

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Artem Bakhtin

A tool for querying multi-model data

Department of Software Engineering

Supervisor of the bachelor thesis: Ing. Pavel Koupil, Ph.D.

Study programme: Computer Science

Study branch: Databases and Web Bc.

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date
Author's signature

I extend my sincere appreciation to my supervisor for their invaluable help and support throughout my bachelor project. To my relatives and close ones, your unwavering support and belief in me have been a constant source of strength. This achievement is a result of our collective efforts, and I am deeply grateful to each and every one of you for being the guiding lights in my journey.

Title: A tool for querying multi-model data

Author: Artem Bakhtin

Department: Department of Software Engineering

Supervisor: Ing. Pavel Koupil, Ph.D., Department of Software Engineering

Abstract: Querying over multi-model data is a challenging task even for expert users, as they typically need to master a number of query languages and be aware of the logical representation of the data.

In this thesis, we propose a graphical query language over multi-model data and implement it in the form of a prototype application. The proposed query language primarily targets less experienced users, aiming at simple querying over data with only knowledge of its structure. The work includes an attached prototype that represents the data using a categorical representation strikingly similar to a graph. We take advantage of this similarity and therefore store the data in the Neo4j graph database. For proof of concept, we translate our proposed language into Cypher and transitively query over the multi-model data stored using the categorical representation in Neo4j.

Keywords: Multi-Model Data Graphical Querying Query by Example Neo4j Novice Users

Contents

Introduction	3
1 Graph query languages	5
1.1 Cypher	5
1.1.1 Usage	5
1.1.2 Cypher syntax features	5
1.1.3 Visualization	6
1.2 SPARQL	8
1.2.1 Usage	8
1.2.2 SPARQL syntax features	8
1.2.3 Visualization	9
1.3 Gremlin	10
1.3.1 Usage	11
1.3.2 Gremlin syntax features	11
1.3.3 Visualization	13
1.4 ArangoDB	14
1.4.1 Usage	14
1.4.2 AQL syntax features	14
1.4.3 Comparison	16
1.4.4 Visualization	16
1.5 RavenDB	17
1.5.1 Usage	17
1.5.2 RQL syntax features	17
1.5.3 Visualization	19
2 Visual Query Language	21
2.1 Language Description	21
2.1.1 Basic Operations	21
2.1.2 Conditionals	22
2.1.3 Transformations	23
2.1.4 Set Operations	24
2.2 Language Limitations	25
3 Algorithms	27
3.1 Root finding algorithm	27
3.1.1 JSON to Graph convert	27
3.1.2 Finding a root node in a graph	27
3.2 Graph to instruction set	28

4	Programmer documentation	31
4.1	Intro	31
4.2	MVC	31
4.3	Front-end architecture	32
4.3.1	Model	32
4.3.2	Widgets	32
4.3.3	Edges	34
4.3.4	View	34
4.3.5	Main functions	35
4.4	Back-end architecture	37
4.4.1	Main Classes and Functions	37
4.4.2	Model	41
4.4.3	Use cases	41
5	Testing	47
5.1	Introduction	47
5.2	Unit tests	47
5.3	Integration Tests	47
5.3.1	Projection Test	48
5.3.2	Selection Test	48
5.3.3	Aggregation Test	48
5.3.4	Multiple Selection Test	49
5.3.5	Order By Test	49
5.3.6	Order By Test	50
6	User documentation	51
6.1	User Documentation	51
6.1.1	Requirements	51
6.1.2	Installation guide	51
6.1.3	Application Installation	51
6.1.4	Interface of the web editor	55
	Conclusion	59
6.2	Thesis Conclusion	59
6.3	Future Work	59
	Bibliography	61
	List of Figures	63

Introduction

Querying over data represented by a variety of logical models is a big challenge. Typically, users must actively understand multiple query languages to be able to query over data stored in different database systems (so-called polyglot persistence [1]) and then must laboriously transform and piece together the query results. An alternative to this approach is multi-model DBMS [2], which integrate multiple storage strategies within a single system and offer a unified interface. However, in this case, these are often originally single-model systems that have secondarily extended their storage strategy or adopted a new interface. Thus, the way the system is extended is also reflected in the way the data is queried, where the user often needs to be aware of the logical representation of the data and assemble the query accordingly. For example, PostgreSQL¹, originally a relational dbms and now a multi-model database supporting document models (namely JSON [3] and XML [4]), uses the SQL/JSON and SQL/XML language to query over multi-model data. If users want to query over a document model, they must use language elements that allow them to traverse the hierarchical structure of an embedded JSON document (i.e., an extension for JSON). The same applies to XML documents and SQL language extensions towards querying over XML data. However, you can't use traditional SQL to query over JSON data directly just as you can't use extensions for JSON or XML to query over relational data. In other words, users really need to be aware of the logical representation of the data and not just the structure, as is common with single-model data.

Moreover, today we are talking about Industrial Revolution 4.0, where potentially large numbers of employees will be changing employers in the coming years. And as the interest in working with data has been growing recently, a large number of them will be heading to this field. The assumption that people will be trained to work with multi-model data and actively use multiple query languages is misguided. This task is often challenging even for today's expert users of database systems. And it is also costly, since in practice we must have a database expert for each system.

Thus, we need to create a unified query language over multi-model data, where users will only work with a single data schema. Moreover, in order to make querying over the data suitable for novice users, we will ideally also need a graphical and simple representation of the queries (as inspired by [5]). The area of unified representation of multi-model data is addressed, for example, in [6, 7], and for querying over such data, the MM-quecat² tool has been implemented to support querying over data using the MMQL language [8, 9]. However, this is a text-only language that is based on SPARQL [10], thus expert users are targeted.

¹<https://www.postgresql.org>

²<https://www.ksi.mff.cuni.cz/~koupil/mm-quecat/index.html>

In this thesis, we focus on graphical querying over multi-model data and propose a prototype of a simple query language and tool to query such data. In the actual design of the graphical language, we take advantage of the design properties of the unified representation of multi-model data [6], which represents data as a small category [11] that structurally corresponds to a multigraph. The individual goals of the bachelor thesis can be summarized as follows:

- *Analysis of graph query languages* We will analyze selected graph query languages and their graphical visualization.
- *Graphical query language* We will propose the basics of a graphical query language (VQL), taking inspiration from existing graphical query languages.
- *Graphical query tool* We design and implement a tool for graphical querying over multi-model data.

Outline In Chapter 1, we survey popular graph query languages and their graphical representations (if any exist). In Chapter 2, we use examples to illustrate the design of a proposed graphical query language VQL over a unified multi-model representation. In Chapter 3, we describe the basic algorithms for translating VQL into the chosen graph query language. In Chapter 4, we provide programming documentation for the querying tool over the data. In Chapter 5, we describe test scenarios that verify the expressive power of the VQL language. In Chapter 6, we provide user documentation. Finally, we conclude and outline future work.

Chapter 1

Graph query languages

1.1 Cypher

Cypher [12] is the Neo4j's¹ declarative query language for graphs. Its language syntax consists of clauses and functions that are familiar to developers experienced with SQL.

1.1.1 Usage

Cypher is specifically designed for retrieving and processing data from graph databases. Language is tightly integrated with Neo4j DBMS and cannot be evaluated independently of this database management system. Neo4j has various applications in different technology sectors. For instance, the language is widely used for fraud detection. The Cypher language has significant advantage over relation databases, because of its effective mechanisms for crime detection. Additionally, Neo4j is well-suited for applications in social networks and real-time recommendations, as it one of the best at connecting individuals and their interests. Furthermore, Neo4j is employed in various scenarios where efficient data management and mastery of data management systems are crucial.

1.1.2 Cypher syntax features

Cypher offers a unique approach to matching patterns and relationships in graph databases. It utilizes ASCII symbols, such as dashes and arrows, to represent the connections between nodes. By using round brackets, users can define nodes and their relationships, while square brackets allow for the specification of relationship type and setting additional properties. Language supports default functions such as sorting, filtering, aggregation and etc. Cypher is schema-less, meaning it does not support Data Definition Language (DDL) for defining schemas. However, it fully supports Data Manipulation Language (DML) operations, enabling users to create, delete, and modify data. The Merge operation is particularly useful for performing "match or create" operations in Cypher. Another important feature is database administration through Data Control Language (DCL). Cypher provides a set of commands, including GRANT, DENY, and REVOKE, to manage administrative rights. It's worth mention that these features are only available

¹<https://neo4j.com>

in the Enterprise Edition. Querying in Cypher resembles SQL queries, but with specific clauses like `OPTIONAL MATCH`. Additionally, in implicit mode, users can work with rollbacks and commits to ensure data integrity.

1.1.3 Visualization

Neo4j

The visualization of the language is built upon the Neo4j DBMS, a scalable graph database known for its schema-free data model. Neo4j holds a dominant position in the market as a graph database manager. It is the only DBMS that offers visualization for the Cypher language, making it the obvious choice for representing the language visually. With Neo4j, users can query data from both desktop and web applications. The database excels at visualizing a wide range of user queries. For instance, when working with a test database, simple queries like retrieving a list of players or players with specific properties, as well as listing all clubs or clubs with specific properties, are represented as individual floating nodes in the visual output (refer to Figure 1.1). If there are relationships between the resulting nodes, these connections will also be displayed in the resulting graph.

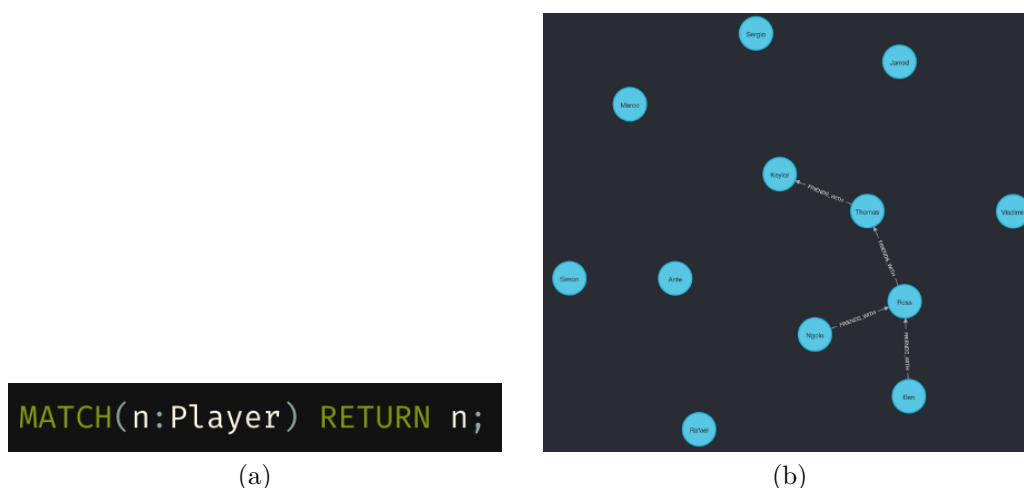


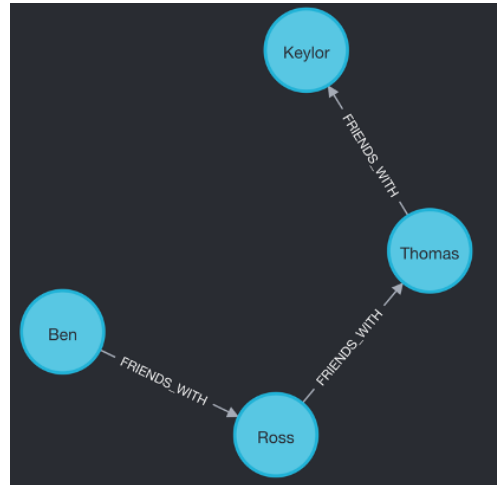
Figure 1.1: Query that lists all players (a) and the resulting visualization of the query (b)

The application also includes a drag-and-drop functionality for the floating nodes, although its practical use is limited. Cypher, on the other hand, offers the capability to perform complex graph traversals. For instance, with the second query, users can find all the friends of friends for a given player (refer to Figure 1.2). The resulting graph is visualized as an oriented connected graph, showcasing four nodes in total. There is possibility set the depth of the traversal, such option is a unique feature for graph database.

Neo4j is able to visualize any graph representation, until user keep nodes integrity. However, it does not support visualization of aggregations. The results of aggregations can only be viewed in table representation. Additionally, Neo4j does not support the visualization of node properties or null data. Null values, which often occur as a result of the query with `OPTIONAL MATCH` clause, have

```
MATCH
(p:Player {surname: 'Konte'})-[:FRIENDS_WITH *]-(:a:Player)
RETURN a;
```

(a)



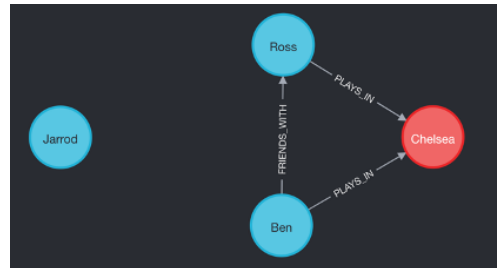
(b)

Figure 1.2: Query that finds player friends of friends (a) and the resulting visualization of the query (b)

no representation in the resulting graph. An example of such a situation is show on Figure 1.3.

```
MATCH(p:Player {nation:"England"})
OPTIONAL MATCH(p)-[:PLAYS_IN]-(:t:Team {name: 'Chelsea'})
RETURN p,t;
```

(a)



(b)

"p"	"t"
{ "nation": "England", "surname": "Chilwell", "name": "Ben", "age": 25, "height": 180 }	{ "country": "England", "size": 27, "name": "Chelsea", "foreigners": 19 }
{ "nation": "England", "surname": "Barkley", "name": "Ross", "age": 28, "height": 185 }	{ "country": "England", "size": 27, "name": "Chelsea", "foreigners": 19 }
{ "nation": "England", "surname": "Bowen", "name": "Jarrod", "age": 25, "height": 182 }	null

(c)

Figure 1.3: Query that finds English players in Chelsea club (a), result of query as graph (b) and as a table (c)

On the other hand, visualization of nodes group is possible. Using UNWIND clause user is allowed to work with lists. Grouping players of two teams in one result in graph on Figure 1.4.

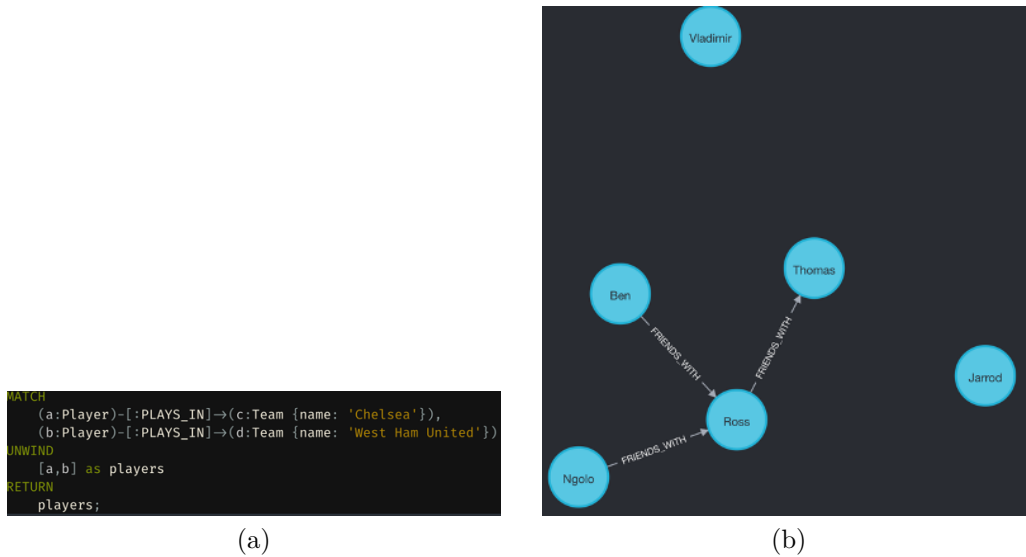


Figure 1.4: Grouping players in one team (a) and result of the query (b)

1.2 SPARQL

SPARQL [10] is a semantic query language, which is developed for processing data in RDF² (Resource Description Framework) format. Unlike SQL, SPARQL can be used to query both relational databases and NoSQL graph databases. One of the key differences between SPARQL and SQL is that SPARQL queries can be executed on multiple data stores, allowing for accessing and merging linked data. Queries can access multiply data stores. Such feature is not accidental. This feature helps achieve main goal of the language – to share and merge linked data.

1.2.1 Usage

SPARQL language is used for analytics in graph databases. SPARQL extends basic SQL analytics by graph examination of relationships. With SPARQL, it becomes possible to perform operations such as friend-of-friend relationship analysis, finding the shortest path between nodes, and calculating PageRank scores. These SPARQL features are actively used in Fraud Detection, Money Laundering Detection, Social Network and Recommendation Engines. While the first three areas are well-known from the Cypher language, the use of SPARQL in recommendation systems is a newer application in the realm of graph databases. Both SPARQL and Cypher serve similar purposes with some minor differences in their respective fields of usage.

1.2.2 SPARQL syntax features

SPARQL does not have clauses such as INSERT, DELETE, or UPDATE, which means that it cannot directly modify the graph by adding or removing information. However, it does provide a specific syntax for graph traversal. Using

²<https://www.w3.org/RDF/>

OPTIONAL clause user can include necessary information or receive unknown response. This is made possible through the use of three-valued logic. It's important to note that SPARQL does not support operations related to the database schema, even though it can be used to work with relational databases.

1.2.3 Visualization

Sparql-vizualizer

Most web applications associated with the SPARQL language do not provide visualization capabilities. Instead, the query results are typically presented in a tabular format. In my demonstration of the query results, I have used a sample database. The SELECT clause is the key component for data projection in SPARQL, similar to its counterpart in SQL. However, the WHERE clause is different in SPARQL, as it consists of RDF triples for specifying the selection criteria (refer to Figure 1.5).



Figure 1.5: SPARQL query listing all buses (a) and its result (b)

There is another way, how user can apply selection, using FILTER clause (see Figure 1.6).

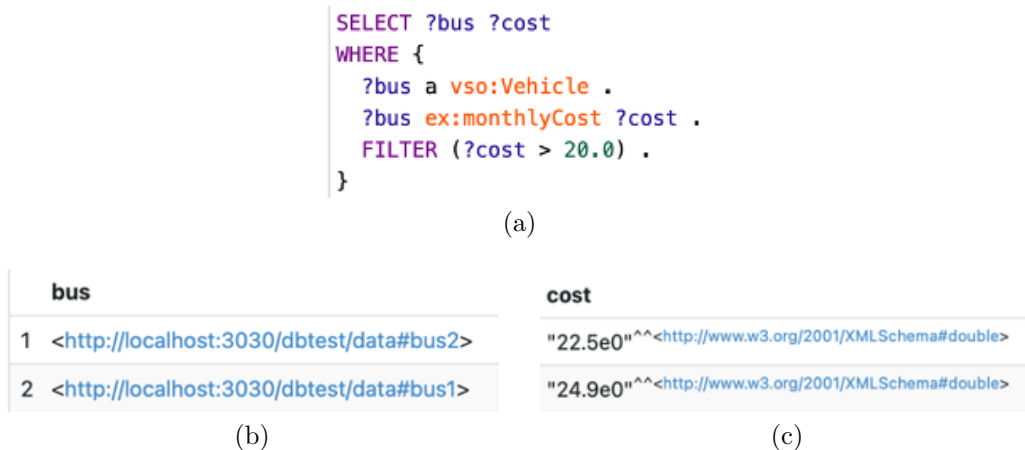


Figure 1.6: SPARQL query that list all buses (a) and the resulting visualization (b, c)

SPARQL also supports grouping and filtering grouped items (see Figure 1.7).

Also, UNION clause is supported for table view. Table can hold almost all query results. The unusual behavior can be found in query with OPTION clause (see Figure 1.8).

The query shown above may result in missing values in the output. In such cases, the corresponding cells are left empty without a specific value. While

```
SELECT ?pass ?ticketName
WHERE {
  ?pass a ex:Passenger .
  ?pass ex:hasTicket ?ticket .
  ?ticket gr:name ?ticketName
} GROUP BY ?pass ?ticketName
```

(a)

```
SELECT ?pass ?ticketName
WHERE {
  ?pass a ex:Passenger .
  ?pass ex:hasTicket ?ticket .
  ?ticket gr:name ?ticketName
}
GROUP BY ?pass ?ticketName
HAVING (substr(?ticketName, 1) = "1")
```

(b)

pass	ticketName
1 <http://localhost:3030/dbtest/data#passenger1>	"123456789"@en
2 <http://localhost:3030/dbtest/data#passenger2>	"123459876"@en
3 <http://localhost:3030/dbtest/data#passenger2>	"987654321"@en

(c)

Figure 1.7: Grouping and filtering grouped items (a,b) and the results (c,d)

```
SELECT ?bus ?cost
WHERE {
  ?bus a vso:Vehicle .
  OPTIONAL {?bus ex:monthlyCost ?cost } .
}
```

Figure 1.8: Optional clause

exploring SPARQL visualization solutions, I found only one public project (as depicted in Figure 1.9). The project is not commercial and was made for educational purpose. Project is publicly available on the web. Users can download the source code and deploy the project on their own machines. The tool allows users to define their own databases and query the data. The results of each query are displayed in a dedicated area within the web application. The resulting graph includes the complete query path next to each vertex, where each vertex represents an RDF object or subject. The connections between vertices represent RDF triples, and the edges correspond to predicates. Each edge is presented by its full URI, which can be shortened using predefined prefixes.

Unfortunately, the application is currently experiencing issues with displaying its web page components. Despite this bugs, the tool shows great promise and potential.

1.3 Gremlin

Gremlin [13] is a query language specifically designed for traversing and querying graph databases. It is a graph traversal language that provides a uniform way to interact with diverse graph databases, regardless of their underlying storage structure Gremlin traversal can be written either in declarative or imperative way. Also, the language works for both OLTP and OLAP based graph databases. Gremlin language is based on Groovy, but it is supported by mostly mainstream

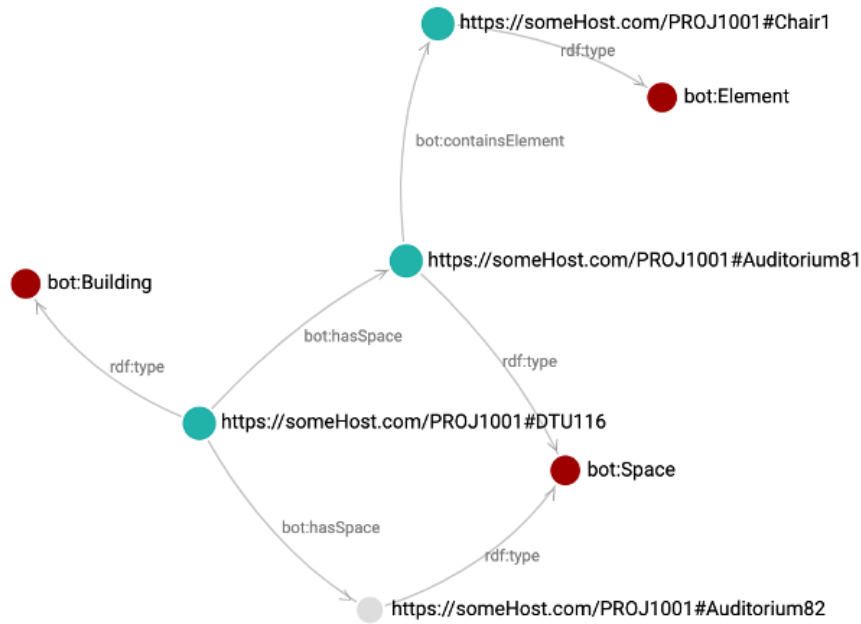


Figure 1.9: Graph visualization

programming languages.

1.3.1 Usage

Gremlin is primarily used for complicated graph traversals. The language provides a wide range of graph operations, such as calculating distances, performing joins, finding paths, and executing graph analytics algorithms.

1.3.2 Gremlin syntax features

I utilize the Apache TinkerPop console to interact with the Gremlin server. Apache TinkerPop is a graph library and system contributed by a third party. By using the symbols ":>" followed by the actual command written in the Groovy language, users can send commands to the Gremlin server. To create a new graph instance, users can use the command illustrated on Figure 1.10.

```
gremlin> graph = TinkerGraph.open()
```

Figure 1.10: Initializing new graph

Next thing that should have be done is providing TraversalSource to new created graph. I personally use predefined embedded traversal engine – graph (see Figure 1.11).

```
gremlin> g = traversal().withEmbedded(graph)
```

Figure 1.11: Initializing new traversal

The new 'g' graph is available for traversal. Before we proceed with the traversal, some edges and vertices should be added to the graph. The language

also provides support for various graph operations, including remove, update, delete, and insert. The user is able to create new vertex using `addV()` method. To remove the vertex or edge exists `drop()` function can be used. To add a new vertex, we first specify its type and then add the properties one by one. In this example, I will use the same graph as shown in the Cypher example (refer to Figure 1.12).

```
gremlin> v1 = g.addV("Player").property(id, 1).property("name", "Ngolo").property("nation", "France").property("age", 38).next()
==>v[1]
gremlin> v2 = g.addV("Player").property(id, 2).property("name", "Ross").property("nation", "England").property("age", 28).next()
==>v[2]
gremlin> v3 = g.addV("Player").property(id, 3).property("name", "Simon").property("nation", "Denmark").property("age", 32).next()
==>v[3]
gremlin> v4 = g.addV("Team").property(id, 4).property("name", "Chelsea").property("country", "England").next()
==>v[4]
gremlin> v5 = g.addV("Team").property(id, 5).property("name", "Milan").property("country", "Italy").next()
==>v[5]
```

Figure 1.12: Adding vertices

Edges can be added using `addE()` method. Let's add `friends_with` and `plays_in` edges same as in cypher graph. Every `plays_in` edge has contract expiration time and `friends_with` edge has duration of friendship indicated (see Figure 1.13).

```
gremlin> g.addE("plays_in").from(v1).to(v4).property(id, 6).property("contract_expire", 2.5)
==>e[6][1-plays_in->4]
gremlin> g.addE("plays_in").from(v2).to(v4).property(id, 7).property("contract_expire", 4.7)
==>e[7][2-plays_in->4]
gremlin> g.addE("plays_in").from(v3).to(v5).property(id, 8).property("contract_expire", 1.3)
==>e[8][3-plays_in->5]
gremlin> g.addE("friends_with").from(v1).to(v2).property(id, 9).property("for", 10.0)
```

Figure 1.13: Adding edges

Next, gremlin support aggregation functions such as `count` (see Figure 1.14).

```
gremlin> g.V().count()
==>5
gremlin> g.E().count()
==>4
```

Figure 1.14: Aggregate function `count()`

Gremlin allows user to query the graph in more succinct way than over graph languages (see Figure 1.15).

```
gremlin> g.V().has('Player', 'name', 'Ngolo').outE("plays_in")
==>e[6][1-plays_in->4]
```

Figure 1.15: Gremlin query

Command finds all edges `plays_in` connected with player named Ngolo. In other words, Gremlin traversals graph using only `plays_in` edges (see Figure 1.16).

```
gremlin> g.E().has("contract_expire", gt(2.0))
==>e[6][1-plays_in->4]
==>e[7][2-plays_in->4]
```

Figure 1.16: Query: Find all edges with contract expiring more than in two years

Additionally, the language enables users to perform depth-first traversal and breadth-first traversal with ease. Gremlin is naturally designed for depth-first traversal. If the graph structure is known, performing a depth-first traversal can be achieved straightforwardly (refer to Figure 1.17). In the given graph, the vertices to visit in a depth-first traversal are simply the first and fourth vertices.

```
gremlin> g.V(1).outE('plays_in').inV()
==>v[4]
```

Figure 1.17: Depth-first traversal

1.3.3 Visualization

Gephi

The Gephi³ desktop application has been chosen as graph visualizer for Gremlin graph. Gephi is interactive visualizer tool. Using Streaming plugin user can connect to Gephi via gremlin console (see Figure 1.18).

```
gremlin -remote connect link:tcp:gephi
NoConnection to Gephi - http://localhost:8080/workspace1 with stepDelay:1000, startNodeColor:[0.0, 1.0, 0.5], colorFadeg, colorFadegRate:0.7, startSize:10
g.sizeOfElements:0.33
gremlin -> graph
NoLinkGraph(vertices:5 edges:4)
NoFile
```

Figure 1.18: Gephi initialization

Once the connection successfully establish “:> graph” command sends the graph to the visualizer tool (see Figure 1.19). Gephi allows editing graph and executing traversal. Such feature is a must have for analyzing the graph. It also offers built-in features that are highly valuable, including finding the shortest path, generating heat maps, and performing weighted traversals.

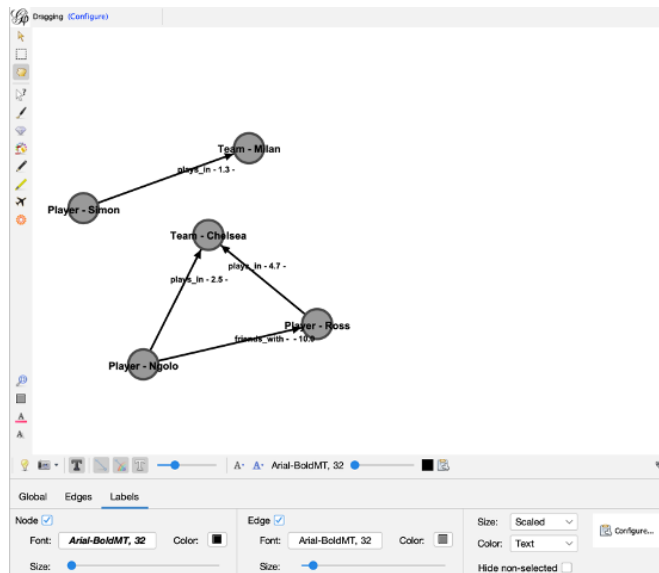


Figure 1.19: Gephi interface

Another great feature is the option to choose from the various layouts and additionally the user can adjust the graph visualization settings according to his

³<https://gephi.org>

needs. For instance, the tool enables users to adjust the size of labels for different elements in the graph and choose which graph properties to display. Gephi is a powerful tool, but it has a somewhat complex and outdated interface.

1.4 ArangoDB

ArangoDB⁴ is a NOSQL mutli-model graph DBMS. It provides native support for graphs, JSON and key/value data format models. ArnagoDB has its own SQL like language for querying called AQL (ArangoDB Query Language) [14].

1.4.1 Usage

ArangoDB, like many other graph databases, finds applications in fraud detection, network infrastructure operations, social network analytics and management, recommendation engines, and more. However, what sets ArangoDB apart from many other NoSQL databases is its ability to combine different data models within a single database and execute queries that involve multiple data models simultaneously. This feature provides users with a wide range of graph processing capabilities, including graph traversal, shortest path finding, pattern matching, and more.

1.4.2 AQL syntax features

ArangoDB offers multiple ways to access its functionality, including a web interface, the terminal-based tool "arangosh," and various drivers. It's important to note that AQL (ArangoDB Query Language) is not a Data Definition Language (DDL), so it cannot be used to modify database schemas or structures. Additionally, AQL does not support user creation or permission settings, making it unsuitable as a Data Control Language (DCL). Because of its limitations in making structural changes, AQL cannot be used to create document collections. In my case, I used the web interface to interact with the database, where collections can be created to store either documents or edges (as shown in Figure 1.20).

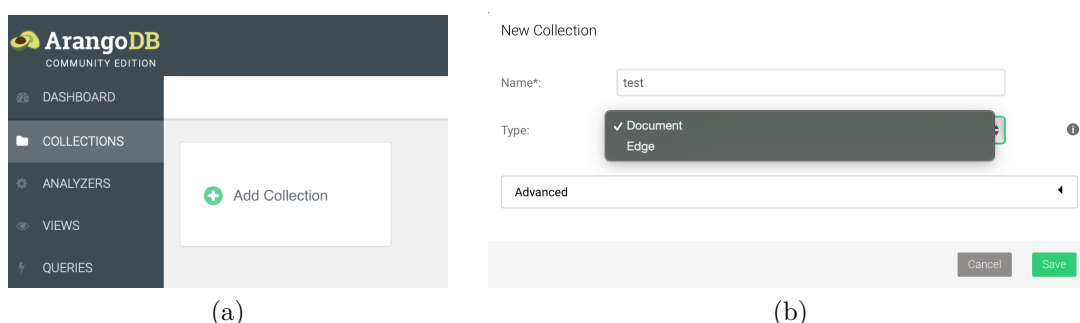


Figure 1.20: Creation new collection

To create a new graph instance in ArangoDB, the user is required to have a document collection and an edge collection that connects the documents. In

⁴<https://www.arangodb.com>

my case, I chose to implement a simple graph representing friends. Data manipulation in the collection can be performed using operations such as INSERT, UPDATE, REPLACE, REMOVE, and UPSERT (an example is illustrated in Figure 1.21). Data can be inserted in collection by iterating over array of JSON objects. Document object is an array of key/value attributes, where attributes are separated with ”:”.

```

1 //INSERT DOCUMENTS
2 let data = [
3   {"_key": "1", "name": "Evelyn", "surname": "Scott", "age": 30},
4   {"_key": "2", "name": "Keith", "surname": "Nguyen", "age": 21},
5   {"_key": "3", "name": "Juan", "surname": "Brown", "age": 43},
6   {"_key": "4", "name": "Alex", "surname": "Olson", "age": 22},
7   {"_key": "5", "name": "Nick", "surname": "Ross", "age": 20}
8 ]
9 FOR d IN data
10  INSERT d INTO friends
11 //UPDATE
12 let edges = [
13   {"_from": "friends/1", "_to": "friends/2"},
14   {"_from": "friends/1", "_to": "friends/3"},
15   {"_from": "friends/3", "_to": "friends/4"}
16 ]
17 FOR d IN edges
18  INSERT d INTO friendsWith
19 UPDATE "5" WITH {"age": 25} IN friends
20 //REMOVE "5"
21 REMOVE "5" IN friends
22 //READ
23 FOR f IN friends
24  RETURN f

```

Figure 1.21: Crud operations

To create documents in ArangoDB, you can use two data types: string and number. Additionally, ArangoDB supports data types such as null, boolean, array, and object. Edges are created using the ”_from” and ”_to” attributes, allowing for the creation of directed edges. In AQL, data projection can be achieved using the RETURN operator. Data returns as array of JSON object, but its possible to additionally wrap the data into array or into object. Return data can be filtered, sorted and limited. Another feature is that documents collection can be merged together by mapping attributes (see Figure 1.22).

For such purpose user should iterate over collection and merge individual attributes. ArangoDB also provides a bunch of build-in functions to process dates, objects, arrays, string, geo data and etc. Graph traversing can be done by iterating over documents and setting INBOUND, OUTBOUND or ANY operating with appropriate traversing depth. In my case, I can complete INBOUND traversal to depth 1..1, in other words I get only friends of person with 1 id. In the first query, you will receive the names Keith and Juan. By changing the depth to 1..2, the resulting array will include the names Keith, Juan, and Alex, as the friend of a friend is added to the return array (refer to Figure 1.23).

```

1 //FILTER PROJECTION
2 FOR f IN friends
3   FILTER f.age >= 30
4   FILTER f.age <= 50
5   RETURN f.name
6 //LIMIT PROJECTION
7 FOR f IN friends
8   LIMIT 2
9   RETURN f.name
10 //SORTING PROJECTION
11 FOR f IN friends
12   SORT f.name
13   RETURN f.name

```

Figure 1.22: Data Projection

```

1 FOR friend IN 1..1 OUTBOUND
2   "friends/1" friendsWith
3   RETURN friend.name

```

(a)

```

1 FOR friend IN 1..2 OUTBOUND
2   "friends/1" friendsWith
3   RETURN friend.name

```

(b)

Figure 1.23: Search friends (a) and Search additionally friends of friends (b)

1.4.3 Comparison

AQL (ArangoDB Query Language) shares many similarities with SQL. However, there are some differences in the operators used, such as WHERE in SQL being equivalent to FILTER in ArangoDB, SORT corresponding to ORDER BY, and COLLECT to GROUP BY, among others. Additionally, SQL is an DDL and DCL meanwhile AQL is not. ArangoDB allows user to define its own schema, but schemas are not mandatory unlike SQL. But the key difference is in data models. ArangoDB is document-oriented meanwhile SQL is a relation database. But it possible to say that collection correspond to tables in relation model as well as for loops roughly corresponds to SELECT statements in SQL. AQL is less power language then SQL, but it has unique ability to perform multi-model query. Language allows to combine joins and graph traversals in one query and easily switch between data-models.

1.4.4 Visualization

The web interface allows users to have an overview of the created graph (refer to Figure 1.24). There are plenty of graph configuration options. The user can choose displaying labels, graph coloring, layout type, arrows and lines type, limit vertex displaying and etc. Web interface also provides possibilities to edit graph inside visualisation. User is able to create new edge/vertex and edit existing one. Web Interface has mostly visualisation tool, but there is also tool for starting traversal from chosen node and user also can configure depth of this traversal. Result of such traversal appears immediately on screen. User is able to perform some sort of traversing using visualisation tool. Web interface does not support AQL query visualisation or building visual queries. Results of all query returns

as array of JSON objects.

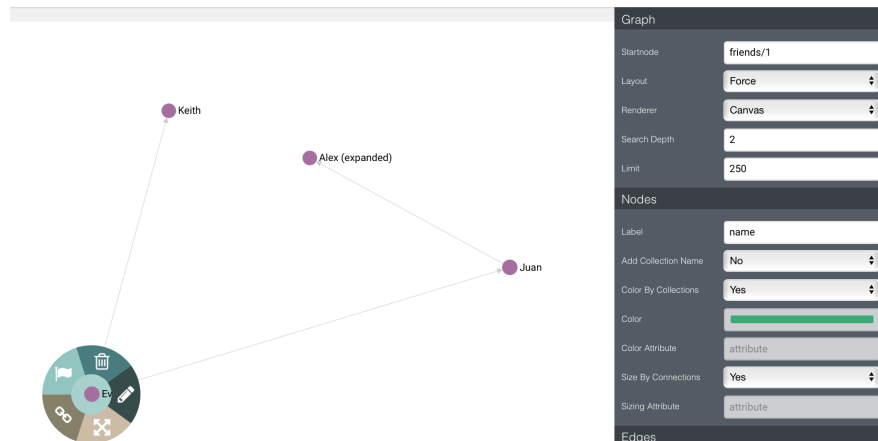


Figure 1.24: Graph visualisation

1.5 RavenDB

RavenDB⁵ is a NoSQL document-based database that stores data in the JSON format. However, it also provides built-in support for other data formats such as XML and binary. The database can be deployed on a single node or a cluster of nodes, offering scalability and flexibility. Each node in the cluster supports master-master replication, allowing multiple nodes to host data. Unlike many other NoSQL databases, RavenDB is an ACID database.

1.5.1 Usage

Database uses for standard NoSQL cases such as fraud detection and identity authentication, data management, IoT, sensor data and etc. Due to high performance scalability, RavenDB is often used for effective scaling. One worth mentioning feature of RavenDB is its ACID (Atomicity, Consistency, Isolation, Durability) compliance. This means that the database guarantees transactional integrity and reliability, ensuring that operations are executed reliably and consistently. RavenDB provides a robust and versatile solution for managing and storing data in various formats, with support for clustering and ACID compliance.

1.5.2 RQL syntax features

RavenDB is an open-source database written in C#. Meanwhile, the database is cross-platform and it is available for Linux/MAC OS/Windows systems. The Docker image is also available for Windows and Linux. I personally utilized Docker to launch a RavenDB server on my local machine. The DBMS allows users to manage the database through a GUI interface that can be accessed by browser using server URL (see Figure 1.25).

RavenDB's management studio provides a robust toolkit for efficiently managing databases and servers. RavenDB has its own specific language for querying a

⁵<https://ravendb.net>

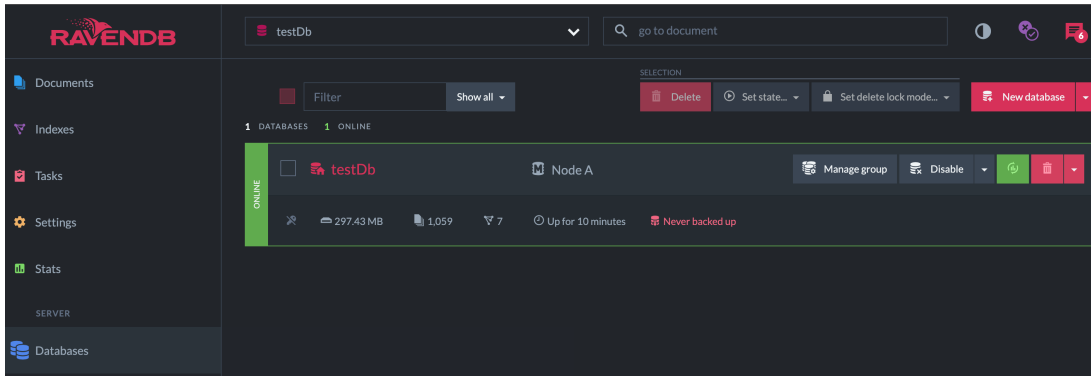


Figure 1.25: GUI Interface

data. RQL [15] is designed to efficiently execute query within RavenDB pipeline. RQL in only DQL language with some features of DML language. While RQL allows users to update existing data, it does not support the insertion of new data. Two types of queries can be executed: dynamic queries and indexed queries. In an indexed query, users specify the index to be utilized by the database, leading to enhanced performance. The management studio also provides users with sample data, which I leveraged for testing purposes.

RQL syntax is very similar to SQL. RQL has such common methods as SELECT, FROM, WHERE, GROUP BY, UPDATE and WITH (an example is illustrated in Figure 1.26). They are almost identical to appropriate SQL methods. But there is no crucial method such as join. Technically RavenDB provides user with INCLUDE method, but this method only fetch required document. Also the database doesn't support foreign keys. Foreign keys in RavenDB representation is just a string with other document id. One of the unique RavenDB methods are LOAD and DECLARE.

```

1 // quering data with index and javascript select
2 from index 'Orders/Totals' as i
3 where i.Total > 10000
4 load i.Company as c
5 select {
6   Name: c.Name,
7   Region: c.Address.Region,
8   OrderedAt: i.OrderedAt
9 }
10 // quering with dynamic query, using search function
11 from 'Products'
12 where search(Name, 'Louisiana')
13 order by NameForSorting desc
  
```

Figure 1.26: Query examples

The LOAD operation in RavenDB allows for the loading of data from external documents. On the other hand, the DECLARE method is utilized to create JavaScript functions, which can be employed for data filtering and processing within the UPDATE operation. Worth mention the fact that RavenDB SELECT methods supports Javascript selects. RQL (Raven Query Language) incorporates a variety of built-in functions to facilitate query operations. These functions cover

a range of functionalities, including text manipulation, aggregation, and mathematical operations. In addition to these common functions, RavenDB offers spatial functions that are not commonly found in other databases. Moreover, RavenDB introduces a unique function called MoreLikeThis, which retrieves documents similar to a given document (as demonstrated in Figure 1.27).

```

1 // results in 28 location from 597 available
2 from index "Orders/ByShipment/Location"
3 where spatial.within(ShipmentLocation, spatial.circle(4500, 0, 0))

```

Figure 1.27: Spatial Query Example

1.5.3 Visualization

Management studio provides user with numerous statistical charts and diagrams. But unfortunately, GUI does not provide user with visual query language or visualisation of query results. Every query output is displayed in table format (see Figure 1.28).

Preview	Id	Company	Employee	Freight	Lines	OrderedAt	RequireAt
	orders/34-A	companies/69-A	employees/4-A	2.94	[...]³	1996-08-14T00:00:00...	1996-08-28T00:00:00...
	orders/35-A	companies/69-A	employees/4-A	12.69	[...]²	1996-08-15T00:00:00...	1996-09-12T00:00:00...
	orders/56-A	companies/30-A	employees/7-A	107.83	[...]³	1996-09-11T00:00:00...	1996-10-09T00:00:00...
	orders/59-A	companies/69-A	employees/1-A	7.56	[...]³	1996-09-16T00:00:00...	1996-10-14T00:00:00...
	orders/81-A	companies/28-A	employees/4-A	87.03	[...]³	1996-10-14T00:00:00...	1996-11-11T00:00:00...
	orders/89-A	companies/60-A	employees/7-A	15.51	[...]¹	1996-10-23T00:00:00...	1996-11-20T00:00:00...
	orders/105-A	companies/28-A	employees/3-A	1.3	[...]²	1996-11-12T00:00:00...	1996-11-26T00:00:00...
	orders/150-A	companies/60-A	employees/5-A	60.26	[...]²	1996-12-27T00:00:00...	1997-01-24T00:00:00...
	orders/186-A	companies/60-A	employees/3-A	73.83	[...]¹	1997-02-03T00:00:00...	1997-03-03T00:00:00...

Figure 1.28: Query Result

User is able to filter output, export and display some statistic about query. Also user can obtain visual statistic about each index performance. Another visualization feature is map-reduce visualizer. Visualizer allows view internal structure of map-reduce index by displaying relationships between the documents and the result (see Figure 1.29).

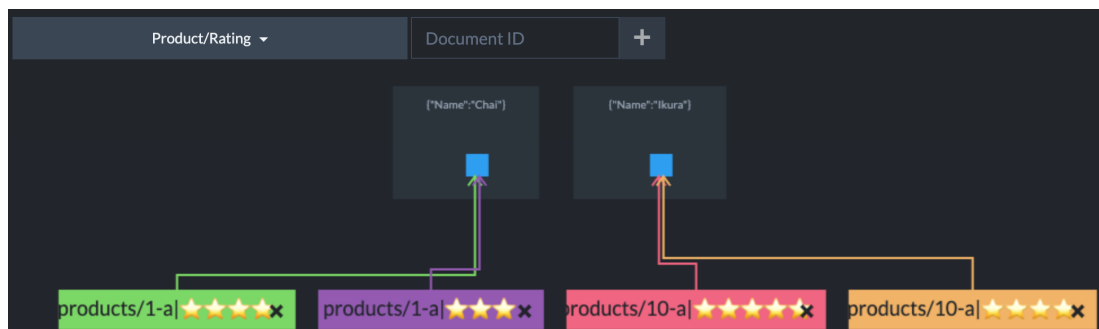


Figure 1.29: Map-reduce relationship

Chapter 2

Visual Query Language

2.1 Language Description

2.1.1 Basic Operations

The design and implementation of a visual query language is one of the core components of the tool. The Visual Query Language (referred to as VQL below) has to fulfill some crucial conditionals. It should be user friendly and intuitive, allowing persons with no programming background to easily orient themselves within it.

Each query in VQL is a directed connected graph. The graph consists of nodes (entities) interconnected with edges. The nodes are represented as rectangles of various colors. Different colors help a user distinguish between different node types. The first and the most important node type is basic vertex, which serves as the foundation for each query and represents entities in graph, such as players, cars, customers and etc. The label of the entity is written inside the rectangle.



Figure 2.1: Non-projection basic vertex

The basic vertex itself can be projection and non-projection. The non-projection basic vertex is represented as an orange rectangle, while the projection basic vertex is represented as a grey rectangle. The VQL does not support the projection of whole entity, so only non-projection basic vertices can be used to represent entities. However, both projection and non-projection basic vertices can be used to represent the properties of an entity. Non-projection basic vertex can be useful in situations where a user wants to set conditionals on a property, but does not want to return the property itself.

To create a relation between two nodes in VQL, an edge can be used. There are various types of edges, but the basic edge is the most commonly used by users to interconnect nodes. Other types of edges are parts of constructions such as transformation or set operation and are automatically included within these



Figure 2.2: Projection basic vertex

structures. Therefore, users do not need to add them manually. To project any property of an entity, the user should choose the entity node as the parent and the property node as the children.

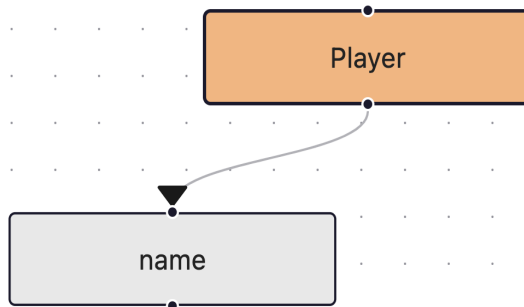


Figure 2.3: Simple relation

2.1.2 Conditionals

Entities in VQL may have multiple properties, which can be represented as either non-projection or projection basic vertices. Both of the types can have a conditional. The conditionals accepts strings and can distinguish between numeric and text clauses. Text conditionals comparison is based on simple string equality, while numeric clauses support equality, greater than and lower than operations.

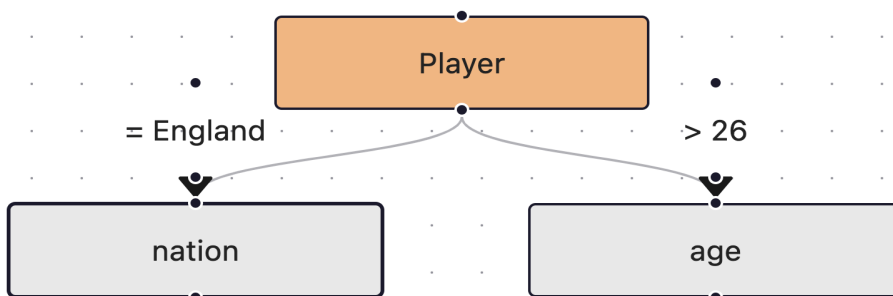


Figure 2.4: Simple condition

Two methods are available for representing conditionals in VQL. The first one works for both text and numeric clauses. User can to set only one simple (without AND and OR operators) condition for each property. Figure 3.4 provides an example of such conditionals. The second method is specific to text conditionals. User can combine multiple conditions into single one using OR operator.

Conditions can be also set for order by structure, but it is described in next chapters.

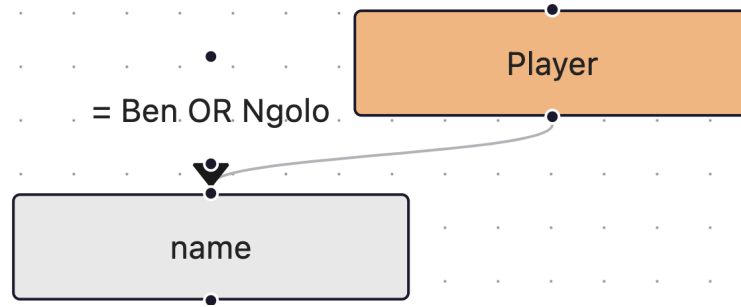


Figure 2.5: Alternative text condition

2.1.3 Transformations

Visual Query Language supports transformation operations such as group by, order by, skip and limit. These operations requires parent to be connected with.

Group by operation is a structure, which includes a blank vertex with group by count label, a group by edge and a parent vertex. The group by edge holds the property name on which group by is executed.

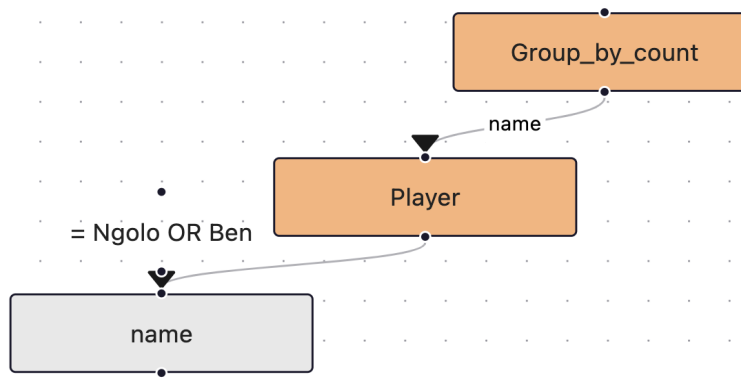


Figure 2.6: Group by

It is important to note that the term "count" is used to specify group by operation type due to its diverse representation in Cypher language. Since there is no direct equivalent of group by operation in Cypher, aggregation functions such as max, min, and count can be used to emulate the standard behavior of group by operation. The count type is chose in VQL to represent group by operation due to its similarity with the default behavior.

Having conditionals is also supported by the VQL. When creating a new group by structure, users can specify the having conditional. Since there is no direct representation of group by operation, having conditional is represented as a where clause.

The next transformation operation in VQL is order by operation. The order by structure consists of a blank vertex, an order by edge, and a parent vertex. The order by edge holds the property name on which the ordering is executed. Unlike the group by operation, it has directed equivalent in Cypher language. VQL supports both ascending and descending types of order by, which are represented by an orange rectangle with the "order by ASC" or "order by DESC" label.

The two remaining operations in VQL, SKIP and LIMIT, share a similar structure. Both operations consist of a blank vertex, a skip or limit edge, and a

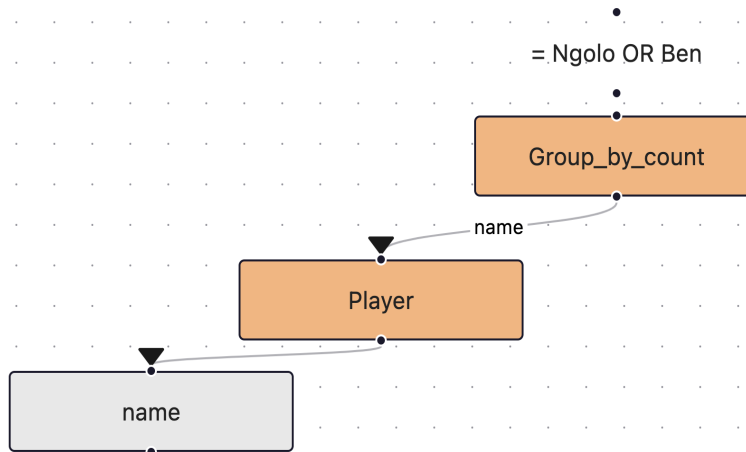


Figure 2.7: Group by with having conditionals

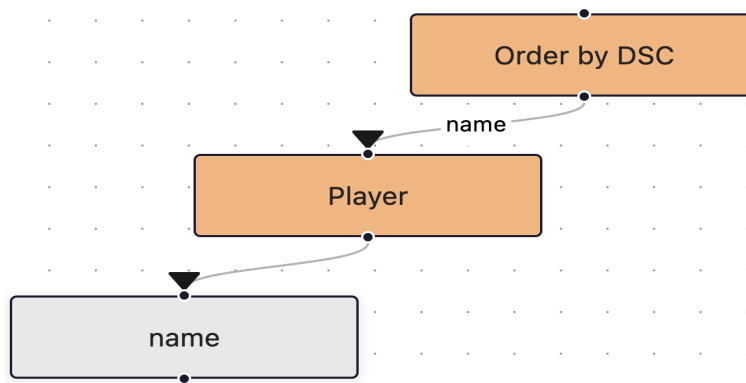


Figure 2.8: Group by with having conditionals

parent vertex. The skip or limit edge holds the number of skips or limits, respectively. These operations have a direct equivalent in Cypher language, making them easily translatable.

2.1.4 Set Operations

The VQL provides support for the union set operation, which is the only set operation that is supported due to the complexity of translating cartesian square and intersection to Cypher. Union has a direct alternative in Cypher, hence it was chosen for support. The union operation has a slightly more complex structure than the transformation operations. Union operation interconnects two parents with vertex and two edges. However, the semantics remain the same - the union operation merges the results of two queries.

The final operation supported by VQL is graph traversal, also known as the "Join" operation. This operation is closely related to the graph structure and is based on the Cypher functionality. However, due to its complexity, it only allows traversal to neighbour nodes. The join structure in VQL consists of two parent entities, two join edges, and a blank vertex with a path string. The path string represents the name of the edge that connects two parent entities. The join operation behaves in the same way as the $() - [] - > ()$ pattern in Cypher queries.

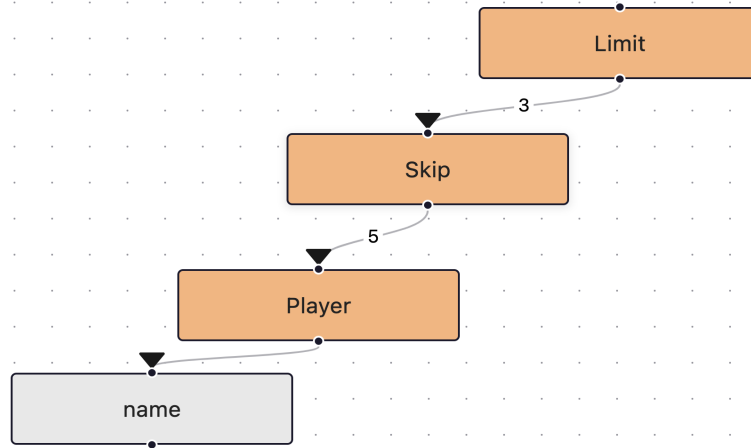


Figure 2.9: Skip and Limit operations

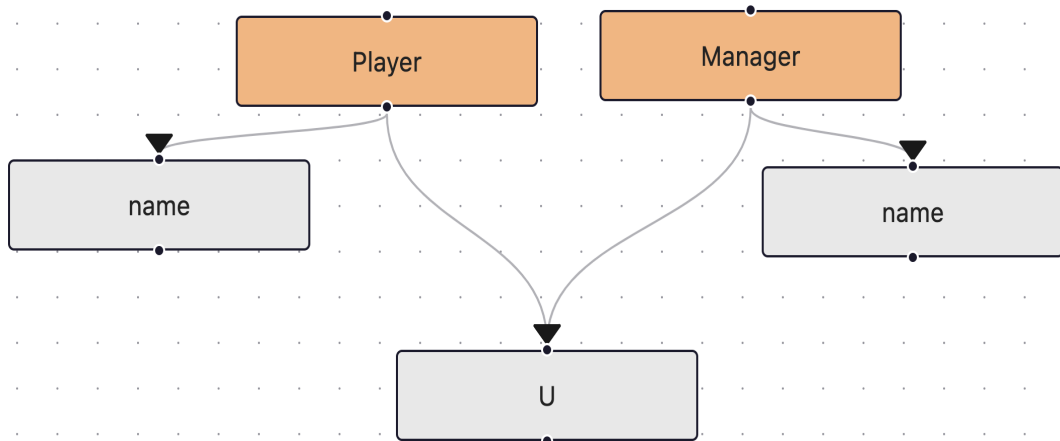


Figure 2.10: Union operations

2.2 Language Limitations

The limitations of the tool can be categorized into two groups: VQL limitations and translation limitations. VQL limitations are related to the editor's inability to visually represent some features of the Cypher language. For example, complex graph traversals are not supported, and the currently implemented traversal only works for nearest neighbors that can be accessed via one edge. A complex traversal such as a multiple edge traversal or finding data by complex traversal patterns are not supported. Additionally, VQL does not support AS statements, RETURN DISTINCT, WITH, and MERGE clauses. DML operations are not supported as well since the main objective of the thesis is to propose a query language. DCL operations are not supported because they are not implemented in Cypher language. It's important to note that VQL operations is a subset of Cypher language operations, and it does not extend Cypher language in any way.

Another limitation of the tool is related to the translation process. The editor supports certain features that cannot be easily translated into Cypher language. These limitations appeared during the design and implementation of the translation algorithms. Some features proved to be very difficult and time-consuming to deal with during the Bachelor's Thesis. For example, operations that do not

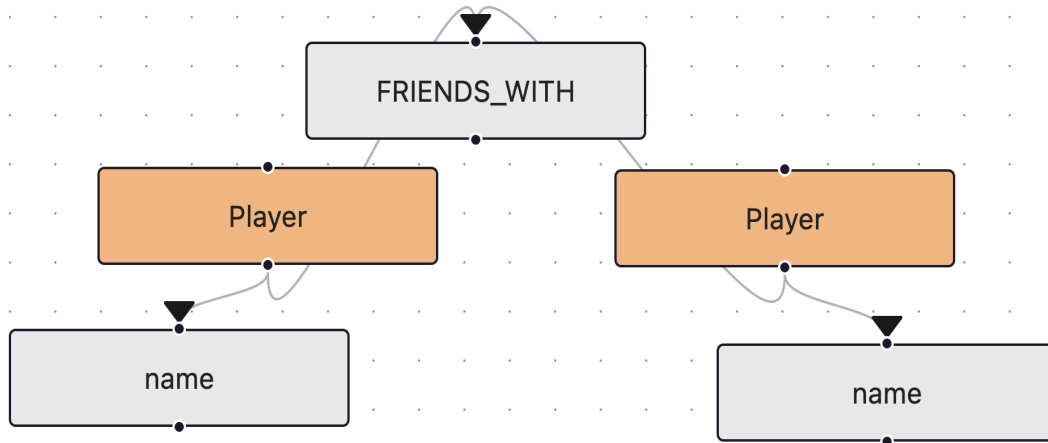


Figure 2.11: Union operations

have direct equivalents in Cypher language such as intersection and the Cartesian square. Implementing such features can be challenging and may not result in a practical solution. To fully implement these features, further research would need to be conducted.

Limitations can be seen as opportunities for improvement, and the fact that some features in the web interface are disabled but still visible brings an avenue for quick reduction of these limitations. By identifying these partially implemented features, further development and improvement can be easily accomplished to enhance the overall functionality of the tool.

Chapter 3

Algorithms

3.1 Root finding algorithm

3.1.1 JSON to Graph convert

Before the root finding algorithm can be executed, it is necessary to preprocess the JSON data. The algorithm takes data as a graph that contains a set of nodes. Deserializing the JSON object to a graph involves iterating through the JSONObjects and mapping them to the corresponding Java classes using the Jackson library. These Java classes, also known as POJOs, are predefined and correspond to the graph entities. Some POJOs can contain nested POJOs, which are often conditional. If Jackson cannot resolve nesting automatically, the nested elements must be parsed manually.

In addition, the nodes in the graph include a string field with the type name, which helps to better distinguish between widgets. The node neighbours are stored as an adjacent list of pairs. Each pair consists of edge and neighbour node. These preprocessing steps ensure that the data is properly formatted and ready for the root finding algorithm to execute.

3.1.2 Finding a root node in a graph

The next step in the deserialization process involves identifying the root of the given model. This is a crucial step, because the root node is a key element in the translation of the model into a set of CYPHER instructions. Once the root node has been successfully identified, the model can be then easily translated into a set of CYPHER instructions.

To identify the root node, a pre-built graph based on the given JSON model is required. The graph should consist of nodes representing the widgets and a list of neighbours connected by appropriate edge. The algorithm for root finding is straightforward. We iterate over each node in the graph until we either reach the end or find a node from which we can reach every other node in the graph.

To determine if a node is a desired root we can run breadth-first search from a chosen node, and if a set of crawled nodes is equal to our graph, then the node is the desired root. The pseudocode for the algorithm is shown below.

If the iteration over the nodes is finished and no root is found, the graph is considered to be rootless. In that case, the query is not valid. It's important to

Algorithm 3.1: Root Finding Algorithm

```
Input: graph – graph representation of the query
1 foreach node in graph do
  // get all nodes from this node
2   nodes = BFS(node)
  // check if root is found
3   if nodes == graph.nodes then
4     | return node
  // no root is found
5   return null
```

keep in mind that the algorithm only returns one a single node and if there are multiple root nodes present, an exception will be thrown.

3.2 Graph to instruction set

The next step in translating the JSON model into an actual Cypher query involves converting the graph with a defined root node into a set of Cypher instructions. Unlike BFS, which traverses the graph in a breadth-first manner, DFS explores the graph in a depth-first manner. Starting from the root node, the algorithm explores as far as possible along each branch before backtracking and moving on to the next branch. Since oriented graph has a unlikely defined root node and all nodes can be reached by the root node, the translation algorithm can guarantee that all nodes and edges will be processed during the conversion and there is no possibility of getting stuck in a cycle. Therefore, the translation algorithm can safely convert the graph into a set of Cypher instructions, even for complex graphs.

Every Cypher query consist of defined set of instructions, with the minimum requirements being to have one entity in the MATCH clause and a corresponding entity in the RETURN statement. Additional clauses such as MATCH OPTIONAL, SKIP, LIMIT, JOIN (graph traversal), ORDER BY and GROUP BY are optional. The UNION operation is also available, which divides the query into sub-queries. Each of these operations can be represented as an appropriate data structure and used to construct the resulting query.

The translation algorithm starts by iterating through the graph, beginning from the root node and processing each triplet, which consists of the parent widget, child widget, and the edge connecting them. However, union and join operations are exceptions to this rule and are known as split operations, which require two child widgets, one parent widget, and two edges connecting the children and the parent. At each iteration step of the algorithm, the triplets or quintuples (in the case of split operations) are categorized into one of the clauses listed before. The edge type, parent widget type and child widget type (in case of split operation children widget types) are used to differentiate between various patterns. For instance, a blank parent vertex connected to the base vertex or a blank vertex through a edge of skip type is considered a SKIP clause. Furthermore, in such a case, the number of SKIP rows in the result is determined by the

edge.

To generate the final query, the translation algorithm processes all the relations in the graph and fills in the relevant clause data structures. The resulting query will only contain non-empty clauses. To encapsulate all the clause data structures and operations related to them, the Query class exists in the backend of the tool. In addition, for union operations, sub-queries can be included within one query. However, the number of union sub-queries is limited to two. The pseudocode for the algorithm is presented below.

Algorithm 3.2: Graph To Instruction Set Algorithm

```
Input: graph – graph representation of the query
Input: query – resulting query
Input: node – current processing node
Input: variable – current variable marking entity
1 Function dispatchOperations(query, graph, node, variable):
   | // iterate through node neighbours
2   | foreach pair in node do
3   |   | child = pair.getValue0()
4   |   | edge = pair.getValue1()
5   |   | dispatchNonSplitNode(query, graph, node, variable)
   |   | // recursively process graph
6   |   | rootFindingAlgoritihm(query, graph, node, variable)
7 End Function
8 if node is a split operation then
   | // Process Union and Join operations
9   | dispatchSplitOperations(node)
10 else
   | // dispatch non split operation
11 | dispatchOperations(node)
```

The recursive nature of the algorithm is reflected in the presented pseudocode, which only covers the processing of non-split operations. The handling of split operations is very similar but requires additional sub-functions that cannot be presented due to large function nesting in the given pseudocode. Once the translation algorithm completes, it returns the query parameter back to the main function, which can then execute the query.

Chapter 4

Programmer documentation

4.1 Intro

The tool has a client/server architecture. The architecture is advantageous for a queering tool as it allows multiple users connect to the server and interact with a database. Additionally, this design enables a clear separation between the edition and evaluation parts of the application, which is highly beneficial. The tool's architecture is divided into two parts: the front-end and the back-end. The front-end application is responsible for creating, editing, and displaying query results, while the back-end handles query deserialization and evaluation.

The application's server-client communication relies on HTTP queries. The client-side sends a graph object, represented as a JSON file, to the server and receives a response in the same format. This graph object, also referred to as the model, contains information about the user's graph query, which will be further discussed in subsequent chapters. In case of unexpected behaviour, the server may also return an error response.

4.2 MVC

The vanilla MVC pattern has been a fundamental pattern for web applications and has dominated the software world in recent years. While there is a trend towards more modern solutions such as MVVM or serverless applications, the MVC pattern is a perfect fit for our application. Furthermore, using React with a Redux library in the front-end brings a new perspective to this concept. The combination of MVC and this technology stack can be extremely useful for managing data and saving its immutability.

The front-end part of the application also employs an MVC pattern. This decision was made because it simplifies operations with the query mode. Isolation the query model from view helps achieve constituency of query model. Additionally, this approach facilitates easier extraction of models from the front-end application and sending them to the server. Redux state container is added to the application for the purpose of better data integrity and to avoid undesirable side effects such as property drilling. Redux turns MVC drawbacks into advantages by utilizing a circular flow to ensure data immutability. It also effectively separates logic from view components, preventing any component from directly manipulating the model or data. React only serves a visualizer role, which perfectly fits

the MVC pattern. The application implements necessary changes only in slice objects and saves data to the Redux store. The Redux logic can be distributed among different utility files without affecting the overall concept.

4.3 Front-end architecture

4.3.1 Model

The model class is created to achieve query data integrity on front-end. The new model object is created when the application starts. The model object composed from two main nested objects: Widgets and Edges.

4.3.2 Widgets

Widgets are represented by three main elements: vertices, operations and conditionals. The operations than consists of transformation and binary operation. The main difference between them is that vertex widget contains only a single vertex element, such as a base vertex, meanwhile an operation is a structure, which consist of both vertices and edges. Conditionals are somewhat different from the other two, as they are vertices but also function as a structure since they always belong to a specific vertex.

Conditionals and widgets are built upon the vertices, which serve as their foundation. The Vertex is a derived class of the Widget class. It extends base class with following fields: id, style, data and position. The id field is a string that stores the unique identifier of the vertex. The data property includes vertex label, which is displayed on the vertex rectangle shape in the editor. The style field contains the CSS styling of the vertex, while the position field specifies the coordinates of the vertex on the editor layout.

Sub-classes of Vertex widget include blank vertex, base vertex, cartesian square vertex, intersection vertex, union vertex, and join vertex. The blank vertex is used in structures as a utility vertex in case when there are no other use cases. The base vertex is used to represent entities and properties. In addition to the base class fields, the base vertex class requires a vertex color to distinguish between different vertex types, a conditional to filter the output result and an isProjection boolean to signify if the value will be projected to the return clause of a query. The conditional field can be left empty if no conditional is specified. The remaining vertices are fundamental vertices in structures with the same name as the vertex.

It would be reasonable to start by discussing transformation operations as the first type of operation. This operation is a structure and it consists of three parts: a transformer, a parent and an edge. The transformer is a vertex that represents the type of transformation being applied. The parent is a vertex on which the transformation is performed, and the edge is an edge of the appropriate type that connects the parent and transformer. Transformations are operations that modify the output result in some way. The operation includes order by, group by, skip and limit operations. Because of the specific implementation of group by, it is divided into group by count, group by min and group by max. However, only group by count is available in the editor. Additionally, group by

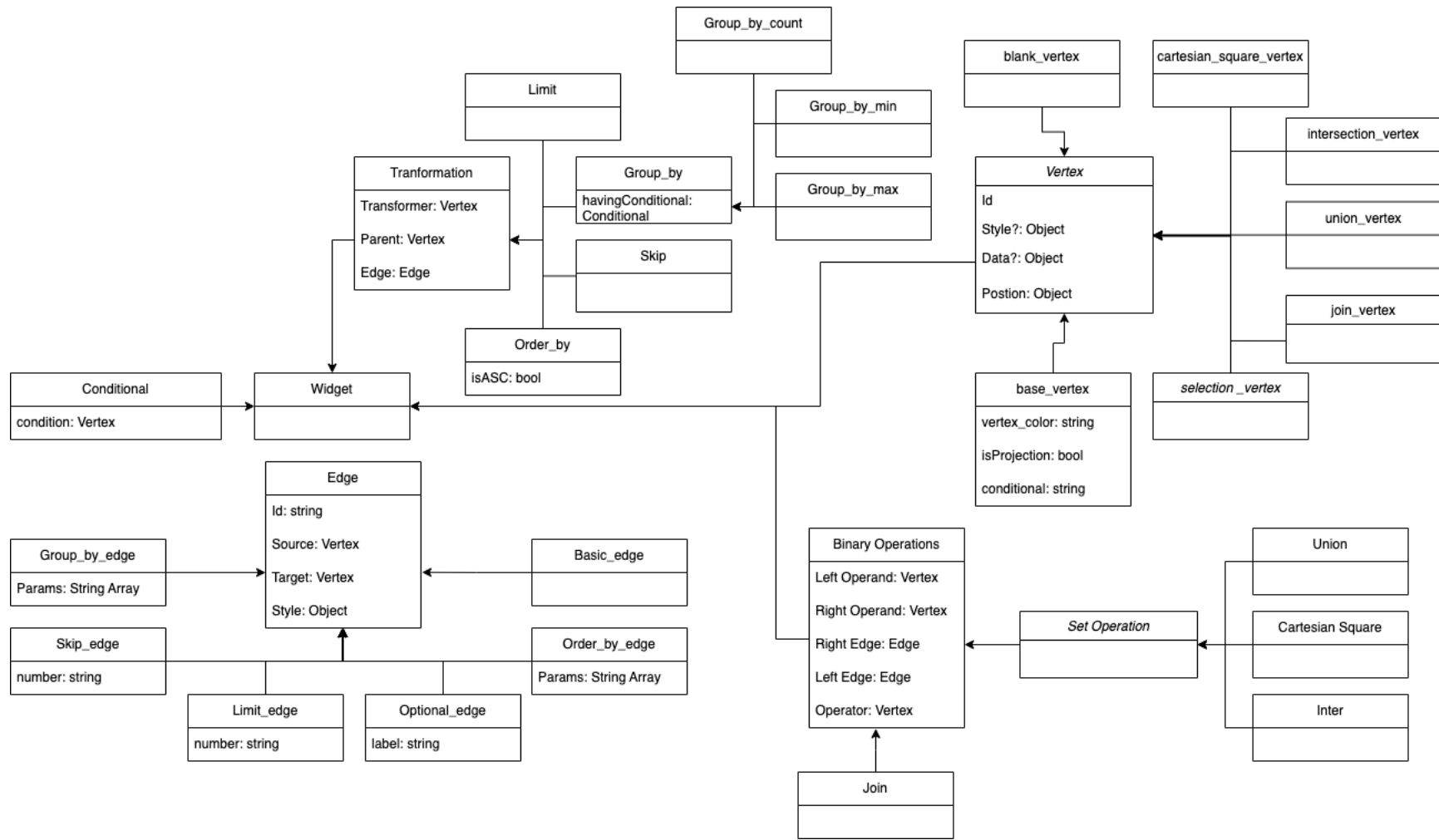


Figure 4.1: Model Class Diagram

is distinct from other transformations in that it has conditionals. This is due to the implementation of the having function in the group by operation.

The following type of operation is a superclass called binary operation. This class represents operations that requires two parent widgets to be connected with. Binary operation includes set operations and join operation. The superclass extends widget class with an operator vertex, left and right vertex operand and two edges connecting the operator with operands. Set operations represent operations performed on a set of results, such as intersection, union, and cartesian square. Currently, only union is available in the editor, since it is not clear how to translate intersection and cartesian square to cypher query. Join operations represent graph traversals.

The preceding text mentioned the conditional widget twice already. This widget serves as a container for string conditions and is connected to another widget. Conditionals can be used in entity properties or in the group by having operation.

4.3.3 Edges

Edges are elements that connect widgets in a query graph. The Edge superclass includes a string edge ID, a source vertex, a target vertex, and style properties that specify the arrow positions and color. The Edge superclass is further divided into subclasses, that can be used to distinguish between different relationships among widgets. Additionally, some edges require specific fields. There are six different edge types, such as group by edge, skip edge, limit edge, optional edge, order by edge, and basic edge. The basic edge is typically used to connect an entity with its properties. The skip and limit edges are used in skip and limit operations, respectively. These edges are similar because they require an additional field indicating the number of rows to skip or limit in the output. The optional edge is used connect entity with an optional property. The group by and order by edges are also two similar edges. Both require an additional parameter field that contains the property name used in the group by or order by operation.

Graph queries are constructed using widgets and edges, which are the fundamental elements. The resulting model objects consist of an array of widgets, accessible under the "widgets" key, and an array of edges, accessible under the "edges" key. Worth mentions, that Edges array contains edges, which connects entity with its properties. Edges, which are included in operations, are stick to the operation and they are stored inside the operations.

4.3.4 View

The View class is a vital component of the MVC implementation. It processes the model object and transmits the processed data to the React Flow library, which then displays the graph in its editor. One of the main issues it addresses is the use of different data types in the model and the Flow Chart visualizer library. React Flow library requires the graph in the format of vertex array and edge arrays, while the model object includes widgets array and edges array. The View class parses the widget data and extracts the vertices and edges from them. The models vertices and edges are desinged in a way that does not require

any additional mapping to React Flow's vertices and edges. The View's parsing method is located in the `updateView` function, which is invoked every time the model changes in the Redux state function.

4.3.5 Main functions

One of the most fundamental classes in the frontend application is the model class, which was previously discussed. This class serves as a structure or container for storing data and enables easy extraction of the data when needed. In this chapter will be mentioned fundamental functions, which participate in query building and editing.

Redux Functions

The majority of the frontend application's logic is organized into functions and stored within a Redux slice named "mainSlice". Each Redux function receives state and action parameters. The state parameter contains all the states (fields) used within the particular slice, while the action parameter is usually used to pass arguments within it.

processBasicVertexCreation The function is executed after base or projection vertex create button is clicked. The function receives "msg" and "type" parameters from the action object. The "msg" parameter is a string message that informs the user to specify the vertex label. Next the "type" parameter is used to add a new vertex to model using `addVertex` function.

addVertex The function receives new vertex as a payload of the action object. The vertex than is added to models widgets array and the `updateView` function is called to update application view objects.

processNodesChanges The function is responsible for handling any user interaction with the widget. Depending on the action payload, the function calls either `processNodesPosition`, `deleteElement`, or `processNodesSelect` functions.

processNodesPosition This function is responsible for updating the position of a vertex based on a given widget id. It takes in a new widget id and new coordinates as parameters. After the update is made, the component is reloaded to reflect the updated position.

processNodesSelect This function is responsible for tracking the user's clicks on the widgets. If the user is currently in a waiting state, the function will handle the user's click event. Depending on the type of waiting object that needs to be created, the function will either invoke `processTransformationCreation` or `processBinaryOperationCreation` to create a transformation operation or a binary operation, respectively.

processTransformationCreation The function is called during the processing of a user click event. The function is invoked from processNodesSelect and it is responsible for finding parent node for transformation operation based on parent id. Once the parent vertex has been successfully found and the transformation edge has been created using the waiting object type, the function proceeds with creating a new transformation widget that includes the transformation edge, the parent vertex, and the corresponding waiting object itself. Finally, the newly created transformation widget is added to the model's widget array. The findWidgetById function is utilized to find the parent vertex by its ID, while the createTransformationEdge function is utilized for dispatching the creation of the transformation edge. Finally, the createTransformation function is utilized for dispatching the creation of transformation widget itself.

handleTransformationRequest The function is responsible for handling the creation request for a transformation operation. It receives transformation vertex id, label and type as the parameters. This function is called after the create new transformation button is clicked. This function is responsible for setting the waiting objects and their type. Additionally, it displays a notification to the user, prompting them to select a parent vertex.

handleSetOperationRequest The function is similar to handleTransformationRequest, but it handles the creation request for a set operation and join operation.

processBinaryOperationCreation This function is similar to processTransformationCreation in terms of its invocation and responsibility, but instead of creating transformation widgets, it creates binary widgets. Additionally, this function requires two parents to be connected with a binary operator, and two binary operation edges should be created to interconnect the parent vertices with the binary operator vertex. Once both parent vertices have been successfully located and the two binary operation edges and binary operator have been created, the binary operation widgets can be created and then added to the model's widget array. To dispatch the binary operation widget creation, the createBinaryOperator function is utilized.

processNodeEdit The function is responsible for handling vertex editing. The function is invoked when the vertex is double-clicked. The function creates a new vertex based on the editor information and replaces the previous node with the updated one.

processEdgeEdit The function is responsible for handling edge editing. The utility function findEdgeBySourceAndTarget is used to find the old edge and then creates a new one based on the edge type specified in the editor.

addEdge The function is responsible for handling creation of a new edge. The function is invoked each time a new edge is created in the editor. The React Flow library notifies the appropriate function when there is a change in the edge set of

the graph, which in our case is the `addEdge` function. Once the source, target, and id variables are received, the function proceeds with creating the new edge and adds it to the edge set.

Utility functions

updateView The "updateView" function belongs to the View class and is called by the Redux `mainSlice` every time the model changes. It parses the widgets and extracts the vertices and edges from them. The extracted data from widgets is then merged with the model's edge array and store into view object.

processTransformationCreation The function is different from the slice function with the same name, because it handles creation on a component level. The purpose of this function is to create a transformation type operation. several parameters, including the ID of the new transformation vertex, the type of transformation being created, a label to be placed on the transformation vertex, and a dispatch function. The function begins by creating a blank vertex to serve as the transformation vertex, using the provided ID and label. It then sets this newly created vertex as a waiting object. Once the user has selected the parent vertex the `processNodesSelect` function is called. This creates a new transformation widget with the waiting transformation vertex, the selected parent vertex, and the appropriate edge. The dispatch function is utilized to invoke the slice function outside of slice.

4.4 Back-end architecture

The application's backend is developed in Java using the Spring Boot framework and the Jackson library. Its primary responsibilities include handling user requests with graph queries, deserializing graph queries from JSON format, managing connections with the NEO4J database, executing translated queries on the database, and returning the results to the user.

The backend application architecture follows the standard Spring Boot application structure, consisting of layers such as presentation layer, service layer and data access layer. The presentation layer is responsible for handling user requests and is represented by the Controller class. The service layer contains the logic of the entire application. This layer is represented by Processor and Parser class, which are service components in the Spring context. The data access layer is represented by a single Repository class, which handles communication with the NEO4J database

By using the Dependency Injection pattern, the components are wired together, and with the help of Inverse Control Injection, these components are managed by the Spring framework.

4.4.1 Main Classes and Functions

This subsection covers the primary classes, their methods, and the main functions. The backend architecture's class diagram is provided below, excluding

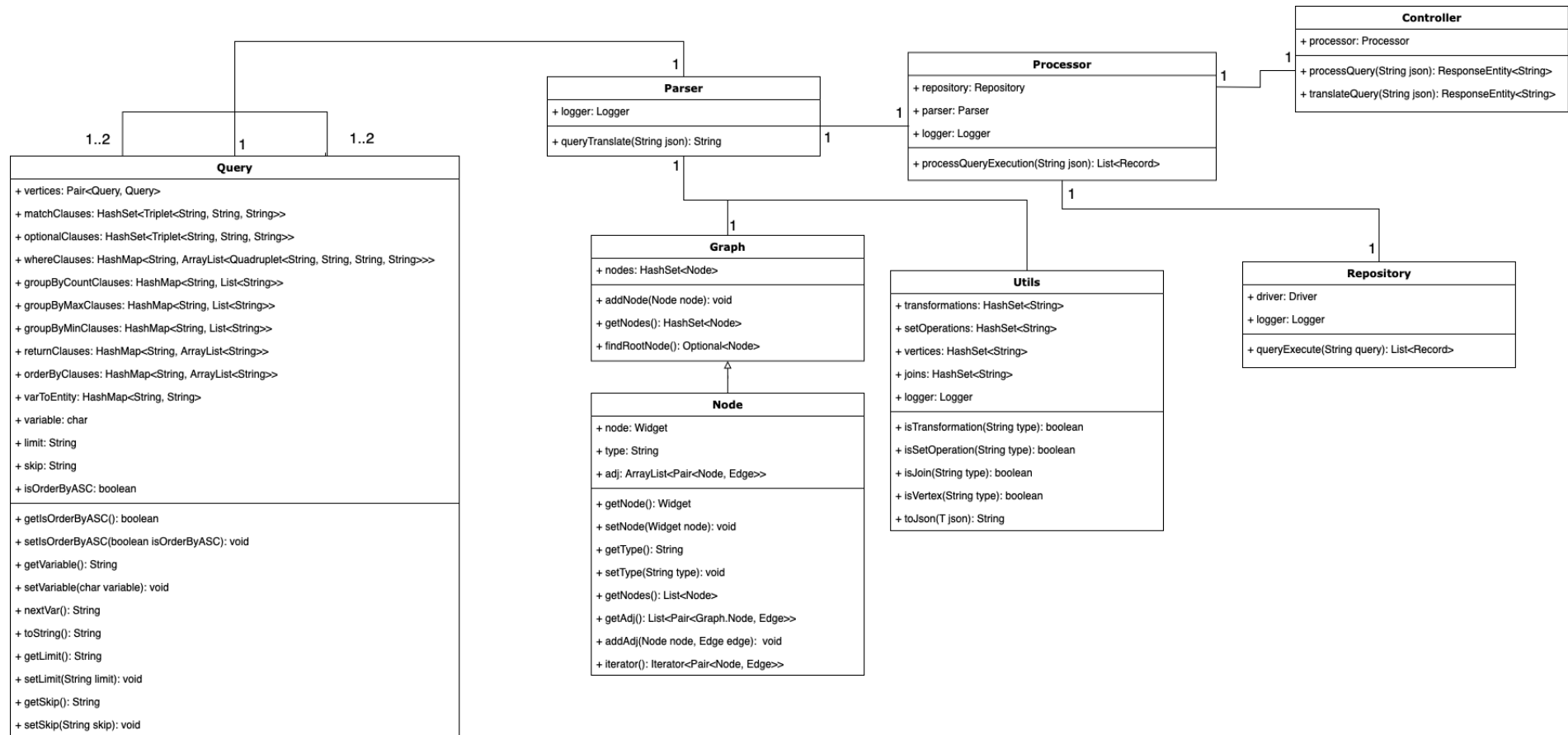


Figure 4.2: Backend Class Diagram

well-known classes such as Pair, Triplet, and Quadruplet, which have no effect on the application logic and are implemented to simplify operations.

AppApplication

The class is the entry point of Spring Boot application and contains the main method.

Controller

The class is responsible for dispatching user request. The class provides back-end application API, which consists of `"/translate"` POST endpoint and `"/execute"` POST endpoint. Once the request is received controller forward request to processor or to parser class. Additionally, the controller is responsible for catching any errors that may occur during query execution, and returns an error message to the user.

Processor

The class is responsible for processing incoming queries. The parser logic is provided by

processQueryExecution The function is responsible for handling incoming query as JSON string. It accepts a single argument, which is the graph query model in JSON string format. The string is processed using the Parser class, after which it is passed on to the repository component for execution on the NEO4J database. The function then returns the results of the execution to the user.

Parser

The class is responsible for translation the the incoming JSON string query into a string query that can be executed in the NEO4J database. The Parser class makes use of the Query class, which represents Cypher query. The `queryTranslate` method serves as the single entry point for the class.

queryTranslate This function is the entry point for the Parser class, responsible for parsing JSON strings containing deserialized query models. The first step is deserialization, where data objects from the JSON model string are mapped to Java data objects. After a successful mapping, the function builds a graph using these objects, which can then be translated to an executable Cypher query using the `getOperationsSet` function.

extractEdges The function is responsible for edges deserialization. It extracts edges from the JSON representation of the query model provided. Once the string edges are extracted, they can be mapped to Java classes, similar to those on the frontend. The function then returns the list of deserialized edges.

extractWidgets The function does similar job as the `extractEdge` function, but for widgets.

buildGraph This function is responsible for constructing a new graph using the extracted widgets and edges. It takes two arguments: an array of widgets and an array of edges. The function then iterates over the widgets array and constructs a graph based on it. To store constructed graph the Graph class with nested Node class is used.

getOperationsSet The function is responsible for translation the deserialized query graph into a set of Cypher instructions. It recursively processes each relation in the graph and translates it into an instruction. The resulting set of instructions is then stored in the Query class.

Graph

The class is responsible for handling the graph representation of the query. The nodes of the graph are stored as a list of nested Node objects, with each Node object representing a vertex in the graph query. Additionally, each Node object contains a list of adjacent nodes along with their corresponding edges.

findRootNode The function is responsible for finding the root note of the graph. Since the graph is the oriented graph representing the query, the root node should be exactly one. The function return empty optional if no root is found.

Query

The class is responsible to hold set of instructions. Each Cypher clause has its own instruction set. Once the whole graph processed the overriden toString method can be called, to return a executable Cypher query.

toString This overridden function returns an executable Cypher query as a string. The function merges all the instructions found into a query and returns the resulting string.

Utils

The class is the container for general static functions. The Utils class provides solution for identifying widget type, translating a object to JSON format and more.

Repository

This class is responsible for communicating with the NEO4J database and executing queries. The main entry point of the class is the executeQuery function, which takes a Cypher query as input, executes it on a running instance of the NEO4J database, and returns a list of records.

4.4.2 Model

The backend query model for the application consists of `Widget` and `Edge` super-classes, which are identical to their frontend equivalents. These classes are used to map serialized frontend graph queries to Java classes for processing.

The backend model class diagram is showed below. It differs from the frontend model as it does not support intersection and Cartesian square operations, which are not included in the diagram. Additionally, getter and setter boilerplate functions are not shown in the diagram, but are replaced by "getters and setters" commented lines of code.

While the Jackson library can easily map nested classes using the `JsonSubTypes` annotation, manual processing of objects is required for three or higher levels of nesting. An example of this is the parsing of conditional `SelectionVertex`, which is the third level of nesting. Custom functions like `processConditionals` are used to map such objects in `QueryByCount` class and `Vertex` subclasses.

4.4.3 Use cases

The section describes a detailed description of the interactions between components, classes, and users. Sequence diagrams are used for the better representation of the textually described process. The sequence diagram illustrates the flow of messages or method calls between different objects or actors in a chronological order. The diagrams in this section focus on illustrating the interaction between written parts of application, rather than interactions between various parts of frameworks or libraries.

In our case, the diagrams represents the interaction between the user and the application following a specific action. The user interacts with the application through a browser application, which involves actions such as creating, deleting, and editing vertices, as well as sending responses. Each of these processes will be discussed in further detail.

As mentioned earlier, the initial and simplest user interaction involves editing the graph query. Once the user wants to create a new vertex, they will click the create new vertex button in the footer component. The creation of a basic vertex will directly trigger an action in the Redux store. However, for the creation of transformation or binary operation, additional handler functions will be used. Once the corresponding Redux store function is called, it will handle the vertex creation process within the Redux slice. During the creation of the vertex, additional information such as the vertex label, parent node, or vertex parameters may be requested for specification. The creation of a basic vertex is illustrated in the sequence diagram below.

The basic vertex creation process requires from the user only to specify the label name. However, for more complex operations, the user needs to specify parent vertex/verticies as well. To handle such The handle such asynchronous response from the user, the handler functions are used. The handler function is responsible for setting waiting object and its type. Once the parent vertex is specified, the function responsible for processing operation creation is called. The processing function then creates the necessary dependencies for the operation. The `utils` file provides functions for creating these dependencies.

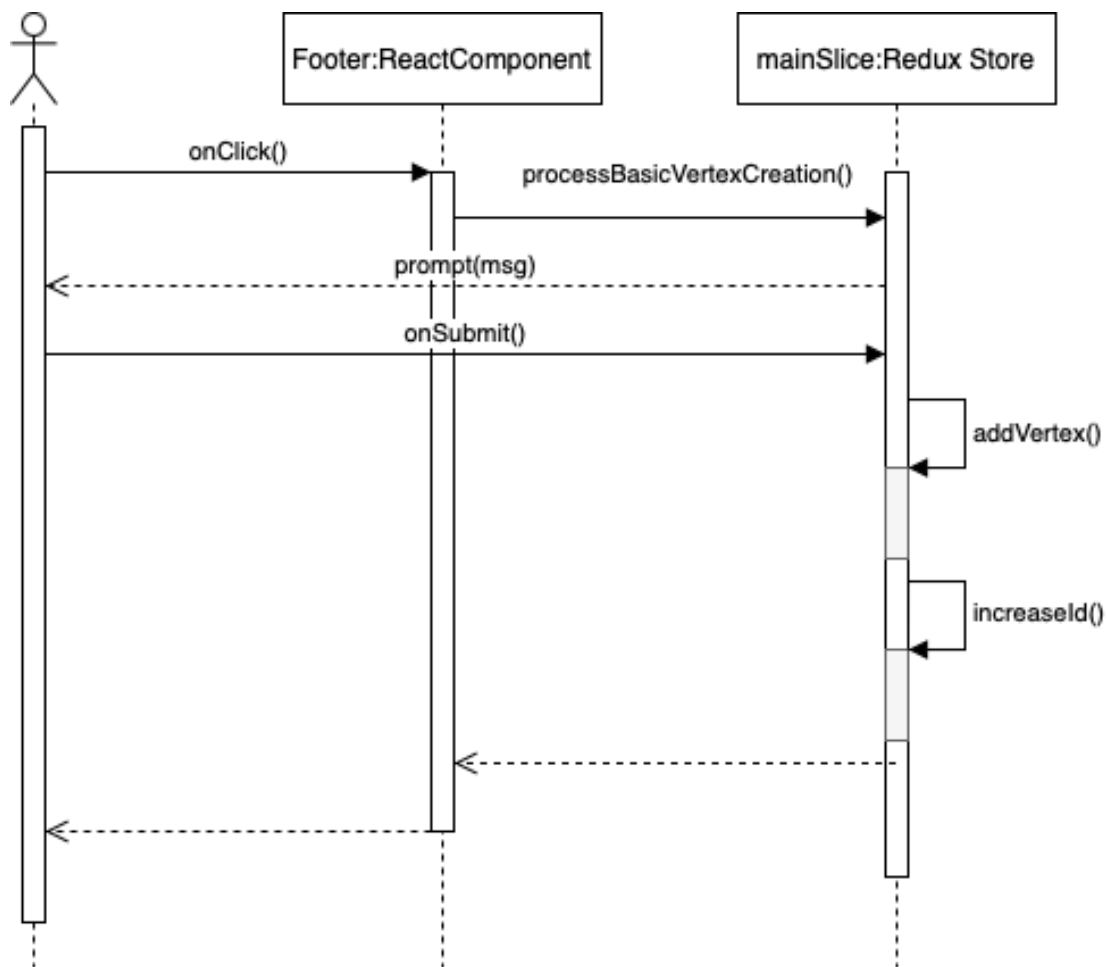


Figure 4.3: Creating Basic Query Sequence Diagram

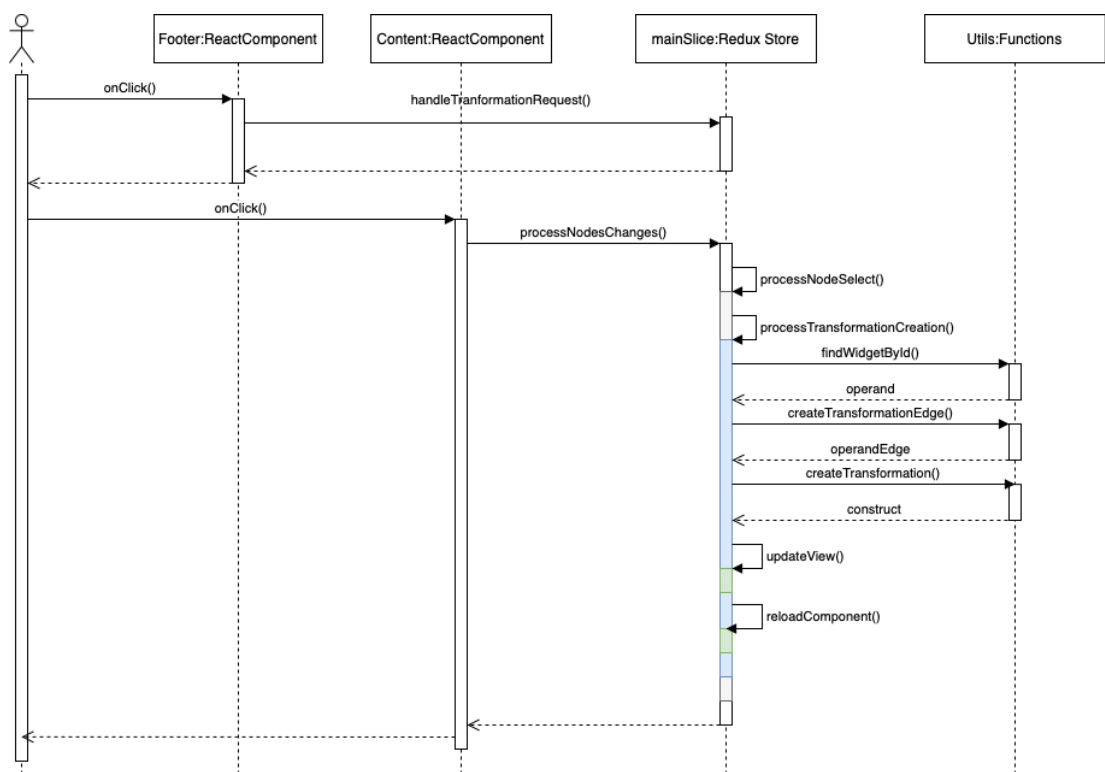


Figure 4.4: Creating Complex Query Sequence Diagram

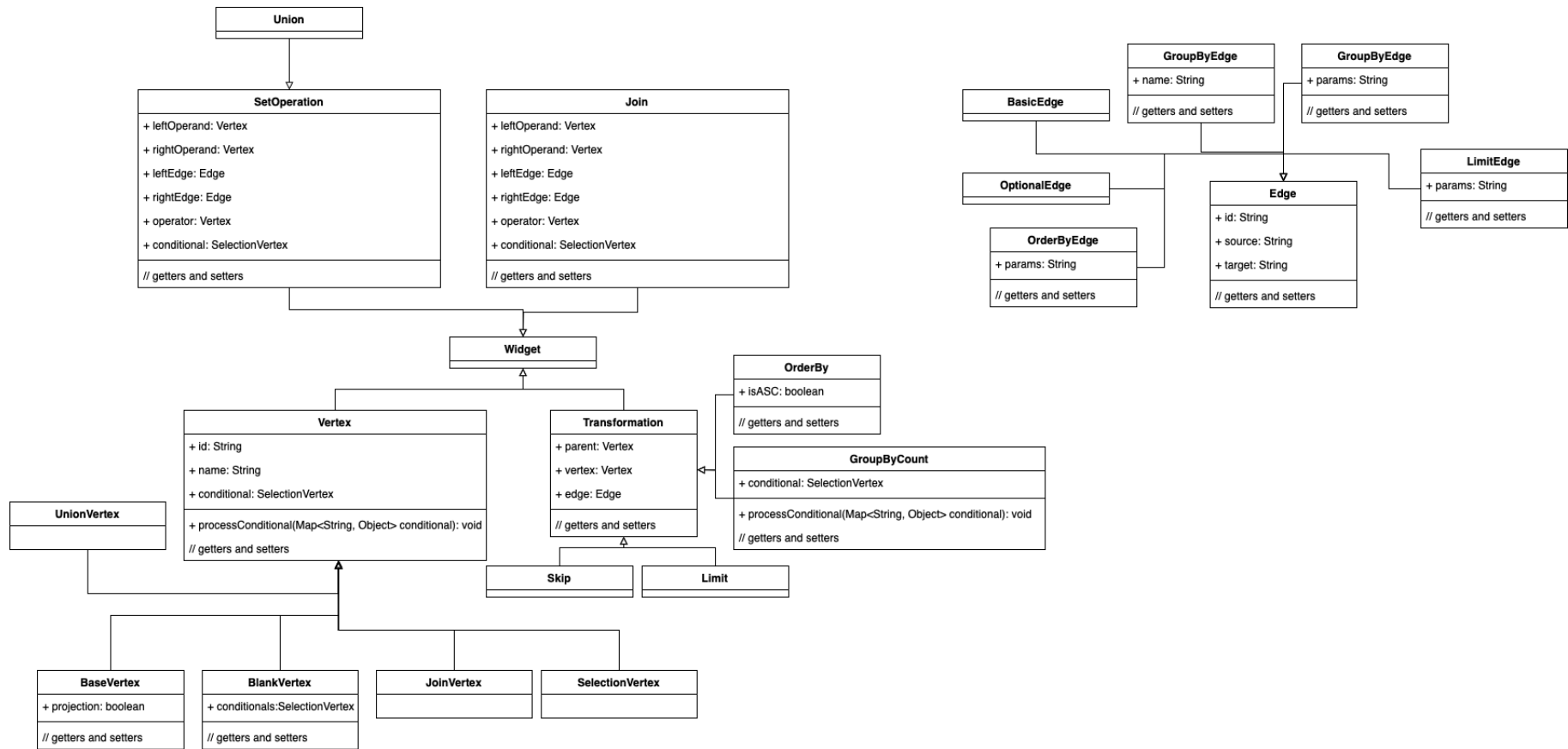


Figure 4.5: Backend Model Class Diagram

Once the user has finished editing the graph query, they can proceed to execute it by clicking the execute button. The responsibility of the execute button is to send the request to the server. After receiving the response, the Controller component is responsible for handling it. The response is then forwarded to the Processor component, where the query is deserialized and executed if the deserialization was successful. The response from the execution is then returned by the Controller component. However, if an error occurs during the process of translation or execution, an error message is returned instead.

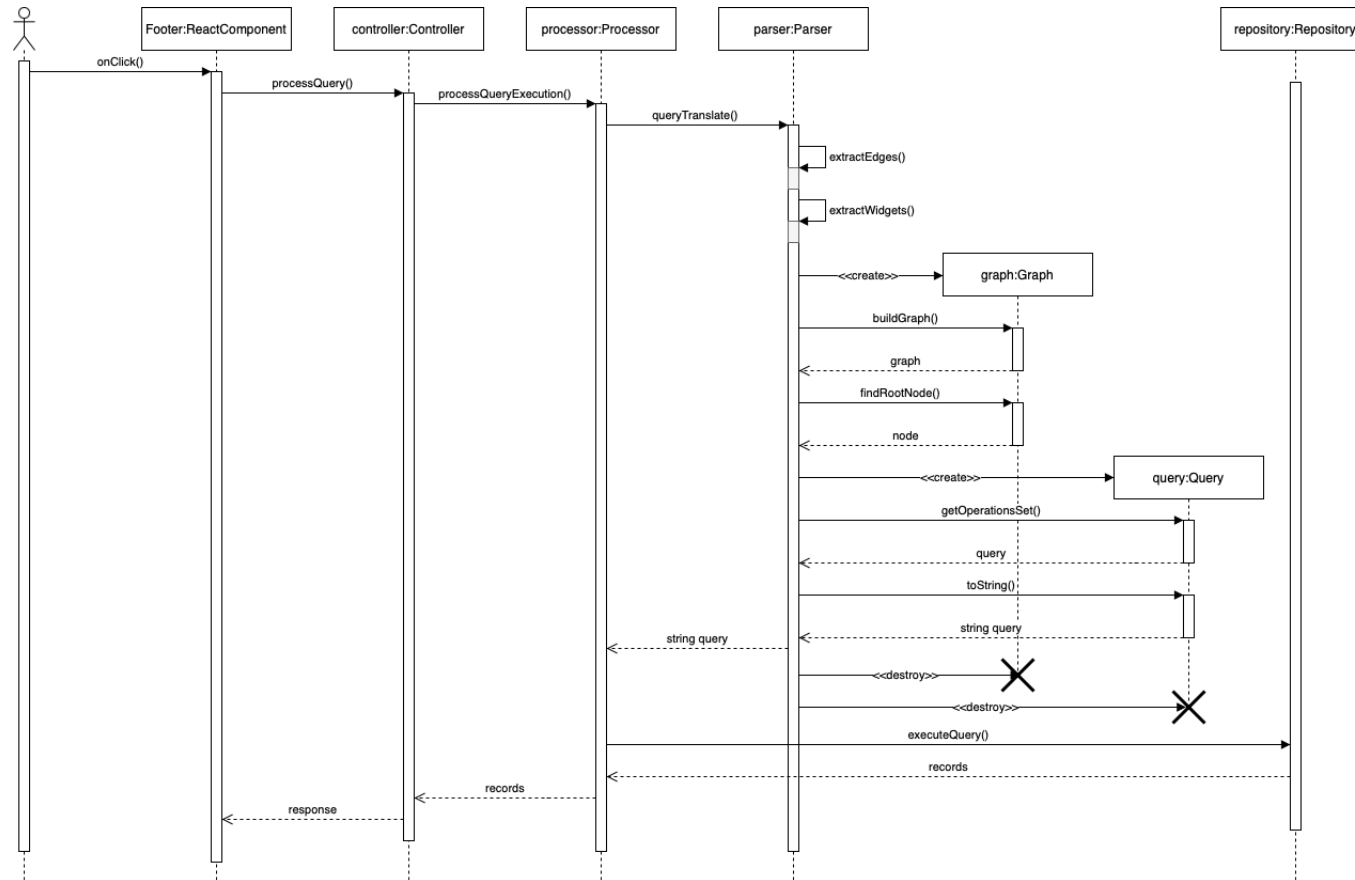


Figure 4.6: Query Execution Sequence Diagram

Chapter 5

Testing

5.1 Introduction

This chapter provides an overview of the testing methods used to ensure the functionality correctness of the application. Two primary methods are used for testing purposes: unit testing and integration testing. All of the tests can be found within the test repository.

5.2 Unit tests

The application utilizes JUnit and Mockito test frameworks for performing unit testing. Unit testing so far is used only on single `findRootNode` method, which is responsible for identifying the root node of a given graph object. As for the other public methods, they has no general use and therefore do not require testing. The test cases for the `findRootNode` method can be found in the `GraphTest` class. These tests verify the correctness of the `findRootNode` function by checking if it can accurately determine the root of a simple graph. Additionally, the tests validate that the function returns an error message in scenarios where no root is found, the tested graph contains a cycle, or there are multiple roots present.

5.3 Integration Tests

Integration tests are the over type of test, which are used for testing the application. Integration testing is a technique that validate correct interaction between different components and modules. In case of our program, it primary tests correctness of the deserilized graph query translation. The translation aspect of the integration tests is performed on the backend of the application and is encompassed within the `ProcessorTest` class. The description of the integration tests includes a graphical representation of the query, which is the same as what the user sees in the editor. Additionally, it displays the executable query in string format, providing the user with a preview of the query before execution. Users can perform integration tests using the examples provided below and the preview function of the editor.

5.3.1 Projection Test

The test aims to validate correct translation of the simple query. The query consists of a single entity "Player" and a single projection property. Below is the visual representation of the query:

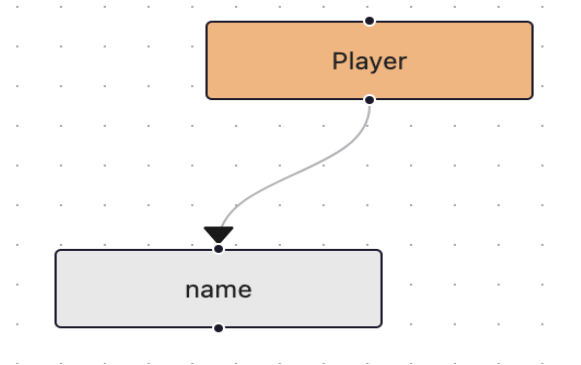


Figure 5.1: Basic Projection

The result query should be as following: **MATCH (a:Person) RETURN a.Name;**

5.3.2 Selection Test

The test aims to validate the correct translation of simple query with selection included. The query consists of a single entity "Player" and a single property with selection conditional. Below is the visual representation of the query:

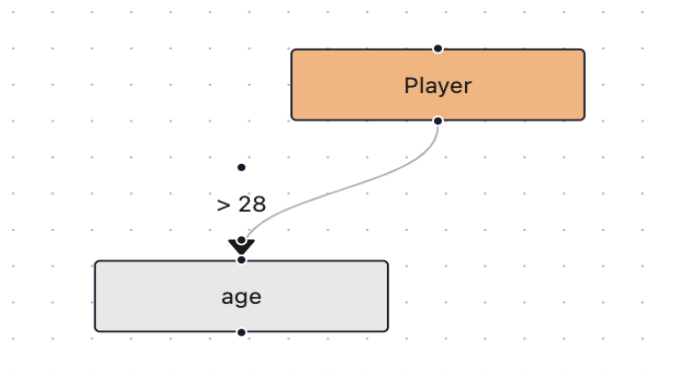


Figure 5.2: Basic Projection with Selection

The result query should be as following: **MATCH (a:Player) WHERE a.age >28 RETURN a.age;**

5.3.3 Aggregation Test

The test aim to validate the aggregation function "group by", specifically the "group by count" version. The query consists of a single entity "Player" and a transformation operation "group by count", which is applied to the "nation" property of the "Player" entity. Additionally, the "group by" operation includes a conditional clause. Below is the visual representation of the query.

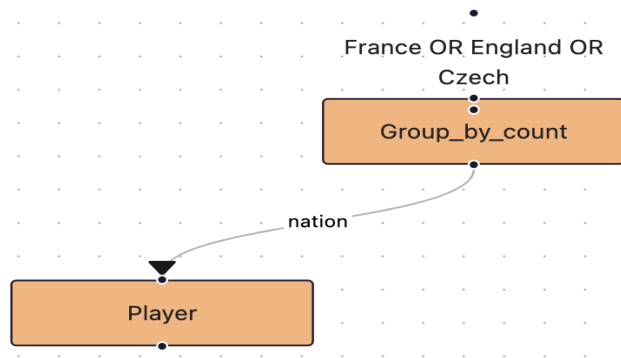


Figure 5.3: Basic Projection with Selection

The result query should be as following: **MATCH (a:Player) WHERE a.nation = 'France' OR a.nation = 'England' OR a.nation = 'Czech' RETURN COUNT (a.nation);**

5.3.4 Multiple Selection Test

The test aim to validate the handling of multiple conditionals. The query consists of a single entity "Player" and three projection properties of its node. Two of these properties include conditionals. Below is the visual representation of the query:

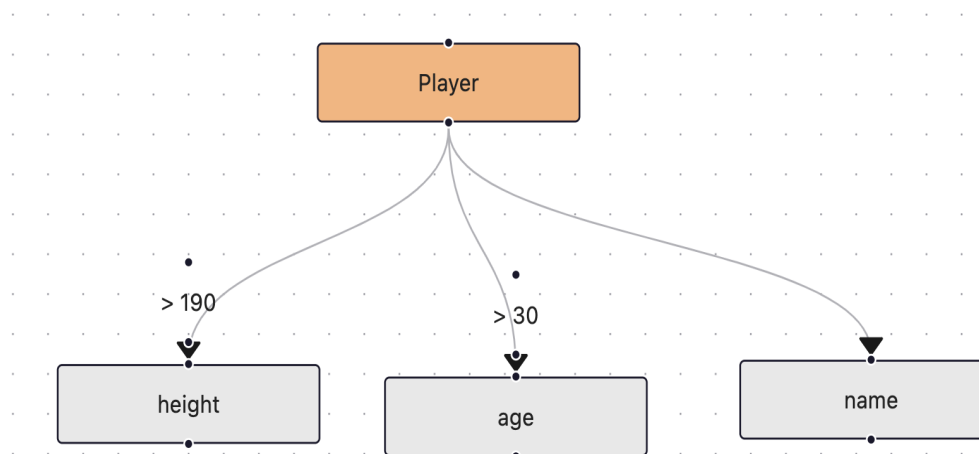


Figure 5.4: Basic Projection with Selection

The result query should be as following: **MATCH (a:Player) WHERE a.age >30 AND a.height >190 RETURN a.age,a.height,a.name;**

5.3.5 Order By Test

The test aim to validate the translation of the order by operation. There are two versions available: ascending and descending. In this example, the descending version is used. The query consists of a single entity "Player", its property name, and an order by descending operation performed on the name field. Below is the visual representation of the query:

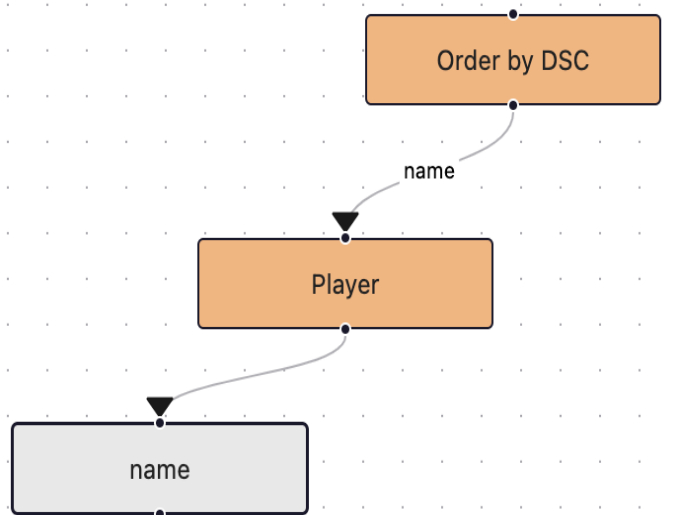


Figure 5.5: Basic Projection with Selection

The result query should be as following: **MATCH (a:Player) RETURN a.name ORDER BY a.name DESC;**

5.3.6 Order By Test

The test aim to validate the translation of union operation. The query consists of two entities, "Player" and "Manager", connected with a union operation. Each entity has a name selection property. Below is the visual representation of the query:

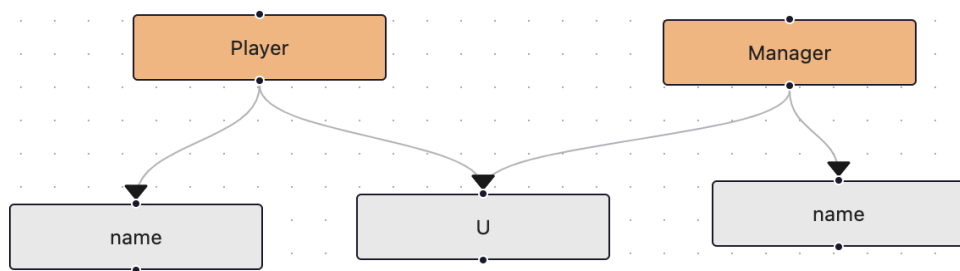


Figure 5.6: Union

The result query should on of the following options: **MATCH (a:Player) RETURN a.name UNION MATCH (a:Manager) RETURN a.name;** or **MATCH (a:Manager) RETURN a.name UNION MATCH (a:Player) RETURN a.name;** Both options are considered correct due to the use of a hash set in the implementation of the translation algorithm, which ensures that the order of the entities in the union operation does not affect the final result.

Chapter 6

User documentation

6.1 User Documentation

6.1.1 Requirements

Java 17 or higher, npm 9.4.0 or higher, IntelliJ IDEA (installation can be done with editor such as Visual Studio Code)

6.1.2 Installation guide

Due to specific authorization problems with the NEO4J database, a more manual installation process is required, as it has been extensively tested and proven to work reliably. The installation guide has been tested on macOS systems with Java 17 installed, although lower versions of Java (from 11) also work without any issues.

6.1.3 Application Installation

Once the user receives a zip file containing the source codes of the application, the installation process of the application can begin. The first step in the installation process is to unzip archived file with source code and open the extracted project with an integrated development environment (IDE). The project is divided into two subprojects: frontend and backend. The user can choose to use a single IDE that supports both projects or use separate IDEs for each project. This approach has been tested with editors such as Visual Studio Code and IDEs like IntelliJ IDEA.

Neo4j database

The correct work of application requires the local instance of NEO4J database running. To create a new instance of the database, you will need to install the Neo4j Desktop application, which is freely available for mainstream operating systems. You can download the application from their official website¹. Once the Neo4j application is installed, you can proceed to create a new database instance.

¹<https://neo4j.com/download/>

To create a new database project, open the Neo4j Desktop application and click on the "New" button located in the top-right corner. Then, select "Create project" from the options. You can provide a suitable name for the project. The next step is to create a new local database. Click on the "Add" button and choose the "Local DBMS" option. It is crucial to set a password for the new database during this step. The backend jar file is configured to use the default username and "password" as the password. If you wish to use a custom password, you will need to recompile the backend jar files with the updated application properties.

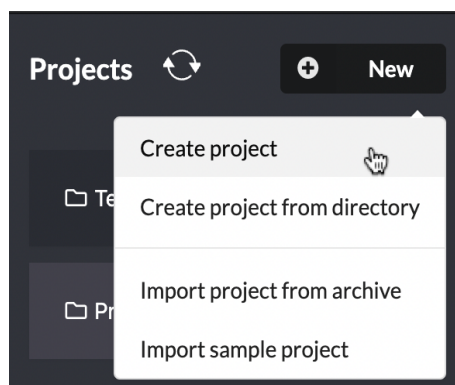


Figure 6.1: Project Creation

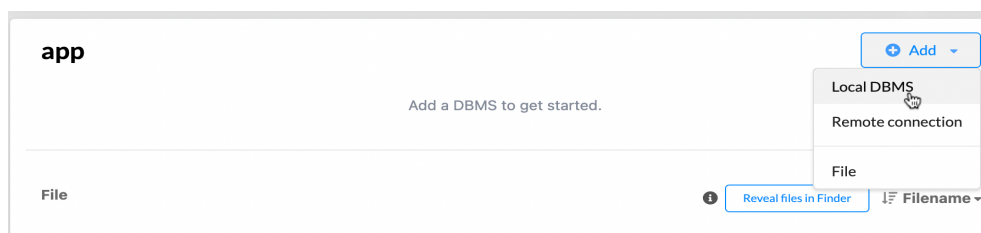


Figure 6.2: New Local DBMS creation

Once you have completed these steps, you can start the application by clicking the "Start" button that appears when you hover over the database label in the Neo4j Desktop application. The Neo4j Desktop application provides full control and management capabilities for the database instance. Once the database instance has started successfully, you have the option to open the Neo4j browser to interact with the database.

The next step is to load the necessary scripts or data into the database. If the user does not have any prepared data, the application's source code provides you with demo data. The script for loading the demo data can be found at "example/loadFootballDb.cypher". To load the script's content, you can open the database browser by clicking the "Open" button at the top of the Neo4j Desktop application. Once the browser is open, you can copy and paste the content of the script into the input area. After executing the script, the entities will appear in the database.

Neo4j database Docker Another alternative for creating a new instance of the NEO4J database is by initializing a Docker container. To accomplish this,



Figure 6.3: Uploading the script

the Docker Desktop application needs to be running, and the following command should be executed:

```
"docker run -p7474:7474 -p7687:7687 -e NEO4J_AUTH=neo4j/password neo4j".
```

After the container is created, the Neo4j browser becomes accessible at localhost (url <http://localhost:7474/browser/>). The browser can be used to load scripts similar to the previous step.

Backend

The backend application already includes a preconfigured `spring-boot-docker.jar` file, so there is typically no need to rebuild the project. However, in certain cases, rebuilding the project can be useful. To rebuild the jar file, the user can navigate to the backend application located at the path `'backend/app'` and execute the following command: `'mvn clean package'`. The backend application is built using the Maven build tool, so Maven can be used to rebuild the jar files.

Alternatively, if you are using IntelliJ IDEA, you can leverage its build-in support for Maven projects. To import the Maven project, you can click on the `pom.xml` file and choose the option "Add as Maven Project". This will enable user to build and manage the project using IntelliJ IDEA's Maven integration. Please note that during this process, reloading of the `pom.xml` file may be required in order to install all the necessary dependencies.

Rebuilding can be required if in the user set the password to anything than default "password". In this case user should navigate to

```
"backend/app/src/main/resources/application.properties"
```

file and set the `spring.data.neo4j.password` property to the password used during database creation.

Shortly after that, a menu bar with script options on the right side will appear. The user can initiate the rebuilding of the jar file by executing the "clean" script followed by the "package" script.

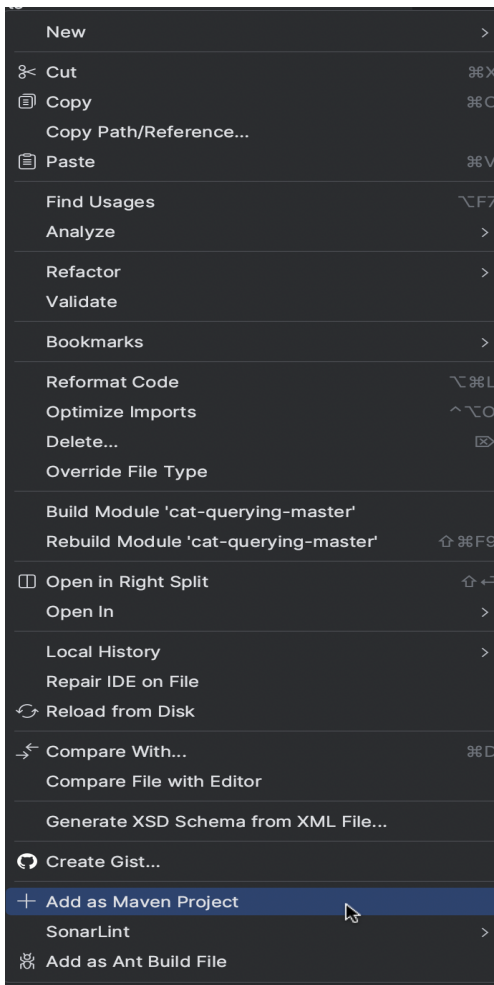
The jar file can be executed either through an IDE or by running the following command from the "backend/app" path: `"java -jar target/spring-boot-docker.jar"`. After successfully launching the jar file, the Spring Boot server will

```

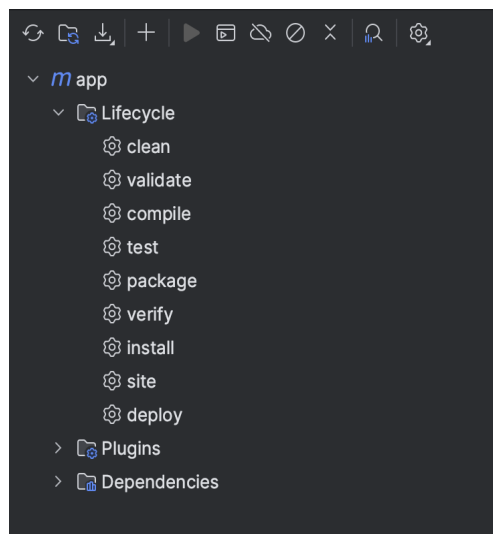
application.properties x
1 spring.data.neo4j.uri=bolt://localhost:7687
2 spring.data.neo4j.username=neo4j
3 spring.data.neo4j.password=password
4 dbms.security.auth_enabled=false
5 # debugging neo4j repository
6 logging.level.org.neo4j.driver.GraphDatabase = debug
7 logging.level.org.neo4j.driver.Driver = debug
8 logging.level.org.neo4j.driver.OutboundMessageHandler = debug
9 logging.level.org.neo4j.driver.InboundMessageDispatcher = debug

```

Figure 6.4: Changing the DB Password



(a)



(b)

Figure 6.5: Import Maven Project (a) Maven Menu (b)

start listening on port 8080. This means that the backend application is now ready to accept incoming requests.

Frontend

The frontend application is developed using JavaScript and utilizes React and React Flow libraries. The project can be found in the frontend repository. To

run the frontend portion of the application, you need to navigate to the "frontend/app" directory and execute the "npm install" command to install all the necessary dependencies. Once the installation is complete, you can execute the "npm start" command to run the React frontend application. Once the application is running, it is available on port 3000. Worth mention that changing the port of the application can result in communication error, do not do this.

6.1.4 Interface of the web editor

In the web interface, users have the ability to create and edit new queries. The application interface consists of three main parts: the editor sandbox, header and footer. The header section includes such parts as a query preview, send preview request button and console button. The preview window is placed in the upper left-hand corner of the screen. The preview window provides user with a way to preview a query before executing it. This feature helps prevent users from sending meaningless queries to the server. To send a preview request, users can click the button with the eye icon on it located in the upper right-hand corner. Also, there is a console open button located near it. The console serves as a response viewer, allowing users to review all the response they receive.

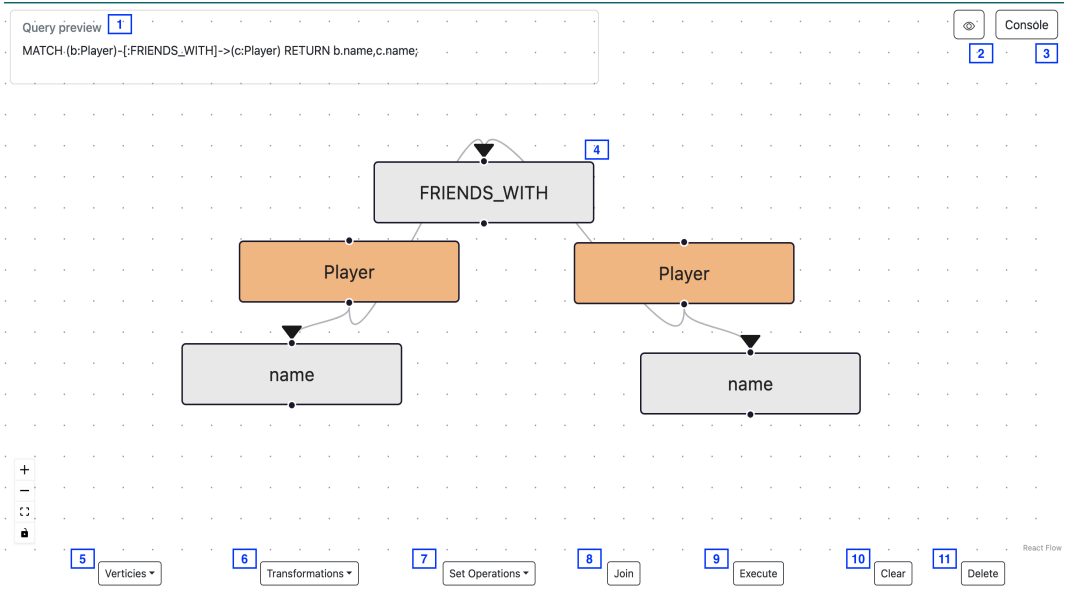


Figure 6.6: 1. Preview window 2. Send preview request button 3. Show console with outputs 4.

The footer of the web interface contains tools for adding and deleting nodes. Creating new entities is possible with buttons located in the bottom part of window. The "Vertices" button can be used to create basic projection and selection node. Created node can be edited later. Node editor can be open by double clicking on a node. Node editor allows user to change label, set conditional and set if the node is projection or not. Next button is transformations. Transformation operations are order by, group by, skip and limit. These operations have default behavior and requires one parent node to be connected with. Next block of operations is set operations with only one currently supported operation available,

which is Union. The union set operation requires two parent nodes and has the default behavior. This operation combine two sub queries. Near the set operation button is join button, which represents Cypher graph traversal and requires two parent nodes to be connected with. The execute button, located in the lower right corner, sends the request to server and saves the response to the console. Clear button can be used to clear whole editor sandbox and delete button is used to remove individual part of the graph.

Vertices creation

To create a vertex, the user should access the vertices menu and select one of the available options. The user can choose to create either a basic vertex or a projection vertex.

After selecting the vertex type, the user needs to specify the name of the vertex. Once the label name is set, the vertex will be created.

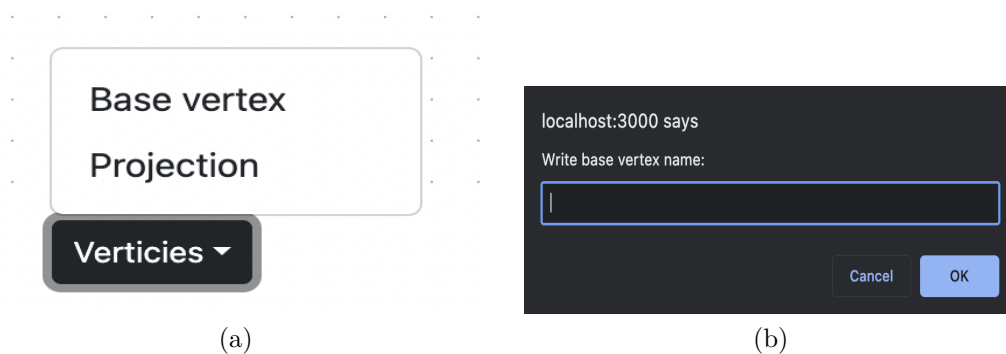


Figure 6.7: Vertex Creation (a) Set Label Dialog (b)

Vertices connection

With the exception of conditional vertices, it is possible to connect every vertex to each other using edges. To create an edge, the user can click on the dot located at the top or bottom of a vertex and, while holding the right mouse button, connect it to the desired vertex. The bottom dot represents the output, while the top dot represents the input, as the graph follows an oriented graph structure.

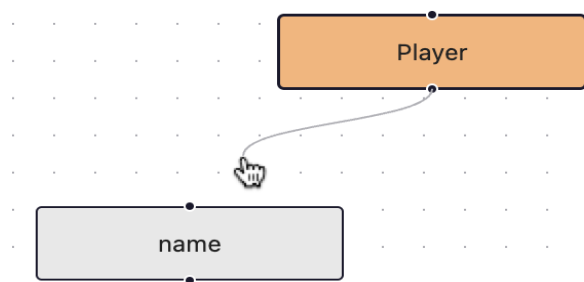


Figure 6.8: Connect Vertices

Vertices and Edge editing

The user is able to edit a node or an edge by double clicking on it. Once user clicks, the editor menu appears. In the node editor menu the user is able to set new node label, set new conditional and change the vertex type from non-projection to projection. The edge editor allows user to change edge label and its type.

By double-clicking on a node or an edge, the user can access the editing functionality. This action triggers the appearance of the editor menu. In the node editor menu, the user can modify the node by setting a new label, defining a new conditional, or changing the vertex type from non-projection to projection. On the other hand, the edge editor enables the user to change the edge label and its type.

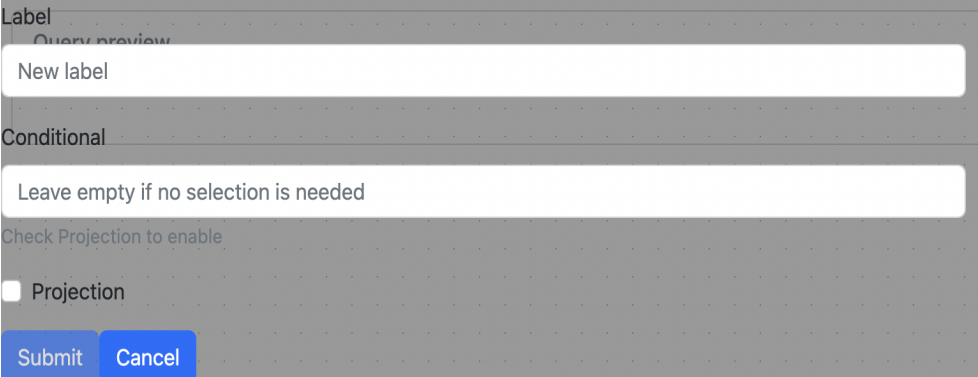


Figure 6.9: Connect Vertices

Transformation Creation

The process of creating a transformation is similar to creating vertices, with the exception of the configuration part. After selecting the desired transformation, the user needs to specify the parent vertex. Depending on the chosen transformation, additional settings may be required. For example, the skip and limit transformations require the user to specify the number of rows to skip or limit. The order by transformation requires the user to set the order by property, while the group by count transformation requires the user to set the group by property and optionally provide a having conditional.

Set Operation and Join Creation

The process of creating a new instance of a set operation or join is similar to creating a transformation, with the exception of the number of required parent vertices. These operations specifically require the user to specify exactly two parent vertices. Once the parents are specified, the join operation will prompt the user to specify the path. The set operations require no additional information except the parents vertices.

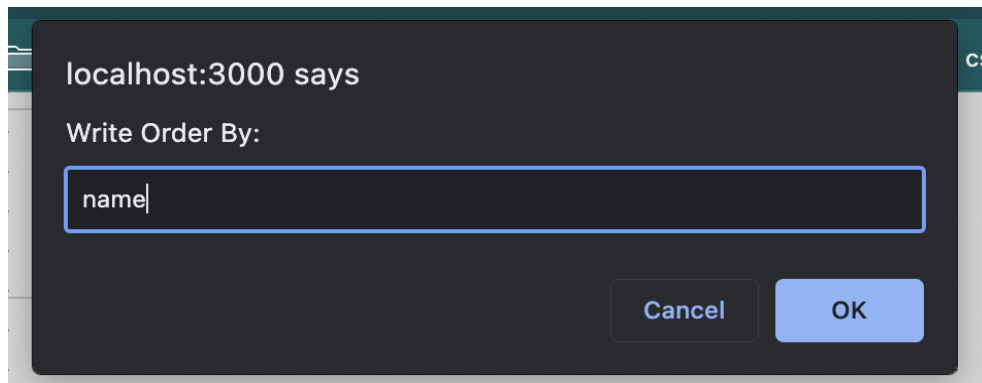


Figure 6.10: Order by name setting dialog

Delete and Clear Operations

The clear operation allows the user to clear the entire editor canvas, removing all elements from the graph. On the other hand, the delete button is used to remove specific elements from the graph. When the delete button is pressed, it turns red to indicate that the delete state is active. The user can then hover over the elements they want to delete, and those elements will become less visible as a visual cue. Clicking on the elements will remove them from the graph. To exit the delete mode, the user simply needs to click on the delete button again. This will remove the red background and return the editor to its normal state. The delete operation supports the deletion of both vertices and edges.

Console

The console tab enables the user to view the output received from the server in chronological order. To access the console tab, the user can click on the node located in the top right corner of the interface.

Conclusion

6.2 Thesis Conclusion

The main objective of the thesis was to develop a user-friendly tool that enables individuals without technical expertise to query multi-model data. To simplify the process, we leveraged the graph representation of unified data stored in Neo4j. The work involved proposing a visual query language (VQL), designing and implementing a web-based editor for creating and modifying VQL queries, as well as designing and implementing the VQL to Cypher query translation algorithm.

The resulting VQL language has less expression power than the Cypher query language, serving as a subset of Cypher. However, during the thesis work, challenges were encountered in translating certain aspects of VQL to Cypher, resulting in some parts of the designed VQL language not being fully translatable yet. Despite this, the VQL language has successfully achieved its objectives and demonstrated its usability for querying multi-model data.

The developed VQL query editor successfully fulfills its goal of being intuitive and understandable to non-expert users. It allows users to construct complex queries using fundamental graph principles.

The translation algorithm demonstrated its capability to successfully translate queries of varying complexity levels. Furthermore, translation process revealed certain limitations of the Cypher query language.

6.3 Future Work

The future improvement can be connected with: expanding the expression power of the designed VQL language

- *Expanding the expression power of the designed VQL language* The expression power of VQL can be extended by implementing support for additional aggregation operations (such as mathematical functions) and incorporating support for Data Manipulation Language (DML) operations, among other features.
- *Integrating tool into management tool family* The tool can be integrated into a comprehensive toolset that encompasses other tools for querying multi-model data, such as MM-quecat [9], in order to provide a unified solution for managing multi-model data.
- *Multi-client application* The tool can be improved to support multiple clients on a single server, enabling the implementation of user accounts and the

ability to save users' history for future reference. This improvement would provide a more personalized and convenient experience for individual users.

Bibliography

- [1] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.
- [2] Jiaheng Lu and Irena Holubová. Multi-Model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.*, 52(3), June 2019.
- [3] Ecma International. JavaScript Object Notation (JSON), 2013. <http://www.JSON.org/>.
- [4] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008.
- [5] Moshé M Zloof. Query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 431–438, 1975.
- [6] Martin Svoboda, Pavel Čuntoš, and Irena Holubová. Categorical Modeling of Multi-model Data: One Model to Rule Them All. In *Proc. of MEDI '21*, pages 190–198. Springer, 06 2021.
- [7] Pavel Koupil and Irena Holubová. A Unified Representation and Transformation of Multi-Model Data using Category Theory. *J. Big Data*, 9(1):61, 2022.
- [8] Daniel Crha. *Unified Querying of Multi-Model Data*. Master thesis, Charles University, Czech Republic, 2022.
- [9] Pavel Koupil, Daniel Crha, and Irena Holubová. MM-quecat: A Tool for Unified Querying of Multi-Model Data. In *Proceedings of the 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece*, pages 831–834. OpenProceedings.org, 2023.
- [10] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C, January 2008. <http://www.w3.org/TR/rdf-sparql-query/>.
- [11] Michael Barr and Charles Wells. *Category Theory for Computing Science*, volume 49. Prentice Hall New York, 1990.
- [12] Andrés Taylor. Cypher Query Language (Cypher), 2011. <https://neo4j.com/docs/cypher-manual/current/introduction/>.
- [13] Marko A. Rodriguez. Gremlin Query Language (Gremlin), 2009. <https://tinkerpop.apache.org>.

- [14] ArangoDb Query Language Query Language (AQL), 2009. <https://www.arangodb.com/docs/stable/aql/>.
- [15] Oren Eini. RavenDb Query Language Query Language (RQL), 2010. <https://www.arangodb.com/docs/stable/aql/>.

List of Figures

1.1	Query that lists all players (a) and the resulting visualization of the query (b)	6
1.2	Query that finds player friends of friends (a) and the resulting visualization of the query (b)	7
1.3	Query that finds English players in Chelsea club (a), result of query as graph (b) and as a table (c)	7
1.4	Grouping players in one team (a) and result of the query (b)	8
1.5	SPARQL query listing all buses (a) and its result (b)	9
1.6	SPARQL query that list all buses (a) and the resulting visualization (b, c)	9
1.7	Grouping and filtering grouped items (a,b) and the results (c,d)	10
1.8	Optional clause	10
1.9	Graph visualization	11
1.10	Initializing new graph	11
1.11	Initializing new traversal	11
1.12	Adding vertices	12
1.13	Adding edges	12
1.14	Aggregate function count()	12
1.15	Gremlin query	12
1.16	Query: Find all edges with contract expiring more than in two years	12
1.17	Depth-first traversal	13
1.18	Gephi initialization	13
1.19	Gephi interface	13
1.20	Creation new collection	14
1.21	Crud operations	15
1.22	Data Projection	16
1.23	Search friends (a) and Search additionally friends of friends (b)	16
1.24	Graph visualisation	17
1.25	GUI Interface	18
1.26	Query examples	18
1.27	Spatial Query Example	19
1.28	Query Result	19
1.29	Map-reduce relationship	19
2.1	Non-projection basic vertex	21
2.2	Projection basic vertex	22
2.3	Simple relation	22
2.4	Simple condition	22

2.5	Alternative text condition	23
2.6	Group by	23
2.7	Group by with having conditionals	24
2.8	Group by with having conditionals	24
2.9	Skip and Limit operations	25
2.10	Union operations	25
2.11	Union operations	26
4.1	Model Class Diagram	33
4.2	Backend Class Diagram	38
4.3	Creating Basic Query Sequence Diagram	42
4.4	Creating Complex Query Sequence Diagram	43
4.5	Backend Model Class Diagram	44
4.6	Query Execution Sequence Diagram	46
5.1	Basic Projection	48
5.2	Basic Projection with Selection	48
5.3	Basic Projection with Selection	49
5.4	Basic Projection with Selection	49
5.5	Basic Projection with Selection	50
5.6	Union	50
6.1	Project Creation	52
6.2	New Local DBMS creation	52
6.3	Uploading the script	53
6.4	Changing the DB Password	54
6.5	Import Maven Project (a) Maven Menu (b)	54
6.6	1. Preview window 2. Send preview request button 3. Show console with outputs 4.	55
6.7	Vertex Creation (a) Set Label Dialog (b)	56
6.8	Connect Vertices	56
6.9	Connect Vertices	57
6.10	Order by name setting dialog	58