The pseudo-code is defined in listening 4.1.

Listing 4.1: The pseudocode of the shortest path algorithm

```
GetOutgoingShortestPath(sourceColumn,targetColumn)
{
    var queue = new Queue<Column>();
    var depth = new Dictionary<Column, int>();
    var parents = new Dictionary<Column, Column>();
    depth[sourceColumn] = 0;
    queue.Enqueue(sourceColumn);
    while (queue.Any())
    {
        var y = queue.Dequeue();
        GetOutgoingNeighbours(y)
            .Where(x => !depth.ContainsKey(x))
            .ForEach(
                x =>
                {
                    depth[x] = depth[y] + 1;
                    queue.Enqueue(x);
                    parents[x] = y;
                }
            );
        if (depth.ContainsKey(targetColumn)) {
            //the shortest path has been found
            break;
        }
    }

    return RestorePath(targetColumn, parents);
}

RestorePath(targetColumn,parents)
{
    var path = new List<Column>();
```

```
    path.Add(targetColumn)
    var curr = targetColumn;
    while (parents.ContainsKey(curr))
    {
        path.Add(parents[curr]);
        curr = parents[curr];
    }

    return path;
}
```

The time complexity and memory complexity is the same as in BFS and can be expressed as $\mathcal{O}(|V| + |E|)$. Where |V| is the number of vertexes and |E| is the number of edges. There is one interesting thing in terms of database comparison. The thing is the method of getting neighbours (`GetOutgoingNeighbours`). There are at least two possible ways how to implement it. The first way is to load all of the mappings to memory and then query for object's neighbours. The second way is to query a database for each object. The first one is more efficient when memory is capable of storing all of the information. But when it comes to the big data world, it is hardly imaginable to load all of the data. Also, if all of the data are loaded at once, it is no longer database comparison. In that case, the only thing a database has to do is to return all of the data. That was the reason why was chosen the second implementation of `GetOutgoingNeighbours`. In this way though there is a lot of communication between a database and an application. It should not affect results since the application and the database server are running on the same machine. Though there are some speed penalties of object-relational mapping, that should be taken on the account. For that reason, these operations were also implemented as stored procedures, which write results to the temporary tables.

4.4.2 The related objects algorithm

Related objects in transformations can be found using Depth-fist or Breadth-first search algorithms. The output of the second algorithm seems to be more natural for understanding related objects. The code of the algorithm is similar to finding the shortest path between two objects. It differs only in the fact, that does not have an `IF` condition in the while loop, it does not have to have a parent dictionary since related objects are added right in the for each loop. Time and memory complexity of this algorithm is the same as the previous one.

4.5 Finding an object in a graph database

Since a meta mapping model in a graph database is more generic comparing to a relational one, finding a particular object is not so simple as in a relational model. To find a model it needs to perform only one lookup in `ObjectVertex` collection using the model name, tenant name and object type. Finding tables and columns is more complicated. Firstly it looks for a particular model. If model has been found, then it searches in `HasEdges` collection, where the owner id is an identificator of the model and item is a table in the same tenant as owner, plus it has to have a desirable table name. The same thing is performed for a column. In case if some of the objects has not been found, a not found exception is thrown.

4.6 Graph operations using ArangoDB

ArangoDB, like many other graph databases, has built-in operations for traversing and finding the shortest path between two nodes. Because of that, the application sends only one request to a graph database, the system performs an operation and sends results back to the client. An example of the query for finding the shortest path is following:

```
FOR vertex[, edge]
  IN OUTBOUND|INBOUND|ANY SHORTEST_PATH
  startVertex TO targetVertex
  GRAPH graphName
  [OPTIONS options]
```

The query for getting all the outgoing or incoming objects is following:

```
FOR vertex[, edge[, path]]
  IN [min[..max]]
  OUTBOUND|INBOUND|ANY startVertex
  GRAPH graphName
  [PRUNE pruneCondition]
  [OPTIONS options]
```

The shortest path or traversal queries can be performed not only on a graph but also on a set of edge collections. The comparison application uses a `TransformationEdge` collection instead of creating a graph.

4.7 Test-driven development

The application was built using Test-Driven Development. After defining models and designing service interfaces, for each service were written tests. Then were implemented the data flow operations.

Testing was complicated by the fact that the data flow operations were implemented in three different ways: using built-in graph database data flow operations, by wringing stored procedures, and by implementing an application that is continually asking for neighbours. Testing the application implementation is the easiest one because it can use an in-memory database. Testing two others is more complicated. A connection to ArangoDB and SQL Server can be mocked, but by mocking the testing does not make sense, because it will test nothing. For that reason were locally created test databases on ArangoDB and SQL Server. Before every test, data in the databases is truncated and then inserted using initial seeders.

4.8 The data transformation from an existing relation DB

Since a graph and a relational model are different, the data from a relational model has to be transformed. The transformation can be done by writing SQL mapping scripts and exporting data to comma-separated values (CSV). Considering that a schema of the mapping model in a relational model should not change so often, the transformation can also be performed right in the application.

A relational mapping model has three types of mappings: model, table and column. Moreover, it contains models, tables and columns in tables `PDM_Model`, `PDM_Table` and `PDM_Column` respectively. A graph meta model consists of `ObjectVertex`, `HierarchyEdge` and `TransformationEdge`. Tranformation is implemented following these rules:

1. `PDM_Model`, `PDM_Table`, `PDM_Column` are transformed to `ObjectVertex`. Additionally it has `ObjectType` set to 1, 2 and 3 respectively.

2. When transferring `PDM_Table`, hierarchy edges are added with `_from` equals to a model, which it belongs to, and `_to` equals to the id of the table.

3. `PDM_Column` information hierarchy is also replicated in hierarchy edges with `_from` equals to a table that it belongs to and `_to` equals to the column identifier.

4. `Model_Mapping` joins pairs of `ObjectVertex`, which have `ObjectType` is equal to one, by `TransfomrationEdge`. `Source_Model` is an identifier of the source object (`_from` property), `Model_Code` is a target identifier (`_to` property).

5. `Table_Mapping` and `Column_Mapping` joins pairs of `ObjectVertex` with `ObjectType` equals to two and three respectively.

What should not be performed in the application is inserting transformed data. ArangoDB clients usually using an `HTTP` connection, which tremendously slows down inserting, especially if it commits every single row. But there is a solution that. Arangoimport is a command-line application that was built for importing data in `JSON` or `CSV` formats. When data is transformed into one of those formats, database generated keys cannot be used. The reason is that every edge collection consists of object identification properties, such as `_from` and `_to`, which are identifiers of vertex collections.

Chapter 5

Performance testing

5.1 Aims

The chapter includes a series of performance tests using the specified operations: finding the shortest path between objects, getting all related objects using incoming and outgoing transformations. The result of these tests is a conclusion either it worthy of using a graph database management system for mapping operations or not.

5.2 Testing environment

All of the tests were conducted on a laptop with parameters, which are specified in table 5.1. During tests were launched only essential programs for running the operating system.

5.3 Test data

Test data was extracted from one of the project MMP customers. Additionally on the data was performed a series of anonymizations. The extracted data was inserted into model, table and column tables. Mapping data was inserted into model, table and column mapping. Additionally was added a column mapping

Notebook	DELL Latitude 5495
Operating system	Windows 10 Pro (64-bit)
Processor	AMD Ryzen 5 PRO 2500 2.00 GHz
RAM memory	16 GB
SSD disk	yes

Table 5.1: Notebook parameters that was used for testing

Table name	Count of rows
PDM_Model	24038
PDM_Table	130326
PDM_Column	1312478
Model_Mapping	23838
Table_Mapping	140584
Column_Mapping	924522

Table 5.2: Count of rows in MSSQL database

Collection name	Count of documents	Import time
ObjectVertex	1466836	55s
HierarchyEdge	1442798	73s
TransformationEdge	1088946	71s

Table 5.3: Count of documents and import time in Arango database

that consists of one hundred thousands transformations. All of the data was placed in a single repository. Then it was transformed into a `CSV` format using the application endpoint for transformation, then was imported into ArangoDB. Amount of rows in tables and amount of documents in collections is described in table 5.2 and table 5.3, respectively. Import time is shown in the table 5.3.

5.4 Test measurements

Following tests were conducted using built-in ArangoDB graph operations (shortest path and traversals), application-side implementation of data flow operations using SQL Server and an implementation that executes stored procedures that write results to temporary tables in SQL Server and then reads results from that tables. The last implementation was added to prove that communication and an object-relational mapping does not significantly affect the speed results of the operations as the internal architecture of databases itself.

The following test measurement charts contain the following abbreviations:

1. `arango` is an implementation that uses built-in ArangoDB operations
2. `mssql app` is an application-side implementation that uses SQL Server and indexes on tables are not defined
3. `mssql sp` is an implementation that uses custom stored procedures and indexes are not defined as well

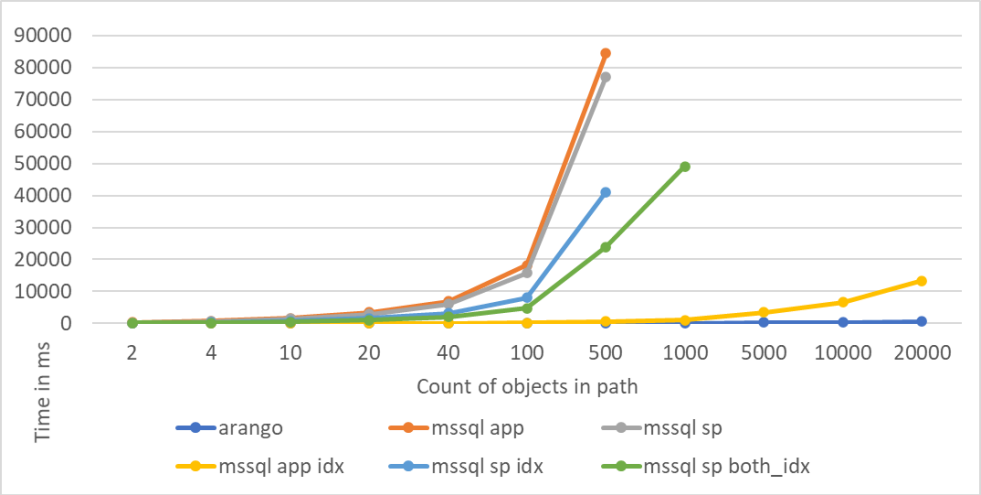

Figure 5.1: The shortest path measurements between two objects, where each vertex has one incoming and one outgoing edge

4. `mssql app idx` is similar to the second one, except it has predefined indexes on the mapping tables

5. `mssql sp idx` is an implementation using storing procedures with indexes on some columns in the mapping tables

6. `mssql sp both_idx` means the same implementation as the `mssql sp`, but with predefined indexes on the mapping tables as well as the temporary visited tables

Test measurements are shown in the charts. Since test measurements are discrete values, the charts should only contain dots. But for better clarity, these distinct values are connected.

5.4.1 The shortest path tests I

The first test measures test finding the shortest path between two objects. These source and target objects are located in the structure that is called a path, i.e. each node has only one incoming edge and only one outgoing transformation edge except the source and destination vertexes, which have only one outgoing edge and one incoming edge. Test results are shown in figure 5.1 and 5.2.

The first thing that stands out a mile is that the C# implementation and implementation that stored procedures are almost equal. The second one is even slower. And slower simply because the table insertions and deletions are not cheap. Adding an indexes that covers `Repository_Code`, `Model_Code`,

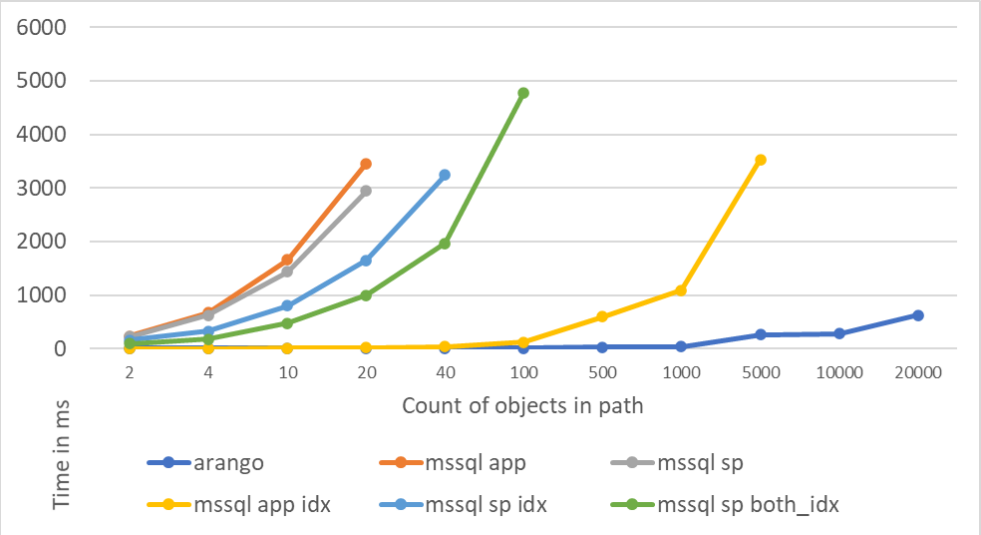

Figure 5.2: The shortest path measurements between two objects, where each vertex has one incoming and one outgoing edge (scaled)

`Table_Code` significantly improved the `mssql app idx` implementation results. Also, it improved the implementation that uses stored procedures. The next question was whether defining an index for the internal table of visited vertexes will improve the result. Adequately defined indexes significantly improve read speed, but it slows down modification and insertion. Turned out that adding an index to the table of visited vertexes was worth it. Although it did not help, because the latency of finding the shortest path which consists of more that one thousand vertexes is about fifty seconds.

In this test, ArangoDB is shining, because, in this particular test, it is capable of finding the shortest path that consists of twenty thousand objects in less than a second.

5.4.2 The shortest path tests II

The first test conditions were far from real. In practice, every object usually has more than one transformation. In these tests, each object had four neighbours: one incoming and three outgoings. The test results are illustrated in figure 5.3 and 5.4.

Comparing to the previous test ArangoDB is still tremendously fast, but the application-side implementation and the stored procedure implementation are more than three times slower. Mainly because of the amount of iterated objects, which was increased three times comparing to the previous test.

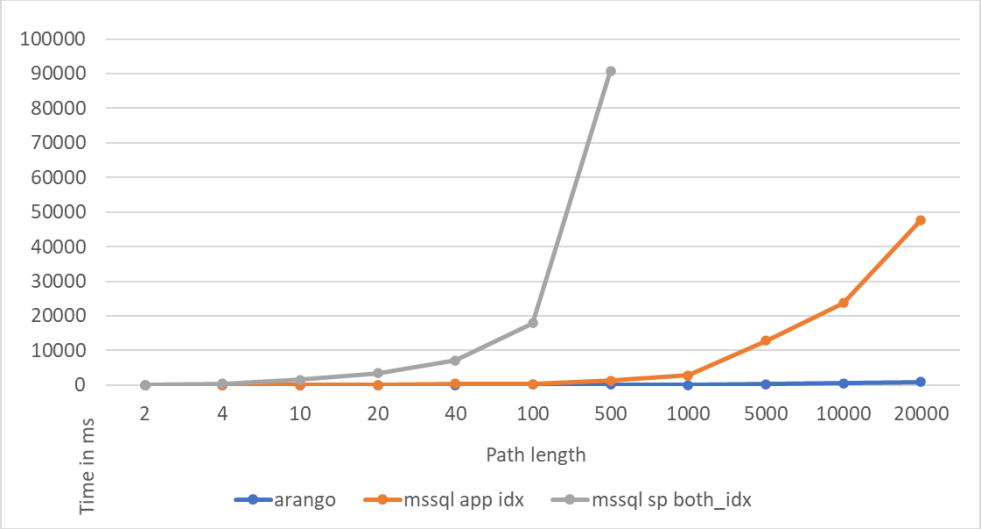

Figure 5.3: The shortest path measurements, where each vertex has one incoming and three outgoing transformation edges

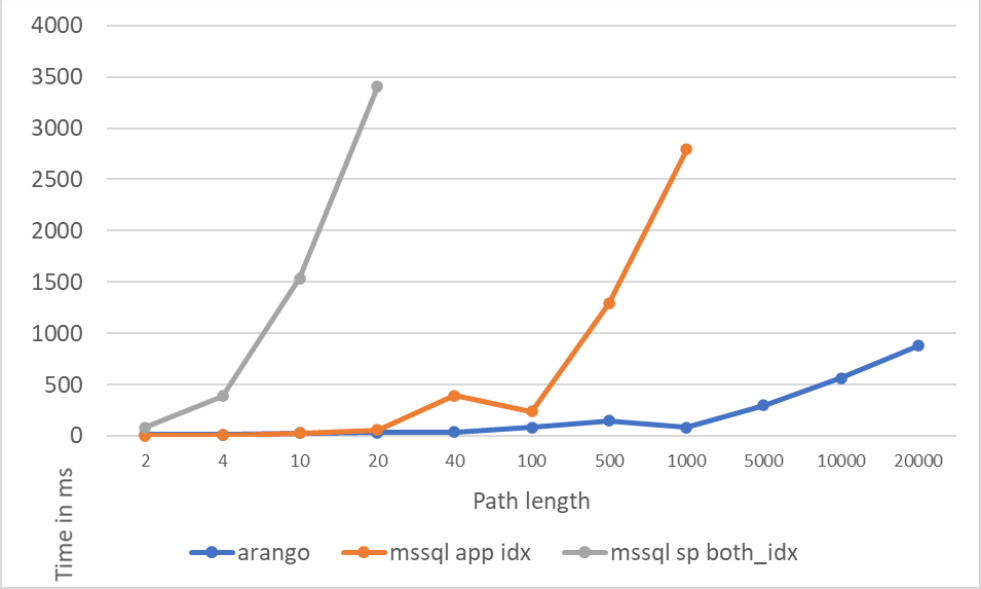

Figure 5.4: The shortest path measurements, where each vertex has one incoming and three outgoing transformation edges (scaled)

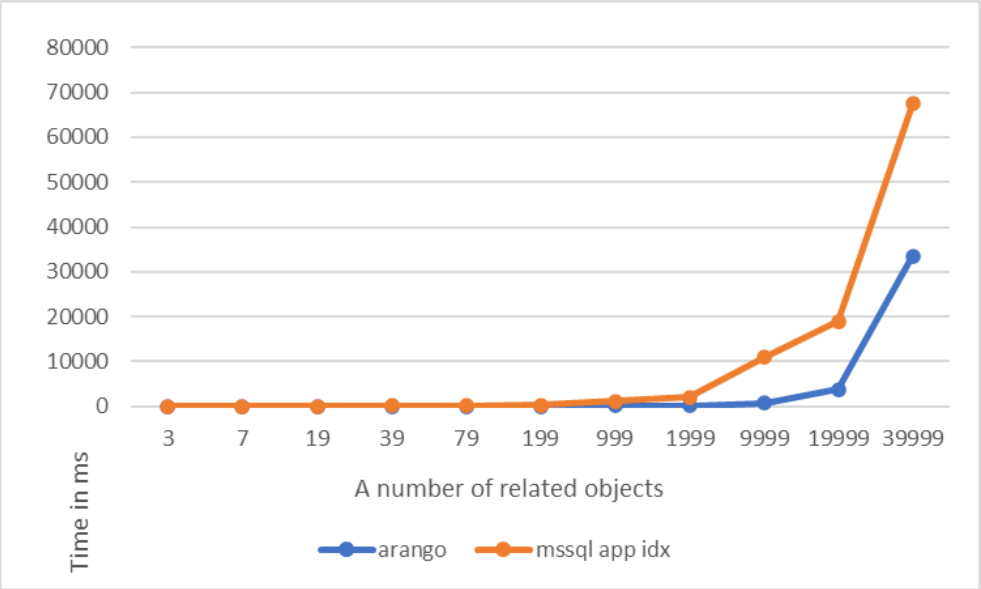

Figure 5.5: The incoming flow measurements, where each vertex has one incoming and three outgoing transformation edges

5.4.3 The incoming data flow tests

Finding all the related objects uses the same Breadth-First Search algorithm as finding the shortest path, which leads to a conclusion that the speed of getting all the related objects will be the same. The measurements, which are shown in figure 5.5, proves that.

What is not expected is that ArangoDB noticeably loses in performance after it reaches the amount of about twenty thousand objects. At forty-thousand objects, it has almost the same latency as the C# implementation with indexes. It might be because of the ArangoDB settings, to be more precise, it reaches the memory limit for buffering the results.

5.4.4 The outgoing data flow tests

The implementation of getting the outgoing data flow is almost the same as getting the incoming data flow, except it iterates through the incoming edges. The results of the test are illustrated on chart 5.6.

5.4.5 Finding an object by attributes

As mentioned in the previous chapter, to perform the data flow operation, a client should know the object identifier(s). Finding a particular object by attributes is usually expected to be in a range of milliseconds. As shown on chart 5.7, the ArangoDB implementation is able to get a column object in

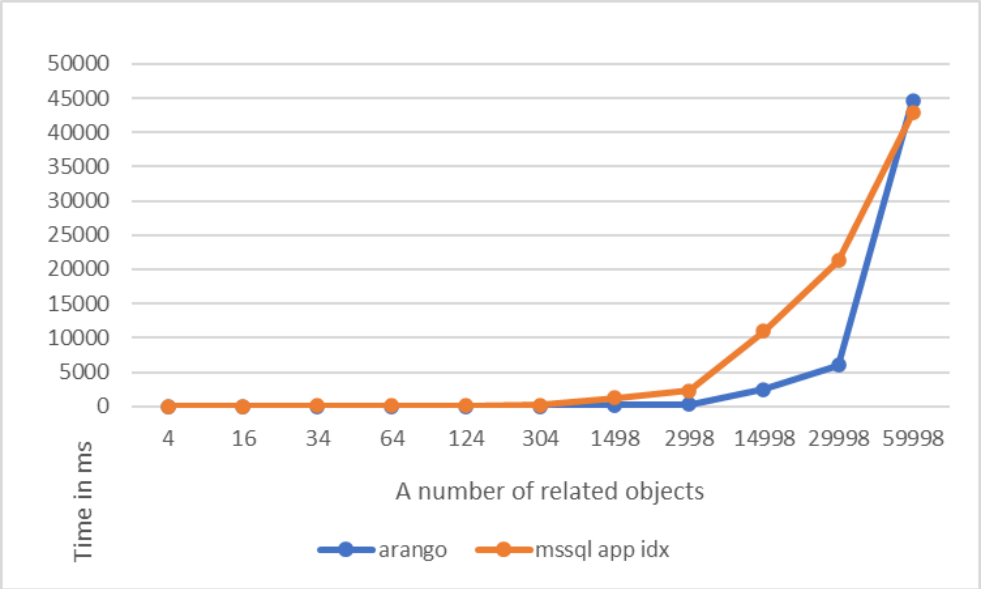

Figure 5.6: The outgoing flow measurements, where each vertex has one incoming and three outgoing transformation edges

about seven seconds. Although adding an additional combined index on the `Object_Vertex` collection had significantly improved the results, which are also shown on figure 5.8.

5.5 Conclusion

After conducting the series of tests, it is clear that firstly, a stored procedure implementation is the slowest one. Secondly, constantly database querying for object neighbours can work relatively well if indexes are adequately defined. Thirdly, ArangoDB should have as much RAM space as possible to get the advantages of being a graph database. Otherwise, the performance is almost the same as the C# implementation that uses SQL Server (with predefined indexes on the mapping tables).

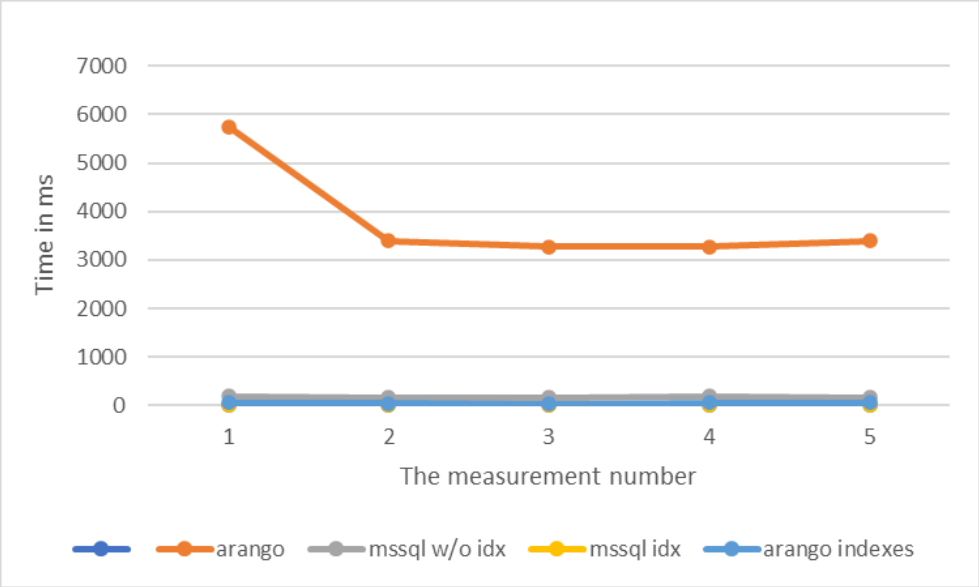

Figure 5.7: The measurements of getting a particular column object by its attributes

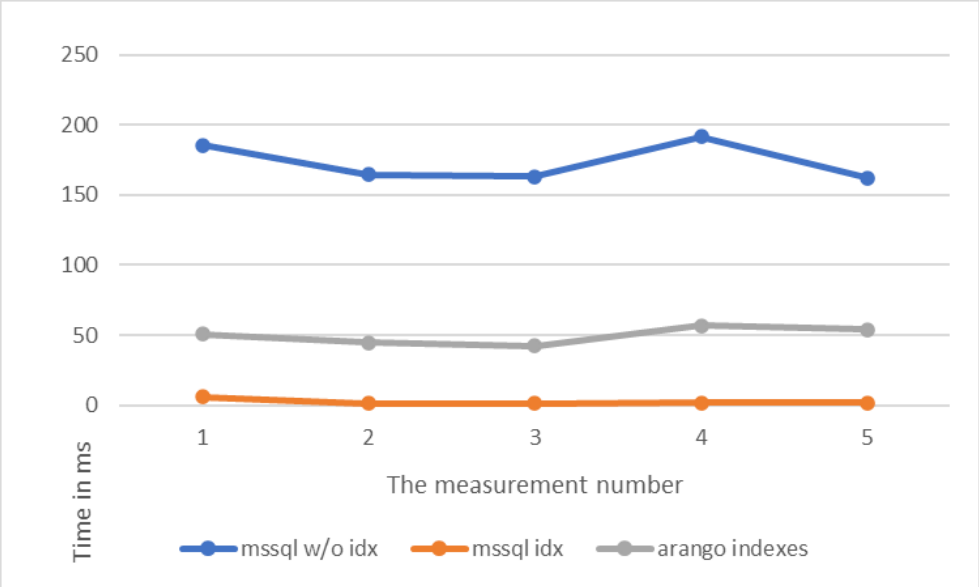

Figure 5.8: The measurements of getting a particular column object by its attributes

Chapter 6

Conclusion

An analysis of the existing mapping model, desirable use cases and research of graph database technologies revealed that a label property graph database is the most suitable for the project MMP requirements. After the graph database research, ArangoDB had been chosen as a graph database for comparison. The main reasons were that it is an open-source multimodel database that consistently gains popularity, and the database is dynamically developed. Then was designed a general meta-model for all mapping types in a label property graph database. Based on this model were implemented data flow operations. The operations were also implemented on a relational model. The series of the tests revealed that a relational model with adequately defined indexes could be somewhat comparative with a model in a graph database. If mapping data is not strongly connected and includes hundreds of transformations between two objects or less, then a relational model can be the right solution. When it comes to thousands of transformations and more, a graph database (ArangoDB in this case) is more suitable for the scenario. Graph database though, comparing to a relational one, requires more RAM space to take advantage of using it.

The bachelor thesis aimed to answer a question if it is worth implementing a relative part of a mapping model using an existing relational mapping model or current relational database is capable of effectively performing the data flow operations. The answer is yes. My advice for the MMP project is to give ArangoDB a try. The database, even in the community version, shows better performance on the specified use cases than SQL Server. The database can be integrated as a separate service, which will continuously (for example one time per day) update its internal collections using a relational mapping model, and it will expose a data flow API.

Microsoft SQL Server in version 19 introduced graph database features. Since the project uses SQL Server version 17, the questing would be if it worth to use a newer version and implement these operations using the new graph features. As of May 2020, Microsoft have not released a library that adds `LINQ`

support of graph querying. The solution though has some benefits comparing to ArangoDB, such as the fact that the project does not have to support an external database and it still uses SQL language with some extensions.

Biblioraphy

1. IAN ROBINSON Jim Webber, Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. 2nd ed. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. ISBN 9781491930892.
2. *Neo4j*. What Is a Graph Database and Property Graph [online] [visited on 2020-05-14]. Available from: `https://neo4j.com/developer/graph-database`.
3. *Neo4j*. Graph Databases for Beginners: Native vs. Non-Native Graph Technology [online] [visited on 2020-05-14]. Available from: `https://neo4j.com/blog/native-vs-non-native-graph-technology/`.
4. *ArangoDb*. Index Free Adjacency or Hybrid Indexes for Graph Databases [online] [visited on 2020-05-14]. Available from: `https://www.arangodb.com/2016/04/index-free-adjacency-hybrid-indexes-graph-databases/`.
5. *GraphDB*. GraphDB Main Page [online] [visited on 2020-05-14]. Available from: `http://graphdb.ontotext.com/`.
6. *AllegroGraph*. AllegroGraph Main Page [online] [visited on 2020-05-14]. Available from: `https://allegrograph.com/`.
7. *Ontotext*. Got meaning? Or Why an RDF Graph Database Is Good for Making Sense of Your Data [online] [visited on 2020-05-14]. Available from: `https://www.ontotext.com/blog/rdf-graph-database-making-sense-data/`.
8. *HypergraphDB*. HypergraphDB Main Page [online] [visited on 2020-05-14]. Available from: `http://www.hypergraphdb.org/`.
9. *ArangoDB*. The Many Faces of a Native Multi-Model Database [online] [visited on 2020-05-14]. Available from: `https://www.arangodb.com/why-arangodb/multi-model/`.

11. *DB-Engines*. DB-Engines Ranking of Graph DBMS [online] [visited on 2020-05-14]. Available from: https://db-engines.com/en/ranking/graph+dbms.

12. *Apache TinkerPop*. The Gremlin Graph Traversal Machine and Language [online] [visited on 2020-05-14]. Available from: https://tinkerpop.apache.org/gremlin.html.

14. *Oracle*. What Is a Data Warehouse? [online] [visited on 2020-05-14]. Available from: https://www.oracle.com/database/what-is-a-data-warehouse/.

16. *DB-Engines*. DB-Engines Ranking of Graph DBMS [online] [visited on 2020-05-14]. Available from: https://db-engines.com/en/ranking/graph+dbms.

17. *DB-Engines*. DB-Engines Ranking [online] [visited on 2020-05-14]. Available from: https://db-engines.com/en/ranking.

18. *Neo4j*. Clustering in Neo4j [online] [visited on 2020-05-14]. Available from: https://neo4j.com/docs/operations-manual/current/clustering/introduction/.

19. *OrientDB*. OrientDB Enterprise Edition [online] [visited on 2020-05-14]. Available from: https://orientdb.com/orientdb-enterprise/.

20. *Microsoft*. CREATE INDEX (Transact-SQL) [online] [visited on 2020-05-14]. Available from: https://docs.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql?redirectedfrom=MSDN&view=sql-server-ver15.

21. *StackOverflow*. Developer Survey Results 2019 [online] [visited on 2020-05-14]. Available from: https://insights.stackoverflow.com/survey/2019.

22. *Medium*. RESTful API Best Practices and Common Pitfalls [online] [visited on 2020-05-14]. Available from: https://medium.com/@schneidenbach/restful-api-best-practices-and-common-pitfalls-7a83ba3763b5.

Used images

10. *OrientDB*. Traditional query polling approach [online] [visited on 2020-05-14]. Available from: https://orientdb.com/docs/last/images/queryPolling.png.

13. *ArangoDB*. Structure of an ArangoDB Cluster [online] [visited on 2020-05-14]. Available from: https://www.arangodb.com/docs/stable/images/cluster_topology.png.

15. *Panoply*. A Data Warehouse Architecture [online] [visited on 2020-05-14]. Available from: https://panoply.io/uploads/versions/diagram8-1---x----750-376x---.jpg.

Appendix A

Acronyms

API Application Program Interface

CRUD Create, Read, Update, Delete operations

CSV Comma-separated values

DB Database

DBMS Database Management System

DD Data Dictionary Tool

ETL Extract, Transform, Load Tools

GDBMS Graph Database Management System

GQL Graph Query Language

HTTP HyperText Transfer Protocol

JSON JavaScript Object Notation

LINQ Language Integrated Query

LPG Label Property Graph

MMP Metadata Management Platform

NoSQL Not only SQL

PDM Physical Data Model

RAM Random-access memory

RDBMS Relational Database Management System

RDF Resource Description Framework

REST Representational state transfer

SQL Structured Query Language

URI Unique Uniform Identifier

Appendix B

Contents of enclosed Micro SD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format