TU WIEN Informatics

Realization and Integration of NoSQL Code Generation into bigER

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Marinko Todorovic
Registration Number 11809906

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Vienna, 4th August, 2023

Marinko Todorovic   Dominik Bork

Erklärung zur Verfassung der Arbeit

Marinko Todorovic

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. August 2023

Marinko Todorovic

Acknowledgements

I would like to express my gratitude to my professor Dominik Bork for his invaluable guidance, mentorship, and support throughout the entire process of conducting this thesis. His expertise and insights were instrumental in shaping the direction of this work.

I would also like to extend my thanks to Philipp-Lorenz Glaser for his assistance and collaboration during this project. His contributions and feedback were invaluable in enhancing the quality of this thesis.

Furthermore, I am grateful to the Technical University of Vienna for providing the necessary resources and environment for carrying out this thesis.

Lastly, I would like to thank my family and friends for their unwavering encouragement and support during this academic journey.

Abstract

bigER is a Visual Studio Code extension, which allows for the visual creation of entity relationship diagrams. Once a diagram is created, bigER provides functionality to generate relational SQL code from it. This thesis describes the extension of bigER to also support the generation of NoSql code. As there are different categories of NoSql database systems, we will look at three real world representatives for those categories and how the database schema should be handled for the transformation. For our three categories: document-based, column-family and graph based databases, this thesis will implement code generators for MongoDB, Cassandra and Neo4j respectively. In our approach, we use the inherent structure of how bigER stores its ER diagram as the intermediate logical model that can be transformed into code for the various NoSql databases. In turn, NoSql code can be imported and an entity relationship model will be created. The result of the transformation will be run in the three databases to show, that the code generated is correct and executable.

Contents

Abstract **vii**

Contents **ix**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives and scope . . . 2

2 Literature Review **3**
2.1 Model Driven Architecture . . . 3
2.2 Data Modelling . . . 4
2.3 Entity-Relationship Diagrams . . . 5
2.4 NoSQL databases . . . 6
2.5 Difficulty with Representation . . . 10
2.6 Similar Tools to BIGER . . . 10

3 Technical Background bigER **13**
3.1 Language Server Protocol (LSP) . . . 13
3.2 Sprotty . . . 14
3.3 Xtend . . . 14
3.4 Usage . . . 15
3.5 Code Generators . . . 15
3.6 Metamodel . . . 15

4 Transforming ER diagrams into NoSQL data schemas **19**
4.1 Adding a new Generator . . . 19
4.2 MongoDB . . . 20
4.3 Neo4j . . . 32
4.4 CassandraDB . . . 43
4.5 Summary . . . 52

5 Showcase **55**
5.1 MongoDB Transformation . . . 57
5.2 Neo4j Transformation . . . 66

5.3 Cassandra Transformation 70
5.4 Summary 74

6 Conclusion **75**

Bibliography **77**

List of Tables **81**

List of Figures **81**

A Generating ER-Diagrams with ChatGPT **83**
A.1 What is ChatGPT? 83
A.2 The Prompt 83
A.3 Results 86

CHAPTER 1

Introduction

Model-Driven Architecture (MDA) is an approach to software development based on making models that show the requirements, design, and implementation of a system. MDA's main goal is to keep the concerns of the application domain separate from those of the implementation platform. This lets developers focus on the system's functionality and behavior instead of the low-level technical details of how it works. This makes conceptualizing and maintaining systems easier for developers, as the idea or concept of the system can be expressed independent of the implementation. bigER [35] is a tool that allows for modelling an Entity-Relationship (ER) diagram. These diagrams can then be exported as SQL Creation Script. The ER diagram represents the entities in the system and the relationships between them. By using this diagram, developers can create a visual representation of the data model, which can be used to generate code for a database schema.

1.1 Motivation

The Motivation for the bachelor thesis on the implementation of an ER diagram to NoSQL schema code transformer and it's subsequent integration in the open source project bigER is to address the need for tools to support the design and maintenance of NoSQL databases.

Most database systems have traditionally been relational in nature, which for the better part of 5 decades have been well established solutions for the organization and storage of structured data with fixed database schemas. Yet, NoSQL databases have gained increasing popularity in recent years due to their ability to handle large volumes of data and support of flexible data models, which do not need to adhere to fixed schemas. These capabilities make NoSQL databases a valuable tool for modern applications that require scalable and efficient data storage solutions.

However, there are challenges and trade-offs when choosing NoSQL databases solutions, one of which is the planning and design of the appropriate schema structures that can effectively capture the data and relationships being stored. This especially holds true for developers, who may be more familiar with the relational model and traditional database design tools. Another challenge is the fact that some of the NoSQL databases are not intended to adhere to a fixed database schema like their relational counterparts, and thus can lead to difficulty maintaining consistency during development.

Building on bigER, which already provides ER modeling functionality, an ER diagram to NoSQL schema code transformer could help mitigate those problems and provide massive savings on time by automating the conversion process.

1.2 Objectives and scope

Utilizing a visual modeling tool, such as an Entity-Relationship (ER) diagram, can help to streamline the design process of databases. The entities and relationships in a database are graphically represented by ER diagrams, which are frequently used in RDBMS architecture. However, direct translation of ER diagrams to NoSQL schema code is not supported, so the **goal of this bachelor thesis** is to extend the bigER project by automating the transformation of an ER diagram to various NoSQL database code, which will aid in the design, documentation and creation of NoSQL databases.

We will focus on the following objectives:

1. Exploring the challenges and limitations of transforming ER diagrams into NoSQL data models, as well as importing NoSQL code back into bigER.
2. How to design NoSQL schema code, even though some of those databases are schema-less by nature?
3. Contribution to bigER by adding code generating functionality for document based, column-family and graph based NoSQL databases.

We will approach these objectives in turn, looking at existing literature on the topic of NoSQL databases and their representation, as well as similar tools with which we can potentially compare methods to. Then we will look at the actual implementation of the code generators into bigER [20] and see, how we can translate it's internal logical model of the ER-Diagram representation into their respective NoSql Schema code.

CHAPTER 2

Literature Review

Due to the growing amounts of data [37] being created and stored worldwide, the traditional choice of database systems, namely the relation model, are increasingly becoming inadequate for the management of large amounts of data. This is mainly due to the issue of vertical scaling with relational-database-management-systems (RDBMS), where the servers specification are increased in order to maintain performance with ever growing data, which has a definite limit. Also, the performance decrease when joining data across tables is significant with larger volumes of data.

bigER [20] utilizes Entity-Relationship (ER) modeling [33], which is arguably the standard for data modeling, by providing a visual interface for modeling ER diagrams. It uses the language-server-protocol [10], which allows for the communication of textual language features to the Visual Studio Code (VS Code) client and is extended to also support graphical editing. Sprotty extends the server with graphical language capabilities and interfaces with VS Code through Sprotty VS Code Extensions.

But let's first list of key concepts of data modelling that are used in bigER. Next we will look at the underlying software stack used for bigER and finally we will take a look at NoSQL databases and their implementation, as well as the three popular instances we will provide the export for.

2.1 Model Driven Architecture

Model Driven Architecture (MDA) is a software development approach that aims to separate the functional requirements or specifications of a system from the implementation details in a specific platform [30]. The use of models is emphasized to represent the various aspects of a system, which abstracts away the underlying technology or platform-specific details. The goal of MDA is to improve productivity, portability, and maintainability of

software systems by allowing developers to focus on the core concepts and functionality, while automating the generation of platform-specific code or artifacts.

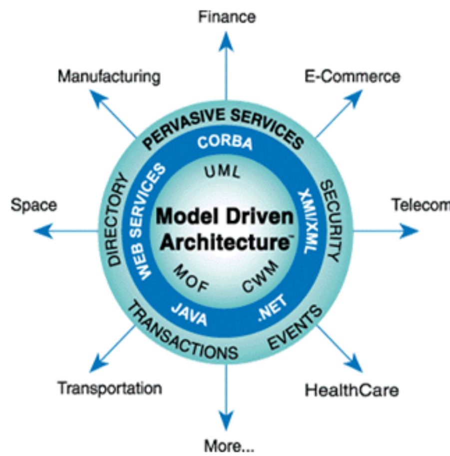


Figure 2.1: Visual Top-Level View on the Model Driven Architecture

bigER applies a similar concept with it's existing SQL code generation ability, where the system's conceptual model is maintained using the more abstract ER diagram, which is platform-independent, but automates the transformation from the conceptual to the physical model or platform-specific implementation, in this case the SQL creation script. Based on that concept, this thesis aims to extend this layer of the MDA approach, by adding support for various NoSQL implementations such as MongoDb, Neo4j and CassandraDb.

2.2 Data Modelling

Data modeling [5] is a way to describe a software system using entity relationship diagrams (ERD), which show how the data structures in a company's database are set up in a table. It is a very clear statement of the business needs of the company. Data models are used for many things, from high-level conceptual models to logical and physical data models. The entity-relationship diagram is usually used to show how these models work. It is used as a guide by database analysts and software developers when designing and putting a system and its database into action.

Conceptual Model A Conceptual Model refers to a high-level representation of entities and their relationships that make up a particular system or data model. The aim of such a model is to give a clear and complete picture of the data needs and how the different parts of the system work together.

Logical Model Logical models are more detailed than conceptual models, and they describe the relationships between data entities in a more specific way. Logical

models show how the database is put together, including the tables, columns, and connections between them. They focus on data modeling and don't worry about the technology used to implement it. This is because logical models are independent of the database system that will be used to create the database.

Physical Model The most detailed model is the physical model, which is used as a blueprint for realizing the logical model into a certain database management system. It extends the Logical Model by specifying the types of the columns. Some database structures, like indexes, partitions, and constraints, are part of physical models. They are built based on logical models and give all the information needed to build the real database system. Physical models show the exact database technology that will be used to build the system.

2.3 Entity-Relationship Diagrams

Peter Chen introduced his design of Entity-Relationship (ER) modelling in his 1976 paper[33]. ER modeling is a popular method for representing the data structure in relational databases. It provides a conceptual and visual framework for comprehending the relationships between various data types (entities) and their attributes. ER diagrams are effective tools for designing, analyzing, and communicating the structure of a database system because they are graphical representations of these models.

2.3.1 Fundamental Components of ER Diagrams

The ER diagram has a few fundamental parts to it.

Figure 2.2: Basic example of a simple er diagram showing its components.

1. **Entities**: An entity is a type of real-world thing, like a person, a product, or an event, that stands for itself. In a computer, an entity is the same as a table that stores information about an item. Entities are usually shown in ER diagrams as circles with their names written inside.

2. **Relationships**: They describe how entities relate to each other. While in a typical relational database system, relationships are expressed by having foreign keys in the table pointing to some target table. In an ER diagram, the relationships connect to the related entities via a diamond shape and lines. The lines indicate cardinality constraints, such as $1:1$, $1:n$ or $n:m$.

3. **Attributes**: Attributes are the features or traits of an entity that tell us more about it. In a computer, columns in a table are like characteristics. In ER diagrams, traits are often shown as ovals that are linked to the things they belong to.

4. **Primary Keys** and **Foreign Keys**: Primary keys uniquely identify records or rows in a table, while foreign keys references records of other tables. In ER diagrams, keys are often underlined or marked with a special symbol, while foreign keys are shown by lines that link entities.

2.4 NoSQL databases

NoSQL databases, or "Not Only SQL", are a new type of database which are designed to handle massive amounts of data across distributed servers. This data may also not be structured in a relational way like in the relational models. This allows for a more flexible and scalable way to store unstructured data such as log files, data gathered from sensors and other types of data that do not have to be fit into a tabular form. This is because most of the NoSQL models do not require a fixed schema for their data, as supposed to their relational counterparts. So changing data formats does not necessarily hinder further use of the database without reconfiguration of the schema.

These NoSQL databases come in a variety of forms, each with special qualities and applications. Here are quick descriptions of the top three NoSQL databases:

2.4.1 Document-Oriented MongoDB

A document-oriented database called MongoDB [15] is made to hold data in the form of documents that resemble the JSON file format.

MongoDB is, as it is described on their website: "a general-purpose document database designed for modern application development and for the cloud. Its scale-out architecture allows you to meet the increasing demand for your system by adding more nodes to share the load." [16]

Large amounts of data that must be quickly accessed and adjusted, like user activity data or real-time analytical data, are frequently stored in this way. Due to its adaptability and scalability, MongoDB is frequently employed in modern web applications.

MongoDB makes it easy and natural for developers to map to objects in their application code. These documents are stored and grouped in **collections**, which can be thought of as a database table in a SQL database, and the documents **fields** represent the fields of the table. But in contrast to a table in a relation database, a collection is far more flexible as it does enforce any fixed structure or schema, at least not by default. In the later sections, we will look into the mechanism at our disposal with which we can enforce a schema onto collections. We will use this similarity as our basis for our transformation of the ER-model to MongoDB.

MongoDB stores data in flexible, JSON-like documents. An example of a simple command to insert a document into a collection might look like this:

```
1 db.collection('users').insertOne({
2   name: 'John Wick',
```

```
    age: 30,
    email: 'john@example.com'
});
```

And a query might look like this:

```
db.collection('users').find({ age: { $gt: 20 } });
```

This would return all users who are older than 20.

2.4.2 Column-Family CassandraDB

Apache's CassandraDB is an open-source, distributed, NoSQL database that is used to manage enormous volumes of data across numerous servers. Because of its high scalability, it is ideal for storing data that must be promptly retrieved, like in real-time applications and it is also a popular option for data storage in cloud applications. Cassandra supports it's own query language called Cassandra Query Language (CQL), which has SQL-like syntax and allows to create and update the databases schema, as well as access and manipulate it's data.

To illustrate the structure of Cassandra, we can see from [30] that Cassandra is comprised of the top-level container called **Keyspace**, which is comparable to the SQL's database and can be thought of as the tuple **(N, F)**, where N is the keyspace's name and F is a set of **column families**, which can be thought of as the database tables with it's fields. Although there are many similarities syntactically between Cassandra and SQL, one fundamental difference is the absence of foreign keys in Cassandra. Keeping this in mind, we will still keep the foreign keys to different tables in order to prevent loss of information and users can join tables on the client side. However the user is free to add tables that reflect the queries of its application, which is the recommendation when working with Cassandra.

Here is an excerpt of how the Cassandra database is structured from the official documentation [3]:

1. Keyspace: Defines how a dataset is replicated, per datacenter. Replication is the number of copies saved per cluster. Keyspaces contain tables.
2. Table: Defines the typed schema for a collection of partitions. Tables contain partitions, which contain rows, which contain columns. Cassandra tables can flexibly add new columns to tables with zero downtime.
3. Partition: Defines the mandatory part of the primary key all rows in Cassandra must have to identify the node in a cluster where the row is stored. All performant queries supply the partition key in the query.
4. Row: Contains a collection of columns identified by a unique primary key made up of the partition key and optionally additional clustering keys.

5. Column: A single datum with a type which belongs to a row.

Cassandra, as a column-oriented database, is designed for scaling and storing large amounts of structured data. A simple command to create a table and insert data might look like this:

```
CREATE TABLE users (
  user_id int PRIMARY KEY,
  name text,
  age int,
  email text
);

INSERT INTO users (user_id, name, age, email) VALUES (1, 'John Wick', 30, 'john@example.com');
```

And a query, similar to SQL, could look like this:

```
SELECT * FROM users WHERE age > 20;
```

This returns all users who are older than 20.

2.4.3 Graph Database Neo4j

Lastly, the graph database called Neo4j is used to hold information as nodes and relationships. It is ideal for storing relational data, such as recommendation or social network data. Neo4j is a one of the more popular option for data analysis and machine learning applications because of its ability to effectively sift through large amounts of data to uncover links and patterns in between nodes.

Neo4j is a graph database, designed for managing and querying highly connected data.

Instead of tables and documents, we use nodes and relationship-edges to express our data. To interact with our data, Neo4j uses it's own language called **cypher**, a declarative query language similar to SQL.

Figure 2.3: Building blocks of the property graph model as in official documentation [18].

Nodes are the entities in our transformation. They have a view important properties which we can use to transform an entity into a node:

1. **Labels** can be used to denote the names of the entity.
2. **Properties** are key-value pairs which will denote the attributes of the entity. In Neo4j properties can hold any number of key-value pairs.

Relationships link two entities together with a directed and named connection:

1. every relationship has a **start** and an **end** node,
2. a **type** can categorize and add meaning to a relationship,
3. a number of **properties**, just like nodes:
   `-[rel:IS_FRIENDS_WITH {since: 2018}]->`,
4. and a **direction** denoted with the arrow.

Although every relationship has a direction, it can be navigated efficiently in any direction.

An example of a simple command to create a node and a relationship might look like this:

```
1 CREATE (john:Person { name: 'John Wick' })-[:KNOWS]->(jane:Person {
    name: 'Jane Wick' });
```

Here, both the nodes **Persons**, as well as the relationship **KNOWS**. These operations however, can be done separately.

A query could look like this:

```
MATCH (john:Person { name: 'John Wick' })-[:KNOWS]->(friends) RETURN
    friends;
```

The result is all people 'John Wick' knows.

Updating a node for a field, could look like this:

```
MATCH (n:Person { name: 'John' })
SET n.age = 35
```

This would set attribute 'age' to 35 in the node of 'John'.

2.5 Difficulty with Representation

Even though the benefits of NoSQL shine in modern settings where big data and scalability of systems are important, the fact that many NoSQL databases don't have a fixed schema makes it hard to show and organize their structures. Traditional relational database systems have rigid structures that work well with the relational model. For example, ER-Diagrams are a good way to show how the entities and their fixed attributes work together. This helps with planning and thinking about the database before it needs to be built. Tools like bigER can help developer teams keep up with the design of the database and easily apply changes and export them by automatically generating the right SQL code from these diagrams to create or extend those databases. But most NoSQL databases allow for variable attributes and mostly don't force rigid schemas, so it could be argued that ER diagrams are not the best way to show them. As I will go into more detail later, this will require a more creative way to limit how the entities are structured, as well as fix attributes so that a schema can still be realized. In this paper [40] from Kwangchul Shin et al. we have an example of how one can go from the conceptual model to a MongoDB diagram. Still, having a conceptual model like the ERD model as an option will help developer teams by showing the domain before concrete implementations and choosing databases, whether they are relational databases or one of the many NoSQL databases.

2.6 Similar Tools to BIGER

Let's take a look at some similiar tools out there, which can be compared to bigER:

1. **LUNA MODELER** [12] is a similar tool that allows for modelling ER-diagrams and generate MongoDB code from it by generating the javascript files that contain the collection creation statements and use validation syntax to define the schema.

2. **Hackolade** [21] is a desktop application, which allows for data modeling through ER-diagrams and subsequent exporting of it as SQL and NoSQL code. It supports a lot of different NoSQL database such as MongoDB, Cassandra and Neo4j. For

MongoDB, the schema is represented via the validation syntax in the collection creation command. Similarly for Cassandra, CQL code is generated, and for Neo4j it generates Cypher code.

3. **DbSchema** [6] is database diagram designer tool which allow to visualize MongoDB databases as diagrams and generate validation rules in the native MongoDB queries. These validation rules ensure that the data added to the collections, is validated according to the schema.

CHAPTER 3

Technical Background bigER

The tech stack utilized is on the one hand the Java-based language server realized in Xtext [9] and Xtend [29]. All the diagramming capabilities are based on Sprotty [24], which adds graphical language features to the server (with the help of sprotty-server) and works with VS Code through Sprotty VS Code Extensions. For a more comprehensive and complete description of all the features, please refer to the official repository [20] or the VS-Code Marketplace [27], as well as Philipp Glasner's thesis on bigER [34].

3.1 Language Server Protocol (LSP)

The Language Server Protocol [10] enables bigER capabilities such as syntax highlighting, go to definition, find all references and code completion. A language server, which is responsible for the implementation of language-specific features, and a client, such as Visual Studio Code, are able to communicate with one another thanks to the LSP, which is a standardized protocol. The introduction of the Graphical Language Server Protocol (GLSP) [38, 31] opens up new possibilities for visualizing and interacting with conceptual models. As demonstrated in a recent paper [32], GLSP facilitates advanced model visualization and interaction functionalities, such as Semantic Zoom and Off-Screen Elements, within the Eclipse GLSP platform.

The LSP and GLSP frameworks are moving towards a more flexible, cloud-based approach from a monolithic and tightly coupled tools. This allows integration with various platforms such as VS Code, Eclipse Theia, and others, where the users experience a more seamless transition between local an cloud-based editing [31]. This trend is becoming the state-of-the-art in software engineering and is also beginning to be reflected in the arena of modeling tools, underlining the long-standing requirement for flexibility in the modeling method itself [39, 36].

3.2 Sprotty

Eclipse Sprotty [25] is an open-source, web-based diagramming system that is a key part of bigER. It is built with current web technologies like TypeScript, SVG, and CSS, and it uses a linear event loop with a virtual DOM to meet the needs of web apps. The framework is designed with the user in mind focusing on a smooth user experience. It has built-in graphics that make changes easy and improve the user experience.

Sprotty allows for remote backends, where one can be viewing data from many different sources, such as database backends and LSP applications. This works well with the bigER extension because it lets the LSP and cloud IDEs that use it work well together. Sprotty uses a simple, extendable JSON format for client-server exchange.

This framework is even more flexible because of its design based on dependency injection, which lets components be changed or added as needed. Because of this, Sprotty is great for a wide range of uses. Sprotty also works with technologies such as the LSP, Xtext [9] language backends based on LSP, the Eclipse Theia IDE framework [19] (and later Eclipse Che), the Eclipse Layout Kernel, and Eclipse RCP [8].

In bigER, Sprotty is the basis for the dynamic graphical ER diagram designer, which makes it easy for users to make and change ER diagrams. When improving bigER to handle the new NoSQL code creation features, it will be important to understand and use Sprotty's features and powers well.

3.3 Xtend

Xtend is a dialect of Java created and maintained by the Eclipse Foundation. It is supported by the Java Virtual Machine (JVM) and can thereby be seamlessly incorporated and any existing Java libraries can be used with it. A few features of Xtend, such as template expressions and lambda expressions, make it very useful and simplify the programming experience.

bigER uses Xtend to generate code, which converts Entity-Relationship (ER) diagrams into the relevant output like SQL code.

The following is a snipped from the sql generator in bigER:

```
1 private def toTable(Entity entity) {
2     return '''
3         CREATE TABLE «entity.name»(
4         «FOR attribute : entity.attributes.reject[it.type==
    ↪ AttributeType.DERIVED]»
5             «'\t'»«attribute.name» «attribute.datatype.
    ↪ transformDataType»,
6         «ENDFOR»
7         «'\t'»PRIMARY KEY («entity.primaryKey.name»)
8         );«'\n'»
9     '''
```

```
10 }
```

This method converts the entities to an actual SQL table using the Xtend language features like the multiline formating. This kind of syntax enables programmers to create more expressive and succinct code for producing the desired output format by allowing the format to be injectable with values and allowing for in-string loops. For our purposes, Xtend will help us a great deal to clearly display how the output of the transformation will look like, without loosing readability (no use for string builder).

3.4 Usage

The graphical ER diagrams are built with **.erd** files, which allow the users to define entities, relationships and their fields and their cardinality. The following depiction of an example **.erd** file is a slightly modified example of the readme found in the official github repo [20]. Entities A, B and C are described, though entity **C** extends the functionality of **B** with the "extends" syntax, as well as the relationship **Rel** between **A** and **B**.

```
1  erdiagram Model
2  notation=default
3
4  entity A {
5     id key
6  }
7  entity B {
8     id key
9  }
10 entity C extends B {
11    id key
12 }
13 relationship Rel {
14    A -> B
15 }
```

3.5 Code Generators

The way those **.erd** files are transformed to e.g. SQL generation code is via the so called "generators", which contain the exact instructions of how the syntax of the output should look like. These can be implemented in either Xtend or Java.

Next we want to implement the actual generators which interpret the ER-Diagrams and output actual code files that contain the NoSql code. BIGER is structured in a way, that allows for extension of generator functionality by writing custom generator files.

3.6 Metamodel

The Metamodel [13] is the result of the .erd file parsed into BIGER. It describes how the data is stored and how we can access it.

- **Model:** Represents the ER diagram and contains a name, an optional NotationOption specifying the ER notation to be used (Chen, Bachman, etc.), and any number of Entity and Relationship instances.
- **Entity:** It has an optional 'weak' keyword indicating if it's a weak entity, a name, an optional 'extends' field for inheritance, and any number of Attributes.
- **Relationship:** It has a name, an optional 'weak' keyword, and any number of Attribute and RelationEntity instances.
- **Attribute:** Is an attribute in an entity or a relationship. It has an optional VisibilityType (public, private, etc.), a name, an optional DataType, and an optional AttributeType (key, optional, derived, etc.).

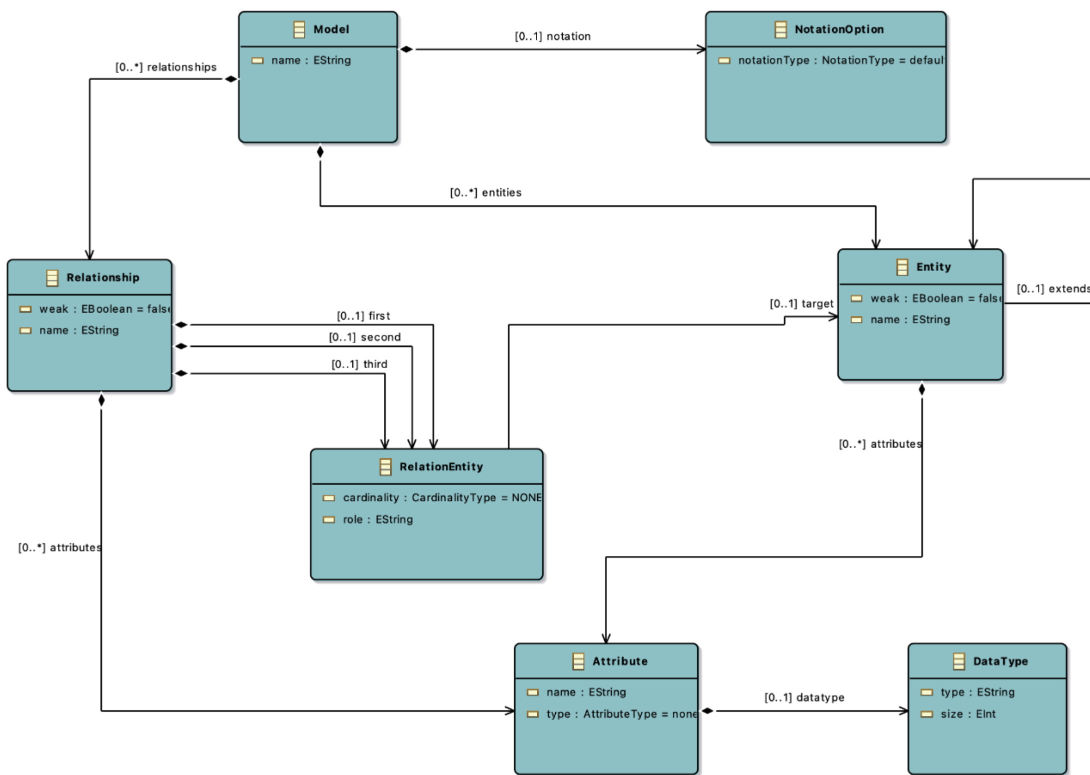
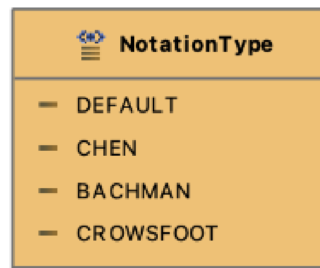
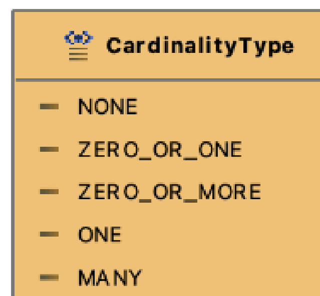


Figure 3.1: The Metamodel of bigER [13]





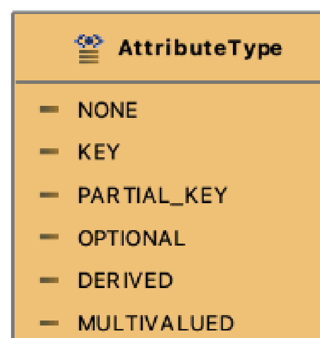


Figure 3.2: The Metamodel of BIGER [13]

CHAPTER 4

Transforming ER diagrams into NoSQL data schemas

As stated before, most NoSQL databases do not enforce Schemas like most of the relational database systems. But in order to represent our ER-Diagram in these NoSQL, some sort of Schema is exactly what we want so we can generate code to represent the diagram which can be imported into the physical NoSQL database. As we will see below, we do have three different cases for our databases. MongoDB does not encourage setting a Schema, but does have functionality to enforce one, as we will see below. Neo4j on the other hand, our approach is different. Here, there is not a fixed Schema to be enforced, but rather we want to show the ER model as a graph itself in order to be used as an example of how entities and relationships are to be represented. And lastly, Cassandra does, very similarly to normal SQL databases, support creation statements for tables and it's fields. The big difference is the lacking support of foreign keys, but our approach does contain foreign keys anyways which indicate relationships.

4.1 Adding a new Generator

In order to implement the logic for the actual transformation of the logical model to the physical target model, we need to implement a Code Generator.

These Generators can be implemented either in Java or Xtend and are located within the `org.big.erd.generator` package.

The IErGenerator interface and its method need to be implemented which looks like this:

```
1 class NewGenerator implements IErGenerator {
2
3   override void generate(Resource resource, IFileSystemAccess2 fsa,
    ↪ IGeneratorContext context) {
```

```
4          val model = resource.contents.get(0) as Model
5          validate(resource, model)
6          val fileName = (model.name ?: 'output') + ".js"
7          fsa.generateFile(fileName, generate(model))
8      }
9
10     def String generate(Model model) {
11        // map model to generated code...
12     }
13 }
```

The Instance of Resource provides access to the whole model. Files can be created by using the provided IFileSystemAccess2. After the basic implementation of the interface, we need to provide a command, so that the generator can be executed. We can do this in the **ErCommandService** class located in the **org.big.erd.ide.commands** package, by adding our command to the hash map where the commands for their corresponding generators are being stored. For further details on how to integrate and fully connect generators to the rest of bigER, please refer to the Wiki chapter[26] of the official repository on new generators.

4.2 MongoDB

Our first NoSQL database we will look at is MongoDB [15], the document-based probably the most popular NoSQL database in our export suite. MongoDB is not strictly schema less but uses rather a flexible schema model, meaning that different documents in a collection don't necessarily need to have the same fields or data types.

4.2.1 Schema Validation in MongoDB

We can use schema validation [22], which exist to ensure there are no unintended improper data types once a schema is established or agreed upon. This is done by defining validation rules. These rules can be defined when creating a collection and adding the logic of the validation as a parameter:

```
1  db.createCollection("students", {
2     validator: {
3        $jsonSchema: {
4           bsonType: "object",
5           title: "Student Object Validation",
6           required: [ "address", "major", "name", "year" ],
7           properties: {
8              name: {
9                 bsonType: "string",
10                description: "'name' must be a string and is required"
11             },
12             year: {
13                bsonType: "int",
```

```
14                  minimum: 2017,
15                  maximum: 3017,
16                  description: "'year' must be an integer and is
    ↪ required"
17              },
18              gpa: {
19                  bsonType: [ "double" ],
20                  description: "'gpa' must be a double if the field
    ↪ exists"
21              }
22          }
23      }
24  }
25 } )
```

In this example from the MongoDB documentation, a students collection is created and the $jsonSchema operator is used to set schema validation rules. Even though the field "address" is not defined in the properties, that field's is required (in the required array) but one can set it as any type desired. These rules get get checked, whenever a new record or document gets inserted to the collection:

```
1 db.students.insertOne( {
2    name: "Alice",
3    year: Int32( 2019 ),
4    major: "History",
5    gpa: Int32(3),
6    address: {
7       city: "NYC",
8       street: "33rd Street"
9    }
10 } )
```

This would return a validation error, because our above schema validation defined the field "gpa" to be of type double.

With this functionality, we can translate the entites and relationships of the ER-Diagram from BIGER to add all the entities and relationships with this $jsonSchema validation.

4.2.2 Mapping Tables

Mapping an ER diagram to MongoDB involves translating the entities, relationships, and attributes in the ER diagram into collections, documents, and fields in MongoDB. Here's a simple mapping table:

ER Diagram	MongoDB
Entity	Collection
Relationship	Collection
Attribute	Field in a Document

Table 4.1: Mapping from ER diagram to MongoDB

Consider an example ER diagram with two entities, ‘Room‘ and ‘Building‘, and a 1:N relationship between them (one Building can have many rooms):

ER Diagram	MongoDB
Room	Room Collection
Building	Building Collection
has (1:N)	Room Document in Room Collection has a Building ObjectID

Table 4.2: Example Mapping from ER Diagram to MongoDB

This example in the .erd syntax:

```
weak entity Room {
    room_nr: INT partial-key
}
entity Building {
    building_id: CHAR(8) key
    address: VARCHAR(255)
}
weak relationship has {
    Room[N] -> Building[1]
}
```

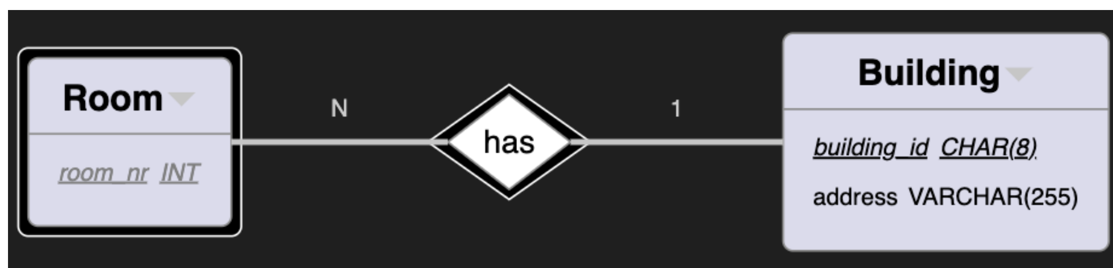


Figure 4.1: The simple ER-diagram example for Transformation.

This should map to the following MongoDB code:

```

db.createCollection("Room", {
  validator: {
```

```
4      $jsonSchema: {
5        bsonType: "object",
6        title: "Room Object Validation",
7        required: ["room_nr"],
8        properties: {
9          room_nr: {
10           bsonType: "int"
11         }
12       }
13     }
14   }
15 });
16
17 db.createCollection("Building", {
18   validator: {
19     $jsonSchema: {
20       bsonType: "object",
21       title: "Building Object Validation",
22       required: ["building_id"],
23       properties: {
24         address: {
25           bsonType: "string"
26         },
27         building_id: {
28           bsonType: "string"
29         }
30       }
31     }
32   }
33 });
34
35 db.createCollection("has", {
36   validator: {
37     $jsonSchema: {
38       bsonType: "object",
39       title: "has (relationship) Object Validation",
40       required: ["room_nr", "building_id"],
41       properties: {
42         room_nr: {
43           bsonType: "int",
44         },
45         building_id: {
46           bsonType: "string",
47         },
48       }
49     }
50   }
51 });
```

In practice, the schema with populated data could look like this:

```
1
2  // Building Collection
3  {
4    building_id: ObjectId('507f1f77bcf86cd799439011'),
5    address: 'main street 1',
6    rooms: [ ObjectId('507f191e810c19729de860ea'), ObjectId('507
     ↪ f191e810c19729de860eb') ]
7  }
8
9  // Room Collection
10 [
11   {
12     _id: ObjectId('507f191e810c19729de860ea'),
13     room_nr: ObjectId('507f191e810c19729de860ea'),
14     room_nr: 1
15   },
16   {
17     _id: ObjectId('507f191e810c19729de860eb'),
18     room_nr: 2
19   }
20 ]
21
22 // has Collection
23 [
24   {
25     _id: ObjectId('793f191e810c19729de86024'),
26     building_id: ObjectId('507f1f77bcf86cd799439011'),
27     room_nr: 1
28   },
29   {
30     _id: ObjectId('793f191e810c19729de86025'),
31     building_id: ObjectId('507f1f77bcf86cd799439011'),
32     room_nr: 2
33   }
34 ]
```

However, as we are not dealing with data only, but the ER model itself, our schema validation will not support the reference of other objects, but only indicate foreign key relationships semantically.

4.2.3 Implementing the MongoDB Export

The transformation to MongoDB code will generate a collection for every entity and every relationship. They should contain all the fields, as well as all the primary keys linking the entities in the relationships. Extended entities will be flattened, which means their attributes will be incorporated into the extended entity and subsequently into the collections schema.

We start by implementing the **generate(Model model)** method. Here we provide the top layer structure of the output document which is comprised of firstly the entities and then the relationships:

```
1  def String generate(Model model) {
2      '''
3          use(«model.name»);
4          «FOR entity : model.entities»
5              «entity.toTable»
6          «ENDFOR»
7          «FOR relationship : model.relationships.reject[!it.isWeak]»
8              «relationship.weakToTable»
9          «ENDFOR»
10         «FOR relationship : model.relationships.reject[it.isWeak]»
11             «relationship.toTable»
12         «ENDFOR»
13     '''
14 }
```

The active ER-Diagram is described by the Metamodel 3.6 of BIGER and is passed to the generate method, so we can access it's entities relationships.

In MongoDB the **use(myDatabase)** statement is used to select the database you want to work with. Next we iterate over all entities by using the **FOR** syntax of Xtend, which iterates over all of the entities. In the body of the loop we have the method **toTable()** is called and the current entity is passed as a parameter, written in Xtend syntax.

In our **MongoDbGenerator**, we have multiple **toTable()** methods for outputting entities and relationships:

```
1  private def toTable(Entity entity) {
2  return '''
3          db.createCollection("«entity.name»", {
4              validator: {
5                  $jsonSchema: {
6                      bsonType: "object",
7                      title: "«entity.name» Object Validation",
8                      required: ["«entity.primaryKeys.map[key | key.
   ↪ name].join(', ')»"],
9                      properties: {
10                         «FOR attribute : entity.getAllAttrWithExtends
   ↪ .reject[it.type === AttributeType.DERIVED] SEPARATOR ','»
11                         «attribute.name»: {
12                             bsonType: "«attribute.datatype.
   ↪ transformDataType»"
13                         }
14                         «ENDFOR»
15                     }
16                 }
17             }
```

```
18          });
19          «'\n'»«'\n'»
20      '''
21  }
```

For the entities, the **toTable()** method outputs a string of a **createCollection** statement, which creates a collection with the entities name passed as the first parameter, as well as a definition of the schema validator passed as the second parameter.

The validation starts with defining the bsonType as "object" and the title. The required field defines, as the name implies, fields cannot be missing in the creation of the collection. As required fields we define the primary key of the entity. The primary key can be accessed through the method **primaryKeys()**:

```
1  private def Iterable<Attribute> primaryKeys(Entity entity) {
2      val keyAttributes = entity.attributes?.filter[it.type ===
   ↪ AttributeType.KEY]
3      // TODO: Fix this
4      if (keyAttributes.nullOrEmpty) {
5          return entity.attributes
6      }
7
8      return keyAttributes
9  }
10
11 private def getAllKeysName(Relationship relationship) {
12     return '''«IF relationship.first?.target !== null»"«relationship.
   ↪ first?.target.primaryKeys.map[key | key.name].join(', ')»"«
   ↪ ENDIF»«IF relationship.second?.target !== null», "«relationship
   ↪ .second?.target.primaryKeys.map[key | key.name].join(', ')»"«
   ↪ ENDIF»«IF relationship.third?.target !== null», "«relationship.
   ↪ third?.target.primaryKeys.map[key | key.name].join(', ')»"«
   ↪ ENDIF»'''
13 }
```

In order to make the keys more easily displayed side by side, we define the method **getAllKeysName()**.

Finally the properties of the fields are listed and defined. For this we iterate with the same for each syntax to iterate over the Attributes of the entity. These then get listed and defined each with the attributes name and it's datatype which is declared as the bsonType. The datatype is transformed with the method **transformDataType**, which maps common SQL data types to the corresponding MongoDB data types:

Table 4.3: Mapping of MongoDB Data Types to SQL Data Types

MongoDB Data Type	SQL Data Type
string	VARCHAR or CHAR

```
private def transformDataType(DataType dataType) {
    // default
    if(dataType === null) {
        return 'string'
    }

    val type = dataType.type.toLowerCase()

    if(type.contains('char') || type.contains('string')) {
        return 'string';
    }

    return type
}
```

Notice as well, the **getAllAttrWithExtends** method in **toTable** in the FOR statement above:

```
private def Iterable<Attribute> getAllAttrWithExtends(Entity entity)
    ↪ {
    val attributes = newHashSet
    attributes += entity.attributes
    if (entity.extends !== null) {
        attributes.addAll(getAllAttrWithExtendsWithNamePrefix(entity.
    ↪ extends))
    }
    return attributes
}
```

Here we collect all the entities attributes into the **attributes** hashset. Furthermore, we check at the end if the entity extends another. Our method to deal with this issue is to flatten these extends relationships by aggregating all the attributes in the extends chain. To do so, we define the method **getAllAttrWithExtendsWithNamePrefix** where we pass the entity which our entity extends:

```
private def Iterable<Attribute> getAllAttrWithExtendsWithNamePrefix(
    ↪ Entity entity) {
    val attributes = newHashSet
    for (attr : entity.attributes) {
        if (!attr.name.startsWith(entity.name)) {
            attr.name = entity.name + '_' + attr.name
        }
        attributes += attr
    }
    if (entity.extends !== null) {
        attributes.addAll(getAllAttrWithExtendsWithNamePrefix(entity.
    ↪ extends))
    }
    return attributes
}
```

It is a recursive method that goes through all the extended entities and collects and prefixes the attributes names with the current entities name. For each of the new entities attributes we check if it's name already starts with the enities name in order to avoid repeating prefixes when going through already prefixed attributes. Finally, all the attributes are returned and iterated over to list out all the properties in the correct JSON validation syntax.

Similarly for relationships there is an overload of the method **toTable()** with relationship as the parameter. A slight difference, the **reject[!it.isWeak]** which separates non weak relationships, so as to prioritize them in order before the weak ones:

```
«FOR relationship : model.relationships.reject[!it.isWeak]»
    «relationship.weakToTable»
«ENDFOR»
«FOR relationship : model.relationships.reject[it.isWeak]»
    «relationship.toTable»
«ENDFOR»
```

Differently as well, the required fields lists now all the keys names. Additionally, we have a second FOR loop which iterates all Keys before the relationship attributes are listed:

```
private def toTable(Relationship relationship) {
    return '''
            db.createCollection("«relationship.name»", {
                validator: {
                    $jsonSchema: {
                        bsonType: "object",
                        title: "«relationship.name» (relationship)
    ↪ Object Validation",
                        required: [«relationship.getAllKeysName»],
                        properties: {
                            «FOR attribute : relationship.
    ↪ getAllKeysNameArray»
                            «attribute.name»: {
                                bsonType: "«attribute.datatype.
    ↪ transformDataType»",
                            },
                            «ENDFOR»
                            «FOR attribute : relationship.attributes»
                            «attribute.name»: {
                                bsonType: "«attribute.datatype.
    ↪ transformDataType»",
                            },
                            «ENDFOR»
                        }
                    }
                }
            });
            «'\n'»«'\n'»
```

```
25          '''
26 }
```

One important difference between the entity and relationship creation statements is that the relationship's title contains the fixed phrase **(relationship)** in order to differentiate it from the entities. This will be important in the **Import** 4.2.4 implementation. Furthermore, the method **getAllKeysName** gets all primary keys of the entities connected through the relationship:

```
1 private def getAllKeysName(Relationship relationship) {
2     return '''«IF relationship.first?.target !== null»"«relationship.
    ↪ first?.target.primaryKey.name»"«ENDIF»«IF relationship.second?.
    ↪ target !== null», "«relationship.second?.target.primaryKey.name
    ↪ »"«ENDIF»«IF relationship.third?.target !== null», "«
    ↪ relationship.third?.target.primaryKey.name»"«ENDIF»'''
3 }
```

It returns a string with these keys separated by comma. In contrast, **getAllKeysNameArray** compiles all these primary keys as an array and returns it, so it can be iterated over:

```
1 private def getAllKeysNameArray(Relationship relationship) {
2     val keys = newHashSet
3     if (relationship.first?.target !== null) { keys += relationship.
    ↪ first?.target?.primaryKey }
4     if (relationship.second?.target !== null) { keys += relationship.
    ↪ second?.target?.primaryKey }
5     if (relationship.third?.target !== null) { keys += relationship.
    ↪ third?.target?.primaryKey }
6     return keys
7 }
```

The results of this transformation is shown in Showcase with the **university.erd** example.

4.2.4 Implementing the MongoDB Import

In order to be able to import NoSQL schemas into BIGER, we had to include the phrase "**(relationship)**" into the title of the relationship creation statement. But now, we can differentiate between entities and relationships, using regular expression we can parse our exported NoSQL code and retrieve the most important of the information we had before. To do this, we will construct a new .erd file with all the information we can extract from the NoSQL code.

In future work, this can be extended where more information could be embedded into the exported code to extract more precisely the original diagram.

The heart of our import functionality are the regular expression patterns, we define them as follows:

```
1     private static final Pattern ENTITY_PATTERN = Pattern.compile("db
    ↪ \\.createCollection\\(\"([^\"]+)\",");
2     private static final Pattern FIELD_PATTERN = Pattern.compile("
    ↪ (?<!properties)([a-zA-Z_]+)(?<!jsonSchema):\\s*\\{\\s*bsonType
    ↪ :\\s*\"([^\"]+)\"");
3     private static final Pattern REQUIRED_FIELDS_PATTERN = Pattern.
    ↪ compile("required:\\s*\\[([^\\]]+)\\]");
4     private static final Pattern TITLE_FIELDS_PATTERN = Pattern.
    ↪ compile("title:\\s*\"([^\"]*\\brelationship\\b[^\"]*)\"");
```

We search for the overall creation statement with the **ENTITY_PATTERN**, the fields of the entity or relationship with **FIELD_PATTERN**, all the required fields, that as we specified in the export, denote our primary key, with **REQUIRED_FIELD_PATTERN** and finally we search for our **(relationship)** phrase with **TITLE_FIELD_PATTERN**.

The following code implements the transformation of a MongoDB .js file into a structure that can be expressed as a .erd file:

```
1  public static void main(String[] args) {
2      if (args.length != 1) {
3          System.out.println("Usage: Java MongoDBInterpreter <file_path>");
4          return;
5      }
6
7      StringBuilder output = new StringBuilder();
8      String filePath = args[0];
9      Map<String, Map<String, Tuple<String, Boolean>>> entities = new LinkedHashMap<>();
10     Map<String, Map<String, Tuple<String, Boolean>>> relationships = new LinkedHashMap
   ↪ <>();
11
12     Map<String, String> keys_from_entities = new HashMap<>();
13     try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
14         StringBuilder sb = new StringBuilder();
15         String line;
16         while ((line = reader.readLine()) != null) {
17             sb.append(line);
18             sb.append(" ");
19         }
20
21         String content = sb.toString();
22
23         Matcher entityMatcher = ENTITY_PATTERN.matcher(content);
24         while (entityMatcher.find()) {
25             String entityName = entityMatcher.group(1);
26             int entityStart = entityMatcher.start();
27             int entityEnd = content.indexOf("});", entityStart);
28             if (entityEnd == -1) {
29                 entityEnd = content.length();
30             }
31
32             String entityContent = content.substring(entityStart, entityEnd);
33             Matcher fieldMatcher = FIELD_PATTERN.matcher(entityContent);
34
35             Map<String, Tuple<String, Boolean>> fields = new HashMap<>();
36             while (fieldMatcher.find()) {
37                 fields.put(fieldMatcher.group(1), new Tuple<>(fieldMatcher.group(2),
   ↪ false));
38             }
39
40             Matcher requiredFieldsMatcher = REQUIRED_FIELDS_PATTERN.matcher(
   ↪ entityContent);
41             if (requiredFieldsMatcher.find()) {
42                 String[] requiredFields = requiredFieldsMatcher.group(1).replaceAll("\"
   ↪ ", "").split(",");
43                 for (String requiredField : requiredFields) {
```

```
44                      requiredField = requiredField.trim();
45                      if (fields.containsKey(requiredField)) {
46                          fields.put(requiredField, new Tuple<>(fields.get(requiredField)
    ↪ .getFirst(), true));
47                      }
48                  }
49              }
50              Matcher titleFieldsMatcher = TITLE_FIELDS_PATTERN.matcher(entityContent);
51              if (titleFieldsMatcher.find()) {
52                  relationships.put(entityName, fields);
53              }else {
54                  entities.put(entityName, fields);
55                  keys_from_entities.put(fields.entrySet().stream().filter(x -> x.
    ↪ getValue().getSecond()).findFirst().get().getKey(), entityName);
56              }
57          }
58      } catch (IOException e) {
59          e.printStackTrace();
60      }
61
62      // Print entities and their fields
63      for (Map.Entry<String, Map<String, Tuple<String, Boolean>>> entry : entities.
    ↪ entrySet()) {
64          output.append("entity " + entry.getKey() + " {\n");
65          for (Map.Entry<String, Tuple<String, Boolean>> field : entry.getValue().
    ↪ entrySet()) {
66              output.append("\t" + field.getKey() + ": " + translateDataType(field.
    ↪ getValue().getFirst().toLowerCase()) + (field.getValue().getSecond() ? " key" :
    ↪ "") + "\n");
67          }
68          output.append("}\n");
69      }
70      // Print entities and their fields
71      for (Map.Entry<String, Map<String, Tuple<String, Boolean>>> rel : relationships.
    ↪ entrySet()) {
72          output.append("relationship " + rel.getKey() + " {\n");
73          // keyed fields
74          int field_counter = 0;
75          for (Map.Entry<String, Tuple<String, Boolean>> field : rel.getValue().entrySet
    ↪ ()) {
76              if (field.getValue().getSecond()) {
77                  output.append((field_counter++ > 0 ? " -> " : "\t")+ keys_from_entities
    ↪ .getOrDefault(field.getKey(), field.getKey()));
78              }
79          }
80          output.append("\n");
81          for (Map.Entry<String, Tuple<String, Boolean>> field : rel.getValue().entrySet
    ↪ ()) {
82              if (field.getValue().getSecond()) {
83                  output.append("\t" + field.getKey() + ": " + translateDataType(field.
    ↪ getValue().getFirst().toLowerCase()) + "\n");
84              }
85          }
86          output.append("}\n");
87      }
88
89      System.out.println(output);
90  }
```

In this implementation, we split the MongoDB file into parts, each containing one of the create-collection statements. For every statement, we search for the patterns we defined and extract away the entities, its fields and keys. If we find our phrase **(relationship)** in the title, then we add a relationship instead of an entity. We employ the **StringBuilder()** function to concatenate all the parts together and forming the new .erd file.

4.3 Neo4j

As Neo4j is a graph database and for most people is as of jet not a very familiar concept, let's first take a look at the basic structures and workings of Neo4j. We will also explore how we can incorporate it's features to be able to transform and represent our ER-diagrams effectively.

4.3.1 Transformation goals

There are different tools to help developers getting started with Neo4j, that transform an existing relational database into Neo4j. One of the most popular tools is neo4j-etl [17]. Although this tools transformation focuses on the actual data of the database, we do get an abstract view of the schema upon configuration, that would fit our purposes in the bigER transformation:

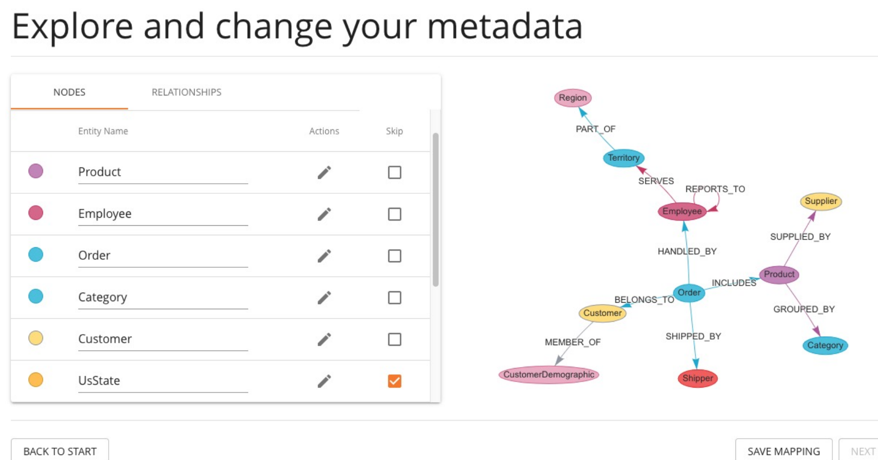


Figure 4.2: Preview of the neo4j-etl transformation tool shows schema as Neo4j graph.

After specifying the database to be imported, the user in neo4j-etl can then choose the coloring and the entity names of the tables which is to be imported. Our goal is to achieve a similar view of all the nodes and relationships to clearly convey the schema of the ER-diagram. In order to work with a Neo4j database, we can either download the Neo4j-Desktop version (https://neo4j.com/download/) or use the web browser version (https://sandbox.neo4j.com/) where we have access to a visual graph viewer.

A few special cases for the transformation of a ER-diagram into a graph database:

1. **Extends relationships** of two entities will be handled twofold. Firstly by adopting all the attributes of the extended entity, and secondly the adding of a **IS_A**-relationship between the nodes of the extended entities.

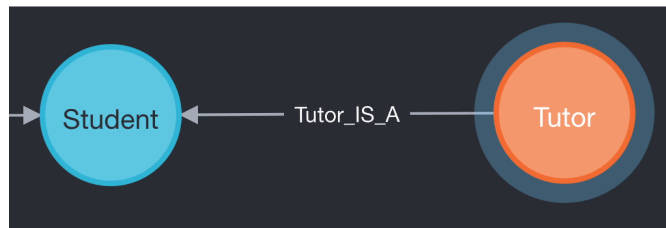


Figure 4.3: The extends relationship mapped in neo4j graph as IS_A relationship.

2. **Ternary relationships** of the ER-diagram, a relationship connected to three entities at the same time, is not supported as such in Neo4j.

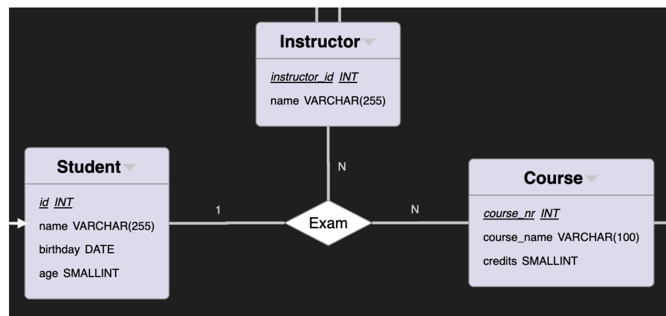


Figure 4.4: Ternary relationship in university example in BIGER.

The way we will denote such a relationship, is by adding an additional node with the name of the relationship, in case of the example above 4.4 called "exam". This node then get's connected to the other three nodes it was connected to in the ER-diagram, and the resulting graph should look like this:

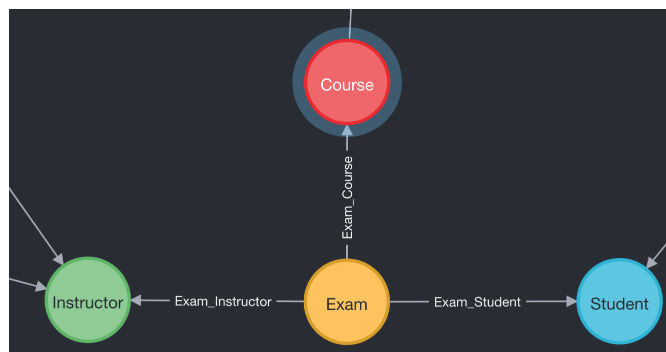


Figure 4.5: Ternary relationship in university example as Neo4j graph.

Regarding the display of the Neo4j graph visualization, we want to display our nodes with their respective entity names, as well as the relationships label with the relationship

names. In order to do that, we will have a special property "name", which will be in the first place of the properties array, and will hold the entities name for nodes and the relationship name for the graph relationship. The rest of the attributes will follow, with their property name prefixed with their entities name and it's value being their data type.

Before we dive into the implementation, the following cypher queries will be useful:

1. `CREATE (Room:Room {name: "Room", Room_room_nr: "INT"})` will create a node with label and a variable number of attributes which are key value pairs.
2. `CREATE (Room)-[Office:Office{name: "Office" }]->(Instructor)` will create a relationship called "Office" with a name attribute, that connects the node "Room" and "Instructor" with the relationship pointing from "Room" to "Instructor".
3. `MATCH ()-[r]-() RETURN r` will return all the nodes and all the relationships of a database and thereby making it possible to see the whole graph of our model.

4.3.2 Mapping Tables

Mapping an ER diagram to Neo4j involves translating the entities, relationships and attributes in the ER diagram into nodes, edges and fields in Neo4j:

ER Diagram	Neo4j
Entity	Node
Relationship	Edge
Attribute	Field in a Node or Edge

Table 4.4: Mapping from ER Diagram to Neo4j

Consider again the example with the two entities, 'Room' and 'Building', and a 1:N relationship between them:

ER Diagram	Neo4j
Room	Room Node
Building	Building Node
has	has Edge (connecting Building and Room)

Table 4.5: Example Mapping from ER Diagram to Neo4j

This example again in the .erd syntax:

```
1 weak entity Room {
2     room_nr: INT partial-key
3 }
4 entity Building {
```

```
5     building_id: CHAR(8) key
6     address: VARCHAR(255)
7 }
8 weak relationship has {
9     Room[N] -> Building[1]
10 }
```

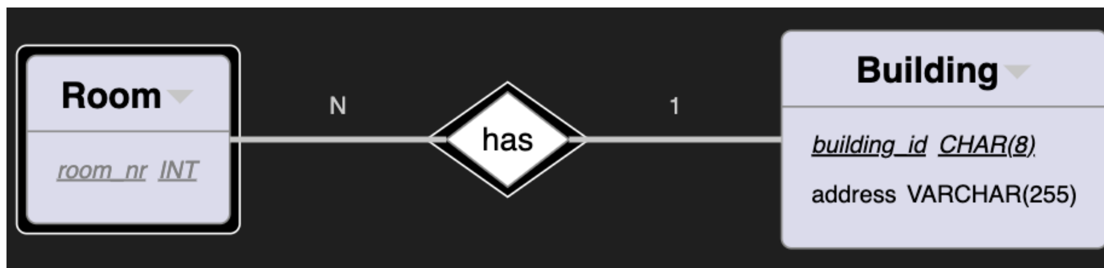


Figure 4.6: The simple ER-diagram example for Transformation.

This should map to something like this:

```
1 CREATE (Room:Room {name: "Room", Room_room_nr: "INT"})
2 CREATE (Building:Building {name: "Building", Building_address: "
    ↪ VARCHAR(255)", Building_building_id: "CHAR(8)"})
3 CREATE (Building)-[has:has{name: "has" }]->(Room)
```

In practice, the schema with populated data could look like this:

```
1 CREATE (b:Building { name: 'Building A', address: '123 Main St',
    ↪ building_id: 'B1234567' })
2 CREATE (r1:Room { name: 'Room 101', room_nr: 101 })
3 CREATE (r2:Room { name: 'Room 102', room_nr: 102 })
4 CREATE (b)-[:HAS]->(r1)
5 CREATE (b)-[:HAS]->(r2)
```

This results in the following graph:

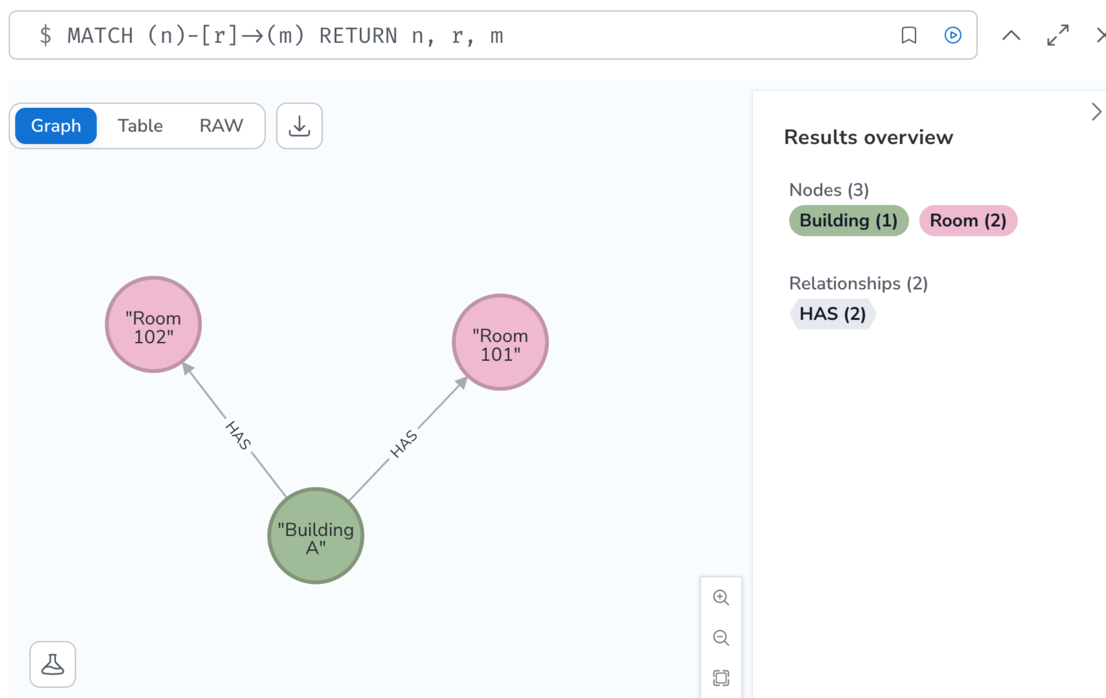


Figure 4.7: The resulting graph from the building-has-room example.

4.3.3 Implementing the Neo4j Export

We start with the generator by implementing the interface **IErGenerator** and integrate it into BIGER like we did in the previous MongoDB generator:

```
1
2  class Neo4jGenerator implements IErGenerator {
3      override void generate(Resource resource, IFileSystemAccess2 fsa,
   ↪   IGeneratorContext context) {
4          val model = resource.contents.get(0) as Model
5          validate(resource, model)
6          val fileName = (model.name ?: 'output') + ".cypher"
7          fsa.generateFile(fileName, generate(model))
8      }
9
10     def String generate(Model model) {
11     return '''
12       // entities
13       «FOR entity : model.entities»
14         «entity.toTable»
15       «ENDFOR»
16       // relationships
17       «FOR relationship : model.relationships»
18         «relationship.toTable»
```

```
19        «ENDFOR»
20        // extends relationships (IS_A)
21        «FOR entity : model.entities»
22          «entity.toTableExtends»
23        «ENDFOR»
24        // constraints
25        «FOR entity : model.entities»
26          «entity.uniqueKeyConstraint»
27        «ENDFOR»
28      '''
29    }
30      ...
31  }
```

The output file will have the extension **.cypher**. The model gets passed to the generate method, where the structure of the output file gets apparent. Firstly, entities will be iterated over and transformed with the **toTable()** method.

```
1 private def toTable(Entity entity) {
2     return '''
3             CREATE («entity.name»:«entity.name» {name: "«entity.name»
    ↪ "«FOR attribute : entity.getAllAttrWithExtendsWithNamePrefix »,
    ↪  «attribute.name»: "«attribute.datatype.transformDataType»"«
    ↪ ENDFOR»})«'\n'»
4         '''
5 }
```

We are creating a node with the label of the entities name. One special characteristic of our creation statement is that we add a fixed first attribute "name" to every node created, which holds the entities name. This is so that the graph visualizer of Neo4j can display the name of the node in the graph, instead of just the value of the first arbitrary attribute.

Then we again list the properties with the same recursive method combination of **getAllAttrWithExtends** and **getAllAttrWithExtendsWithNamePrefix**. This will prefix the attributes names with the names of the current entity, even if they extend other entities:

```
1 private def Iterable<Attribute> getAllAttrWithExtendsWithNamePrefix(
    ↪ Entity entity) {
2     val attributes = newHashSet
3     for (attr : entity.attributes) {
4         if (!attr.name.startsWith(entity.name)) {
5             attr.name = entity.name + '_' + attr.name
6         }
7         attributes += attr
8     }
9     if (entity.extends !== null) {
10        attributes.addAll(getAllAttrWithExtendsWithNamePrefix(entity.
    ↪ extends))
```

```
11     }
12     return attributes
13 }
14
15 private def Iterable<Attribute> getAllAttrWithExtends(Entity entity)
   ↪ {
16     val attributes = newHashSet
17     attributes += entity.attributes
18     if (entity.extends !== null) {
19         attributes.addAll(getAllAttrWithExtendsWithNamePrefix(entity.
   ↪ extends))
20     }
21     return attributes
22 }
```

After that, the relationships have their own **toTable()** method which get

```
1 private def toTable(Relationship relationship) {
2     var first = relationship.first;
3     var second = relationship.second;
4     var third = relationship.third;
5
6     if (first.cardinality == CardinalityType.MANY) {
7         first = second
8         second = relationship.first
9     }
10
11     return '''
12         «IF third?.target === null»CREATE («first.target.name»)-[«
   ↪ relationship.name»:«relationship.name»{name: "«relationship.
   ↪ name»"«FOR attribute : relationship.attributes», «relationship.
   ↪ name»_«attribute.name»: "«attribute.datatype.transformDataType»
   ↪ "«ENDFOR» }]->(«second.target.name»)«'\n'»«ENDIF»
13         «IF third?.target !== null»CREATE («relationship.name»:«
   ↪ relationship.name»{name: "«relationship.name»"«FOR attribute :
   ↪ relationship.attributes», «relationship.name»_«attribute.name»:
   ↪  "«attribute.datatype.transformDataType»"«ENDFOR»})«'\n'»«ENDIF
   ↪ »
14         «IF third?.target !== null»CREATE («relationship.name»)-[«
   ↪ relationship.name»_«first.target.name»:«relationship.name»_«
   ↪ first.target.name» {«FOR attribute : relationship.attributes
   ↪ SEPARATOR ','»«attribute.name»: "«attribute.datatype.
   ↪ transformDataType»"«ENDFOR» }]->(«first.target.name»)«'\n'»«
   ↪ ENDIF»
15         «IF third?.target !== null»CREATE («relationship.name»)-[«
   ↪ relationship.name»_«second.target.name»:«relationship.name»_«
   ↪ second.target.name» {«FOR attribute : relationship.attributes
   ↪ SEPARATOR ','»«attribute.name»: "«attribute.datatype.
   ↪ transformDataType»"«ENDFOR» }]->(«second.target.name»)«'\n'»«
   ↪ ENDIF»
```

```
16          «IF third?.target !== null»CREATE («relationship.name»)-[«
    ↪ relationship.name»_«third.target.name»:«relationship.name»_«
    ↪ third.target.name» {«FOR attribute : relationship.attributes
    ↪ SEPARATOR ','»«attribute.name»: "«attribute.datatype.
    ↪ transformDataType»"«ENDFOR» }]->(«third.target.name»)«'\n'»«
    ↪ ENDIF»
17      '''
18  }
```

The first part deals with the direction of the relationship we want to have. Our general idea is to point towards the **many**-cardinality like one home has many rooms (home ⟶ room). After that, the real creation statement begins, but we have two cases:

1. if there is no third entity connected to the relationship, then we create a normal relationship and the first and the second entity get connected. The ordering depends on the before mentioned cardinality.
2. if there happens to be a third entity connected to the relationship, forming a ternary relationship, we will create a new node with the name of the relationship. We then create three relationships, connecting the newly created node to the other three entities.

Additionally, we need a way to generate relationships of those extended nodes, which should be displayed as a **IS_A** relationship. As an example, if a tutor is also a student and the entity tutor extends the entity student, the result should be `CREATE (Tutor)-[Tutor_IS_A:Tutor_IS_A]->(Student)`. The following method does exactly that, by taking an entity and if it extends an entity, then this relationship gets made:

```
1  private def toTableExtends(Entity entity){
2      return '''
3          «IF entity?.extends !== null»CREATE («entity?.name»)-[«entity
    ↪ ?.name»_IS_A:«entity?.name»_IS_A]->(«entity.extends?.name»)«'\n
    ↪ '»«ENDIF»
4      '''
5  }
```

The order is important, because we are connecting nodes to relationships, by referencing different nodes, these node need to exist before relationship created that reference them.

And finally, we can put constraints for entities and their fields. We will only constrain entities for their primary keys as required fields. For this we use the **create constraint** command like this:

```
1  private def uniqueKeyConstraint(Entity entity) {
2      val keys = entity.primaryKeys
3      if (keys.length > 1)
```

```
4         return '''
5             CREATE CONSTRAINT IF NOT EXISTS FOR (x:«entity.name»)
    ↪ REQUIRE (x.«keys.map[key | key.name].join(', ')») IS UNIQUE;«'\
    ↪ n'»
6         '''
7     else
8         return '''
9             CREATE CONSTRAINT IF NOT EXISTS FOR (x:«entity.name»)
    ↪ REQUIRE x.«entity.primaryKey.name» IS UNIQUE;«'\n'»
10        '''
11 }
```

For our Neo4j export, the results are shown in the Showcase 5 Chapter below.

4.3.4 Implementing the Neo4j Import

In contrast to the previous code import in MongoDB 4.2.4, we don't have the need to add some comment do distinguish a relationship from an entity. However, we indeed have a special cases for importing Neo4j code to handle.

Interpreting **ternary relationships** is not obvious, as it is impossible to know if the entity connecting the other three entities was indeed intended, or was the result of the code export transforming a ternary relationship into that connecting entity. In our code import, we will assume exactly that and always interpret such an arrangement as such, a ternary relationship. This occurs when we can count exactly three relationships ending in one node.

We proceed with the necessary regular expressions:

```
1 private static final Pattern ENTITY_PATTERN = Pattern.compile("CREATE
    ↪  \\((\\w+):(\\w+)\\s*\\{(.*?)\\}");
2 private static final Pattern FIELD_PATTERN = Pattern.compile("(?:\\s
    ↪ *(\\w+):\\s*\"([^\"]*)\")");
3 private static final Pattern RELATIONSHIP_PATTERN = Pattern.compile("
    ↪ CREATE \\((\\w+)\\)-\\[(\\w+):(\\w+)\\s*\\{([^}]*)
    ↪ \\}\\]->\\((\\w+)\\)");
```

We once more search for the node creation statements with the **ENTITY_PATTERN**, the fields of the entity or relationship with **FIELD_PATTERN**, and finally we search for our relationships with **RELATIONSHIP_PATTERN** to search out relationships.

The following implementation expresses the transformation of a Neo4j .cypher file into the familiar .erd file structure:

```
1 public static void main(String[] args) {
2         if (args.length != 1) {
3             System.out.println("Usage: Java Neo4jInterpreter <file_path>");
4             return;
5         }
6
7         StringBuilder output = new StringBuilder();
8         String filePath = args[0];
```

```
9          Map<String, Map<String, String>> entities = new LinkedHashMap<>();
10         Map<String, String[]> relationships = new LinkedHashMap<>();
11 
12         // Counter for Entities in relationships (for ternary relationships)
13         Map<String, Integer> entityRelationshipCounter = new HashMap<>();
14 
15         try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
16             String line;
17             while ((line = reader.readLine()) != null) {
18                 Matcher entityMatcher = ENTITY_PATTERN.matcher(line);
19                 if (entityMatcher.find()) {
20                     String entityName = entityMatcher.group(1);
21                     String entityType = entityMatcher.group(2);
22 
23                     Map<String, String> fields = new LinkedHashMap<>();
24                     Matcher fieldMatcher = FIELD_PATTERN.matcher(entityMatcher.group(3)
   ↪ );
25                     while (fieldMatcher.find()) {
26                         fields.put(fieldMatcher.group(1), fieldMatcher.group(2));
27                     }
28 
29                     entities.put(entityName, fields);
30                 }
31 
32                 Matcher relationshipMatcher = RELATIONSHIP_PATTERN.matcher(line);
33                 if (relationshipMatcher.find()) {
34                     String relationshipName = relationshipMatcher.group(2);
35                     String[] entitiesInRelationship = {relationshipMatcher.group(1),
   ↪ relationshipMatcher.group(5)};
36                     // counting entity accurances in releationships
37                     entityRelationshipCounter.put(entitiesInRelationship[0],
   ↪ entityRelationshipCounter.getOrDefault(entitiesInRelationship[0], 0) + 1);
38 
39                     String relationshipProperties = relationshipMatcher.group(4);
40                     Matcher fieldMatcher = FIELD_PATTERN.matcher(relationshipProperties
   ↪ );
41                     Map<String, String> fields = new LinkedHashMap<>();
42                     while (fieldMatcher.find()) {
43                         fields.put(fieldMatcher.group(1), fieldMatcher.group(2));
44                     }
45                     relationships.put(relationshipName, entitiesInRelationship);
46                 }
47             }
48         } catch (IOException e) {
49             e.printStackTrace();
50         }
51 
52         // Print entities and their fields
53         for (Map.Entry<String, Map<String, String>> entry : entities.entrySet()) {
54           if (entityRelationshipCounter.getOrDefault(entry.getKey(), 0) != 3) {
55             output.append("entity " + entry.getKey() + " {\n");
56             for (Map.Entry<String, String> field : entry.getValue().entrySet()) {
57                 // if fields datatype is the same as entity name, skip as
58                 // this is most likely just the Vertex label for neo4j graph
59                 if (!field.getValue().equals(entry.getKey())) {
60                     output.append("\t" + field.getKey() + ": " + field.getValue() + "\n
   ↪ ");
61                 }
62             }
63             output.append("}\n");
64           }
65         }
66 
67         // Print relationships
68         for (Map.Entry<String, String[]> entry : relationships.entrySet()) {
69             if (entityRelationshipCounter.getOrDefault(entry.getValue()[0], 0) == 3) {
70                 output.append("relationship " + entry.getValue()[0] + " {");
71                 ArrayList<String> acc = new ArrayList<>();
72                 relationships.entrySet().forEach(x -> {
73                     if (x.getValue()[0].equals(entry.getValue()[0])) {
74                         acc.add(x.getValue()[1]);
75                     }
76                 });
```

```
77                  var triad = acc.toArray();
78                  output.append("\t" +  triad[0] + " -> " + triad[1] + " -> " + triad[2]
    ↪ + "\n");
79                  entities.entrySet().stream()
80                      .filter(x -> x.getKey().equals(entry.getValue()[0]))
81                      .findFirst()
82                      .ifPresent( e ->
83                          e.getValue().entrySet().forEach(field -> {
84                              output.append("\t" + field.getKey() + ": " + field.getValue
    ↪ () + "\n");
85                          })
86                      );
87              } else if (entityRelationshipCounter.getOrDefault(entry.getValue()[0], 0)
    ↪ != 0) {
88                  output.append("relationship " + entry.getKey() + " {\n");
89                  output.append("\t" + entry.getValue()[0] + " -> " + entry.getValue()[1]
    ↪  + "\n");
90              }
91              Map<String, String> relationshipFields = entities.get(entry.getKey());
92              if (relationshipFields != null) {
93                  for (Map.Entry<String, String> field : relationshipFields.entrySet()) {
94                      // if fields datatype is the same as entity name, skip as
95                      // this is most likely just the Vertex label for neo4j graph
96                      if (!field.getValue().equals(entry.getKey())) {
97                          output.append("\t" + field.getKey() + ": " + field.getValue() +
    ↪  "\n");
98                      }
99                  }
100             }
101             if (entityRelationshipCounter.getOrDefault(entry.getValue()[0], 0) == 3) {
102                 entityRelationshipCounter.put(entry.getValue()[0], 0);
103                 output.append("}\n");
104             } else if (entityRelationshipCounter.getOrDefault(entry.getValue()[0], 0)
    ↪ != 0){
105                 output.append("}\n");
106             }
107         }
108
109         System.out.println(output);
110     }
```

Just like before, here we also employ the **StringBuilder()** function to concatenate all the parts together and forming the new .erd file.

1. It opens the .cypher file and reads it line by line. It uses regular expressions to match the Cypher statements that create entities and relationships.

2. For each line matching the `ENTITY_PATTERN`, it extracts the **entity** name, type, and properties (fields). This information is stored in a `Map`, where the entity name is the key and another `Map` of field names and values is the value.

3. For every matching the `RELATIONSHIP_PATTERN`, it extracts the **relationship** name and the entities involved in the relationship. It also increments a counter for each entity involved in a relationship, used later for handling a special case when an entity appears in a ternary relationships. This information is stored in a `Map` as well, with the relationship name as the key and an array of the two entity names as the value.

4. After it is finished parsing the file, it prints out all entities and their fields, **unless** the entity is involved in **exactly three relationships**(a special case, indicating a ternary relationship).

5. Then it prints out all relationships. If an entity is involved in exactly three relationships, it prints out a **ternary relationship** in the format `entity1 -> entity2 -> entity3`. Otherwise, it prints out a **binary relationship** in the format `relationshipName: entity1 -> entity2`.

6. It also prints out the fields of the entities involved in the relationships, unless the field value is the same as the entity name, which is considered to be most likely the Vertex label for the Neo4j graph.

4.4 CassandraDB

Let's finally look at our column-oriented database CassandraDB. According to the documentation [1], Cassandra does not support joins, but rather encourages the design of the column families or tables, such that it reflects queries made to it. If one needs something like the join operation, one could do the join operation on the client side. But the Cassandra's recommended way is to add another table that would be the result of that join operation. In contrast to a relational database system, where we normalize data and avoid redundancy between tables, Cassandra performs best when its data model is denormalized. The main benefit of this performance, as join operations especially on large data sets are slow.

4.4.1 Transformation Goals

For our Cassandra code, we can use a very similar method as the usual SQL export, as the only restriction is that of the foreign keys. For those are still written in the tables, but cannot be marked as such. The extends relationships are implemented for those entities that are extended as to have all the attributes of the entity that extends it. Lastly, as we will need to export the relationships as tables as well, we need some way of differentiate them from entity tables, so our import will be able to discrimenate between the two. This will be achieved by using the command ***WITH comment='relationship'***.

4.4.2 Mapping Tables

Mapping the entities, relationships, and attributes in the ER diagram into Tables and Fields in Cassandra:

ER Diagram	Cassandra
Entity	Table
Relationship	Table
Attribute	Field in a Table

Table 4.6: Mapping from ER Diagram to Cassandra

We will illustrate one last time with the same example as before, 'Room' and 'Building', and a 1:N relationship between them:

ER Diagram	Cassandra
Room	Room Table
Building	Building Table
has	has Table (connecting Building and Room)

Table 4.7: Example Mapping from ER Diagram to Cassandra

The .erd syntax:

```
weak entity Room {
    room_nr: INT partial-key
}
entity Building {
    building_id: CHAR(8) key
    address: VARCHAR(255)
}
weak relationship has {
    Room[N] -> Building[1]
}
```

This should map to something like this:

```
CREATE TABLE Room(
  room_nr INT,
  PRIMARY KEY (room_nr)
);
CREATE TABLE Building(
  building_id TEXT,
  address TEXT,
  PRIMARY KEY (building_id)
);
CREATE TABLE has(
  room_nr INT,
  building_id TEXT,
  PRIMARY KEY (room_nr, building_id)
) WITH comment='relationship';
```

In practice, the schema with populated data could look like this:

```
INSERT INTO Room (room_nr) VALUES (101);
INSERT INTO Room (room_nr) VALUES (102);
INSERT INTO Room (room_nr) VALUES (103);

INSERT INTO Building (building_id, address) VALUES ('B123', '123 Main
    ↪ St');
INSERT INTO Building (building_id, address) VALUES ('B456', '456 Elm
    ↪ St');

INSERT INTO has (room_nr, building_id) VALUES (101, 'B123');
```

```
9 INSERT INTO has (room_nr, building_id) VALUES (102, 'B123');
10 INSERT INTO has (room_nr, building_id) VALUES (103, 'B123');
11 INSERT INTO has (room_nr, building_id) VALUES (101, 'B456');
12 INSERT INTO has (room_nr, building_id) VALUES (102, 'B456');
```

In the above example, we have two buildings with IDs **'B123'** and **'B456'**. Building **'B123'** has three rooms **'101', '102' and '103'** and building **'B456'** has two rooms **'101' and '102'**. The **'has'** table represents the relationship between buildings and rooms.

4.4.3 Implementing the Cassandra Export

We start as before by adding a new generator for our Cassandra transformation called **CassandraDbGenerator** and implement the interface:

```
1 class CassandraDbGenerator implements IErGenerator {
2
3     override void generate(Resource resource, IFileSystemAccess2 fsa,
   ↪  IGeneratorContext context) {
4         val model = resource.contents.get(0) as Model
5         validate(resource, model)
6         val fileName = (model.name ?: 'output') + ".cql"
7         fsa.generateFile(fileName, generate(model))
8     }
9     ...
10 }
```

As in previous generators, we define the name and extension of the output file, as well as call our code generation method **generate(model)**:

```
1
2 def String generate(Model model) {
3     '''
4         CREATE KEYSPACE «model.name» WITH REPLICATION = {'class': '
   ↪ SimpleStrategy', 'replication_factor': 2};
5         USE «model.name»;
6         «FOR entity : model.entities.reject[it.extends !== null]»
7             «entity.toTable»
8         «ENDFOR»
9         «FOR entity : model.entities.reject[it.extends === null]»
10            «entity.toTable»
11        «ENDFOR»
12        «FOR relationship : model.relationships.reject[!it.isWeak]»
13            «relationship.weakToTable»
14        «ENDFOR»
15        «FOR relationship : model.relationships.reject[it.isWeak]»
16            «relationship.toTable»
17        «ENDFOR»
18    '''
19 }
```

The first command **CREATE KEYSPACE** is the creation of Cassandra's equivalent to a database. The **name** of the keyspace is passed, which in our case is just the name of the model.

The **replication factor** is a mandatory property and must contain two parameters:

1. The **'class'** sub option defines the replication strategy [23] that should be used. We are choosing the **SimpleStrategy** for our purposes, which defines a replication factor for how data should be spread across the cluster. However, this strategy is generally NOT recommended by Cassandra for use in production. For a production environment the strategy **NetworkTopologyStrategy** is recommended.

2. Depending on the replication strategy previously selected. In our case, SimpleStrategy requires the single required argument, the $'replication_factor'$, which is specified by an integer number and defines the number of replicas to store per range.

The **USE** statement selects our newly created keyspace and allows the rest of the code to be applied to it.

The following FOR loops iterate over entities and relationships, whereby the entities which do not extend any other entities are firstly processed, due to some the folding of the attributes which could lead to redundant naming. Also non weak relationships get prioritized in the ordering for the relationships. We again have separate **toTable()** methods:

```
1  private def toTable(Entity entity) {
2      return '''
3              CREATE TABLE «entity.name»(
4              «FOR attribute : entity.getAllAttrWithExtends.reject[it.
   ↪ type == AttributeType.DERIVED]»
5                  «'\t'»«attribute.name» «attribute.datatype.
   ↪ transformDataType»,
6              «ENDFOR»
7              «'\t'»PRIMARY KEY («entity.getPrimaryKeysName»)
8              );«'\n'»«'\n'»
9          '''
10 }
```

The entity tables are created in a similar fashion as in SQL, the attributes get defined with their data types, as well as the primary key. We can again observer the **getAllAttrWithExtends()** method, which recursively collects all the attributes from the entity and the entities it extends (extends-chain). These are identical to the ones from the MongoDB generator.

Let us next examine the translation of the relations to Cassandra tables:

```
1 private def toTable(Relationship relationship) {
2     val keySource = relationship.first.target?.primaryKey
```

```
3       val keyTarget = relationship.second.target?.primaryKey
4       return '''
5               CREATE TABLE «relationship.name»(
6               «relationship.first.target.foreignKeyRef»,
7               «relationship.second.target.foreignKeyRef»,
8               «IF relationship.third?.target !== null»«relationship.
    ↪ third.target.foreignKeyRef»,«ENDIF»
9               «FOR attribute : relationship.attributes»
10                  «'\t'»«attribute.name» «attribute.datatype.
    ↪ transformDataType»,
11              «ENDFOR»
12              «'\t'»PRIMARY KEY («keySource», «keyTarget»«IF
    ↪ relationship.third?.target !== null», «relationship.third.
    ↪ target.getPrimaryKeysName»«ENDIF»)
13              ) WITH comment='relationship';«'\n'»«'\n'»
14          '''
15  }
16
17  private def weakToTable(Relationship relationship) {
18      val strong = getStrongEntity(relationship)
19      val weak = getWeakEntity(relationship)
20      return '''
21              CREATE TABLE «relationship.name»(
22              «FOR attribute : weak.allAttributes.reject[it.type ===
    ↪ AttributeType.DERIVED]»
23                  «'\t'»«attribute.name» «attribute.datatype.
    ↪ transformDataType»,
24              «ENDFOR»
25              «FOR attribute : relationship.attributes»
26                  «'\t'»«attribute.name» «attribute.datatype.
    ↪ transformDataType»,
27              «ENDFOR»
28              «FOR key : strong.primaryKeys»
29                  «'\t'»«key.name» «key.datatype.transformDataType»,
30              «ENDFOR
31              «'\t'»PRIMARY KEY («weak.getPartialKeysName», «strong.
    ↪ getPrimaryKeysName»)
32              ) WITH comment='relationship';«'\n'»«'\n'»
33          '''
34  }
```

The variables **keySource** and **keyTarget** are derived from the primary keys of the first and second entity of the relationship, respectively. When creating the table, the first fields we define are the foreign key references within the first and second entity of the relationship. If there is a third entity (ternary relationship), we include also it's foreign keys. We then proceed to loop through the relationships attributes and print it side by side with its data types. Similarly to SQL, the primary keys can be set and we will include the primary keys of the relationships entities.

```
private def getStrongEntity(Relationship r) {
    if (r.first.target.isWeak) {
        return r.second.target
    } else {
        return r.first.target
    }
}

private def getWeakEntity(Relationship r) {
    if (r.first.target.isWeak) {
        return r.first.target
    } else {
        return r.second.target
    }
}
```

The weak relationships have their own method, which differs in the ordering of the primary keys, in that the key of the strong entity is the only one that gets written as a field, but both keys get marked in the primary key statement. And lastly, we use Cassandra's WITH statement to add a comment to this table, where we write for every relationship table **"relationship"**, so as to not loose information. This will be relevant for the import of our transformed output later on.

In the official documentation we have an extensive list of the legal data types [11] of Cassandra. Our method of transforming the data types of our logical model to the physical model looks like this:

```
// CQL datatypes https://Cassandra.apache.org/doc/latest/Cassandra/cql/types.html
private def transformDataType(DataType dataType) {
    // default
    if(dataType === null) {
        return 'TEXT'
    }
    val type = dataType.type.toUpperCase()
    if(type == 'VARCHAR' || type == 'CHAR' || type.startsWith('STRING')) {
        return 'TEXT';
    }
    if(type == 'SMALLINT' || type == 'BIGINT') {
        return 'VARINT';
    }
    return type
}
```

As we can see, we focus on the main data types like string and integers, which are converted to the type **'TEXT'** and **'VARINT'** respectively. In future work, this can be extended to support more types such as date. Similarly to the other generators, we do

not have any need to validate the .erd syntax, as we do support extends relationships already and therefore our **validate()** method remains empty.

4.4.4 Implementing the Cassandra Import

To reverse our export, we use the following regular expression patterns:

```
1 private static final Pattern ENTITY_PATTERN = Pattern.compile("CREATE
    ↪ TABLE (\\w+)\\(");
2 private static final Pattern FIELD_PATTERN = Pattern.compile("(\\w+)
    ↪ \\s+(\\w+)(?:,|\\)|\\s)");
3 private static final Pattern PRIMARYKEYS_PATTERN = Pattern.compile("
    ↪ PRIMARY KEY \\(([^)]+)\\)");
4 private static final Pattern WITHCOMMENTRELATIONSHIP_PATTERN =
    ↪ Pattern.compile("WITH\\s*comment\\s*=\\s*'relationship'");
```

As before, we use the regular expressions to identify all the entities, relationships and their fields. We also can detect primary keys because of Cassandras `PRIMARY KEY` syntax. Also, it distinguishes between entities and relationships based on the comment 'relationship' in the schema.

```
1  public static void main(String[] args) {
2      if (args.length != 1) {
3          System.out.println("Usage: Java CassandraInterpreter <
    ↪ file_path>");
4          return;
5      }
6
7      StringBuilder output = new StringBuilder();
8      String filePath = args[0];
9      Map<String, Map<String, Tuple<String, Boolean>>> entities = new
    ↪ LinkedHashMap<>();
10     Map<String, Map<String, Tuple<String, Boolean>>> relationships =
    ↪ new LinkedHashMap<>();
11     var primary_keys = new ArrayList<String>();
12     Map<String, String> keys_from_entities = new HashMap<>();
13
14     try (BufferedReader reader = new BufferedReader(new FileReader(
    ↪ filePath))) {
15         StringBuilder sb = new StringBuilder();
16         String line;
17         while ((line = reader.readLine()) != null) {
18             sb.append(line);
19             sb.append(" ");
20         }
21
22         String content = sb.toString();
23         String[] parts = content.split("CREATE TABLE");
24
```

```
25          for (String part : parts) {
26              if (part.isEmpty()) {
27                  continue;
28              }
29
30              Matcher entityMatcher = ENTITY_PATTERN.matcher("CREATE
    ↪ TABLE" + part);
31              if (entityMatcher.find()) {
32                  String entityName = entityMatcher.group(1);
33                  Matcher fieldMatcher = FIELD_PATTERN.matcher(part);
34
35                  // Regular expression to detect primary key
    ↪ definition
36                  Matcher primkMatcher = PRIMARYKEYS_PATTERN.matcher(
    ↪ part);
37                  while (primkMatcher.find()) {
38                      var f = primkMatcher.group(1).split(",");
39                      for(int i = 0; i < f.length; i++) {
40                          primary_keys.add(f[i]);
41                      }
42                  }
43
44                  Map<String, Tuple<String, Boolean>> fields = new
    ↪ HashMap<>();
45                  while (fieldMatcher.find()) {
46                      if (!(fieldMatcher.group(1).toUpperCase().equals(
    ↪ "PRIMARY")
47                      && fieldMatcher.group(2).toUpperCase().equals("
    ↪ KEY"))){
48                          if (primary_keys.contains(fieldMatcher.group
    ↪ (1))) {
49                              fields.put(fieldMatcher.group(1), new
    ↪ Tuple<>(fieldMatcher.group(2), true));
50                          } else {
51                              fields.put(fieldMatcher.group(1), new
    ↪ Tuple<>(fieldMatcher.group(2), false));
52                          }
53                      }
54                  }
55
56                  Matcher commentsFieldsMatcher =
    ↪ WITHCOMMENTRELATIONSHIP_PATTERN.matcher(part);
57                  if (commentsFieldsMatcher.find()) {
58                      relationships.put(entityName, fields);
59                  }else {
60                      entities.put(entityName, fields);
61                      for (var f : fields.entrySet()) {
62                          if (f.getValue().getSecond()) {
63                              keys_from_entities.put(f.getKey(),
```

```
↪ entityName);
64                     }
65                 }
66             }
67         }
68     }
69 } catch (IOException e) {
70     e.printStackTrace();
71 }
72 
73 // Print tables and their fields
74 for (Map.Entry<String, Map<String, Tuple<String, Boolean>>> entry
↪  : entities.entrySet()) {
75     output.append("entity " + entry.getKey() + " {\n");
76     for (Map.Entry<String, Tuple<String, Boolean>> field : entry.
↪ getValue().entrySet()) {
77         if (primary_keys.contains((field.getKey()))) {
78             output.append("\t" + field.getKey() + ": " +
↪ translateDataType(field.getValue().getFirst()) + " key" + "\n")
↪ ;
79         } else {
80             output.append("\t" + field.getKey() + ": " +
↪ translateDataType(field.getValue().getFirst()) + "\n");
81         }
82     }
83     output.append("}\n");
84 }
85 // Print tables and their fields
86 for (Map.Entry<String, Map<String, Tuple<String, Boolean>>> rel :
↪  relationships.entrySet()) {
87     output.append("relationship " + rel.getKey() + " {\n");
88     // keyed fields
89     int field_counter = 0;
90     for (Map.Entry<String, Tuple<String, Boolean>> field : rel.
↪ getValue().entrySet()) {
91         if (field.getValue().getSecond()) {
92             output.append((field_counter++ > 0 ? " -> " : "\t")+
↪ keys_from_entities.getOrDefault(field.getKey(), field.getKey())
↪ );
93         }
94     }
95     output.append("\n");
96     for (Map.Entry<String, Tuple<String, Boolean>> field : rel.
↪ getValue().entrySet()) {
97         if (!field.getValue().getSecond()) {
98             output.append("\t" + field.getKey() + ": " +
↪ translateDataType(field.getValue().getFirst().toLowerCase()) +
↪ "\n");
99         }
```

```
100             }
101             output.append("}\n");
102         }
103 
104         System.out.println(output);
105     }
```

Finally, we use the `translateDataType` method in order to transform the data types to SQL like syntax:

```
1 private static String translateDataType(String Datatype) {
2     switch(Datatype.toUpperCase()) {
3         case "TEXT":
4             return "VARCHAR(255)";
5         case "VARINT":
6             return "INT";
7     }
8     return Datatype;
9 }
```

To summarize, the earlier defined regular expression patterns enable us to detect the entities, fields, and relationships using the `Pattern` class in Java. The pattern for entities is set as "`CREATE TABLE (\w+)\(`", which looks for the "`CREATE TABLE`" keyword followed by the entity name. The pattern for fields is set as "`(\w+)\s+(\w+)(?:,\)\s)`", which looks for a word "field name", followed by a space, another word "data type", and then either a comma, a closing parenthesis, or a space. The pattern for primary keys is set as "`PRIMARY KEY \(([^)]+)\)`", which looks for the "`PRIMARY KEY`" keyword followed by the names of the fields making up the primary key enclosed in parentheses. We read the file input line by line using a `BufferedReader`, then the lines split using the "`CREATE TABLE`" pattern. Then we apply the pattern on all of those parts. If the string "`WITH comment = 'relationship'`" occurs, it is considered as a relationship. Otherwise, it is considered an entity.

4.5 Summary

We saw how different methods of each of the databases can be used to restrict and therefor create a schema. For MongoDB we used it's validation function in the creation statements to restrict the types and fields of each collection. For Neo4j we used nodes for entities and edged for the relationships. We added all the fields as attributes and focused on the the outlook of the resulting graph by adding name attributes so the name of the entity can be displayed in the nodes. Lastly, Cassandra's CQL syntax allowed us to more plainly generate the schema. We mapped entities, relationships and their fields to Tables and used the *WITH* notation to differentiate between entity and relationship. Using these methods allowed us not only to generate code, but also to import it back into bigER. We accomplished this by defining regular expressions to pattern match the

syntax of each of the languages of our databases and reconstructing the information into the .erd syntax of bigER.

CHAPTER 5

Showcase

Let's see what the result of this transformation looks like. If we take a look at the **university.erd** example, with the inclusion of the **Tutor** entity which extends **Student**, the input .erd file before the transformation:

```
1  // ER Model
2  erdiagram University
3
4  // Options
5  notation=default
6
7  // Entities
8  entity Tutor extends Student {
9      id: INT key
10     subject: VARCHAR(255)
11 }
12 entity Student {
13     id: INT key
14     name: VARCHAR(255)
15     birthday: DATE
16     age: SMALLINT
17 }
18 entity Course {
19     course_nr: INT key
20     course_name: VARCHAR(100)
21     credits: SMALLINT
22 }
23 weak entity Lecture {
24     title: VARCHAR(255) partial-key
25 }
26 entity Instructor {
27     instructor_id: INT key
```

```
28     name: VARCHAR(255)
29 }
30 entity Department {
31     dept_nr: INT key
32     name: VARCHAR(100)
33     abbreviation: CHAR(5)
34 }
35 weak entity Room {
36     room_nr: INT partial-key
37 }
38 entity Building {
39     building_id: CHAR(8) key
40     address: VARCHAR(255)
41 }
42
43 // Relationships
44 relationship Exam {
45     Student[1] -> Course[N] -> Instructor[N]
46     points: DOUBLE
47 }
48 weak relationship has {
49     Room[N] -> Building[1]
50 }
51 relationship Office {
52     Room[1] -> Instructor[1]
53 }
54 relationship Work {
55     Instructor[N] -> Department[1]
56 }
57 weak relationship include {
58     Course[1] -> Lecture[N]
59 }
60 relationship Location {
61     Building[N] -> Department[1]
62 }
```

Here is this graph visually represented in bigER:

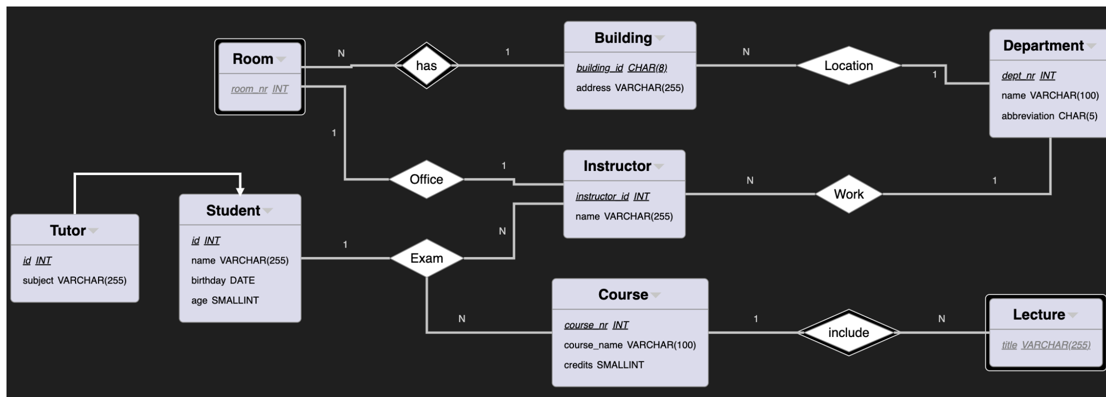


Figure 5.1: University example diagram.

We will use this University example in all of the transformation, so to better compare the resulting output code and observe possible loss of information between the physical models.

In BIGER we can generate all scripts via the context menu:

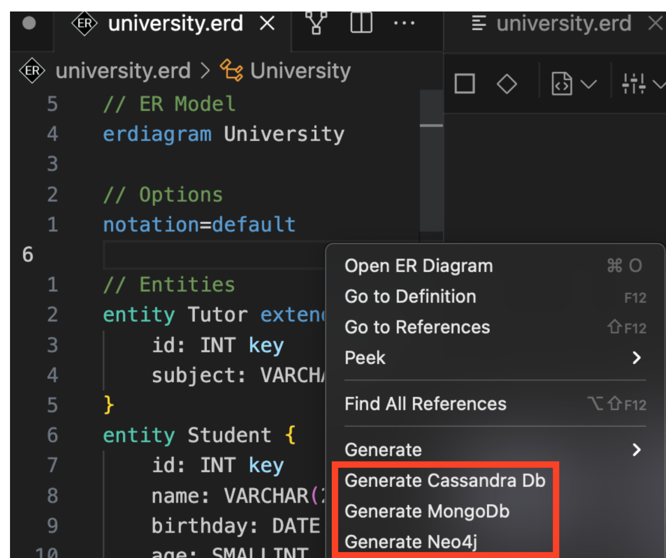


Figure 5.2: University example diagram.

5.1 MongoDB Transformation

5.1.1 MongoDB Setup

In order to test the correctness of our transformation, we will use the containerization tool Docker [7]. To use a MongoDB instance, we can pull from Docker hub [14] with

`Docker pull mongo` to pull the image to our local machine. After that, using the Docker dashboard, we can activate the image:

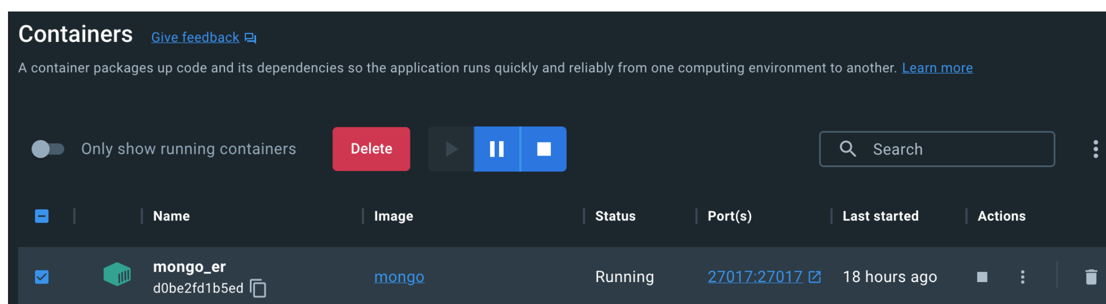


Figure 5.3: Docker MongoDb image in dashboard.

We can use the Visual Studio Code extension [2] to access and interact with our MongoDb instance.

5.1.2 MongoDB Transformation

After exporting this diagram through our MongoDB generator, the resulting .js file output looks like this:

```
use(University);
db.createCollection("Tutor", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      title: "Tutor Object Validation",
      required: ["id"],
      properties: {
        id: {
          bsonType: "int"
        },
        Student_birthday: {
          bsonType: "date"
        },
        subject: {
          bsonType: "string"
        },
        Student_name: {
          bsonType: "string"
        },
        Student_age: {
          bsonType: "smallint"
        },
        Student_id: {
          bsonType: "int"
        }
```

```
27        }
28      }
29    }
30 });
31 db.createCollection("Student", {
32   validator: {
33     $jsonSchema: {
34       bsonType: "object",
35       title: "Student Object Validation",
36       required: ["Student_id"],
37       properties: {
38         Student_birthday: {
39           bsonType: "date"
40         },
41         Student_name: {
42           bsonType: "string"
43         },
44         Student_age: {
45           bsonType: "smallint"
46         },
47         Student_id: {
48           bsonType: "int"
49         }
50       }
51     }
52   }
53 });
54
55
56 db.createCollection("Course", {
57   validator: {
58     $jsonSchema: {
59       bsonType: "object",
60       title: "Course Object Validation",
61       required: ["course_nr"],
62       properties: {
63         credits: {
64           bsonType: "smallint"
65         },
66         course_nr: {
67           bsonType: "int"
68         },
69         course_name: {
70           bsonType: "string"
71         }
72       }
73     }
74   }
75 });
```

```
76 db.createCollection("Lecture", {
77   validator: {
78     $jsonSchema: {
79       bsonType: "object",
80       title: "Lecture Object Validation",
81       required: ["title"],
82       properties: {
83         title: {
84           bsonType: "string"
85         }
86       }
87     }
88   }
89 });
90
91
92 db.createCollection("Instructor", {
93   validator: {
94     $jsonSchema: {
95       bsonType: "object",
96       title: "Instructor Object Validation",
97       required: ["instructor_id"],
98       properties: {
99         instructor_id: {
100          bsonType: "int"
101        },
102        name: {
103          bsonType: "string"
104        }
105      }
106    }
107  }
108 });
109 db.createCollection("Department", {
110   validator: {
111     $jsonSchema: {
112       bsonType: "object",
113       title: "Department Object Validation",
114       required: ["dept_nr"],
115       properties: {
116         abbreviation: {
117           bsonType: "string"
118         },
119         dept_nr: {
120           bsonType: "int"
121         },
122         name: {
123           bsonType: "string"
124         }
```

```
125       }
126     }
127   }
128 });
129 db.createCollection("Room", {
130   validator: {
131     $jsonSchema: {
132       bsonType: "object",
133       title: "Room Object Validation",
134       required: ["room_nr"],
135       properties: {
136         room_nr: {
137           bsonType: "int"
138         }
139       }
140     }
141   }
142 });
143 db.createCollection("Building", {
144   validator: {
145     $jsonSchema: {
146       bsonType: "object",
147       title: "Building Object Validation",
148       required: ["building_id"],
149       properties: {
150         address: {
151           bsonType: "string"
152         },
153         building_id: {
154           bsonType: "string"
155         }
156       }
157     }
158   }
159 });
160 db.createCollection("has", {
161   validator: {
162     $jsonSchema: {
163       bsonType: "object",
164       title: "has (relationship) Object Validation",
165       required: ["room_nr", "building_id"],
166       properties: {
167         room_nr: {
168           bsonType: "int",
169         },
170         building_id: {
171           bsonType: "string",
172         },
173       }
```

```
174       }
175     }
176 });
177 db.createCollection("include", {
178   validator: {
179     $jsonSchema: {
180       bsonType: "object",
181       title: "include (relationship) Object Validation",
182       required: ["course_nr", "title"],
183       properties: {
184         course_nr: {
185           bsonType: "int",
186         },
187         title: {
188           bsonType: "string",
189         },
190       }
191     }
192   }
193 });
194 db.createCollection("Exam", {
195   validator: {
196     $jsonSchema: {
197       bsonType: "object",
198       title: "Exam (relationship) Object Validation",
199       required: ["Student_id", "course_nr", "instructor_id"],
200       properties: {
201         instructor_id: {
202           bsonType: "int",
203         },
204         course_nr: {
205           bsonType: "int",
206         },
207         Student_id: {
208           bsonType: "int",
209         },
210         points: {
211           bsonType: "double",
212         },
213       }
214     }
215   }
216 });
217 db.createCollection("Office", {
218   validator: {
219     $jsonSchema: {
220       bsonType: "object",
221       title: "Office (relationship) Object Validation",
222       required: ["room_nr", "instructor_id"],
```

```
223         properties: {
224           room_nr: {
225             bsonType: "int",
226           },
227           instructor_id: {
228             bsonType: "int",
229           },
230         }
231       }
232     }
233 });
234 db.createCollection("Work", {
235     validator: {
236       $jsonSchema: {
237         bsonType: "object",
238         title: "Work (relationship) Object Validation",
239         required: ["instructor_id", "dept_nr"],
240         properties: {
241           dept_nr: {
242             bsonType: "int",
243           },
244           instructor_id: {
245             bsonType: "int",
246           },
247         }
248       }
249     }
250 });
251 db.createCollection("Location", {
252     validator: {
253       $jsonSchema: {
254         bsonType: "object",
255         title: "Location (relationship) Object Validation",
256         required: ["building_id", "dept_nr"],
257         properties: {
258           dept_nr: {
259             bsonType: "int",
260           },
261           building_id: {
262             bsonType: "string",
263           },
264         }
265       }
266     }
267 });
```

5.1.3 Results in MongoDb

After running the output of our export in MongoDb, we can see a new database called University as a file tree, and after expanding this, we can see all the added collections.

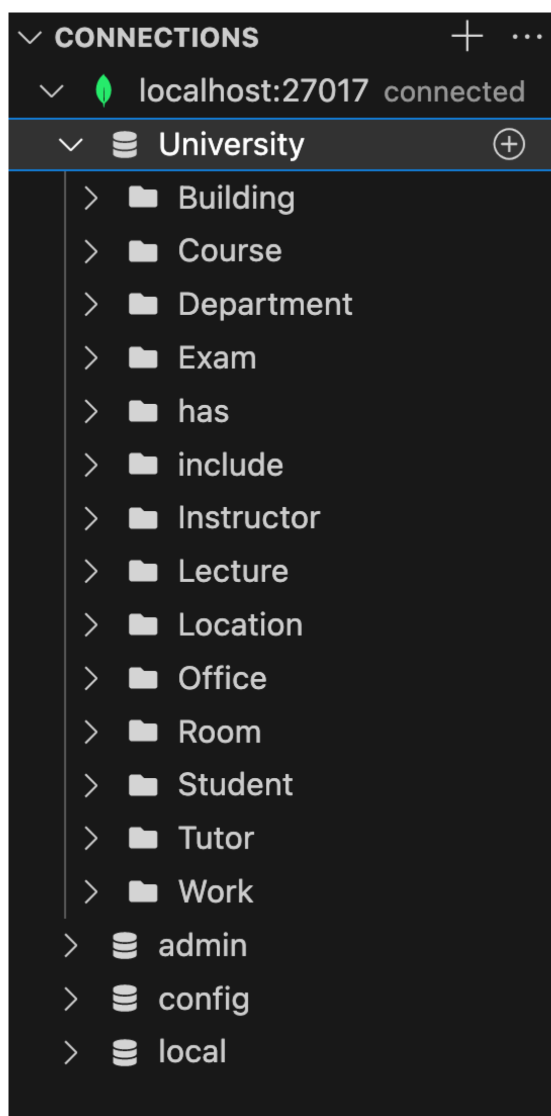


Figure 5.4: MongoDb database after the `js` output is run for University example.

To test the validation of the schematic restrictions, like the data types, we will first insert some correct test data:

```
const database = 'University';

// The current database to use.
use(database);
```

```
5
6 db.Tutor.insertMany([
7     {
8         id: 1,
9         Student_birthday: new Date("1995-12-17"),
10        subject: "Physics",
11        Student_name: "Luke Skywalker",
12        Student_age: 28,
13        Student_id: 1001
14    },
15    {
16        id: 2,
17        Student_birthday: new Date("1990-05-04"),
18        subject: "Computer Science",
19        Student_name: "Leia Organa",
20        Student_age: 33,
21        Student_id: 1002
22    },
23    {
24        id: 3,
25        Student_birthday: new Date("1985-07-19"),
26        subject: "Mathematics",
27        Student_name: "Han Solo",
28        Student_age: 38,
29        Student_id: 1003
30    }
31 ]);
```

This correct insert statement for the collection **Tutor** returns the following success message:

```
1 {
2   "acknowledged": true,
3   "insertedIds": {
4     "0": {
5       "$oid": "64749ae457e8456ed8034610"
6     },
7     "1": {
8       "$oid": "64749ae457e8456ed8034611"
9     },
10    "2": {
11      "$oid": "64749ae457e8456ed8034612"
12    }
13  }
14 }
```

And if we take a look at the file tree under the Tutor collection, 3 new documents can be found.

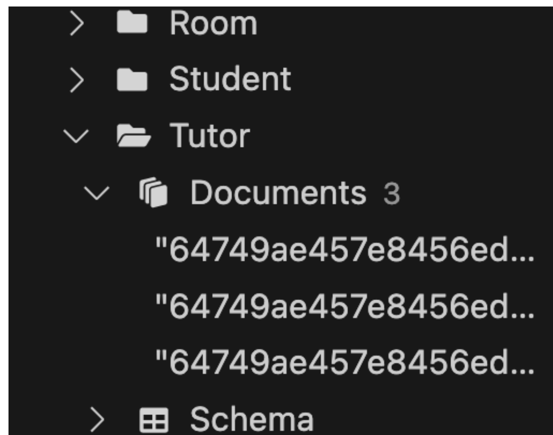


Figure 5.5: MongoDB Tutor collection after we insert correct test data.

Let's check, if the validation works, by changing the first Tutor object. Instead of a **Date** we will define a **String**, like so:

```
...
Student_birthday: new Date("1995-12-17") "wrong data type here",...
```

We can observe the following error message:

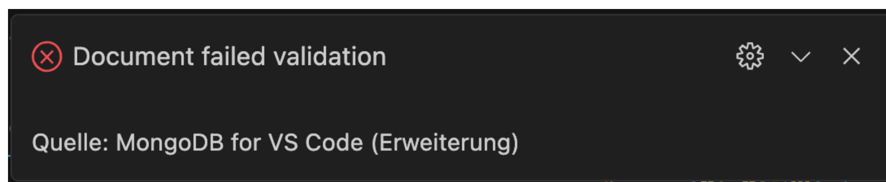


Figure 5.6: MongoDB error message after we insert false test data.

5.2 Neo4j Transformation

5.2.1 Neo4j AuraDB Setup

For our Neo4j demonstration, we will use the online Neo4j aura db [28], which is a cloud based service with access to instances of Neo4j databases, as well as an online graphical interface. In our case, it is Neo4j version 5.

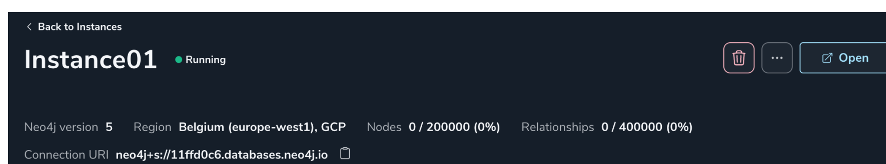


Figure 5.7: Neo4j instance for testing the transformation in aura db.

5.2.2 Neo4j output from bigER

If we export the University example of BIGER, we get the following output:

```
// entities
CREATE (Tutor:Tutor {name: "Tutor", Tutor_subject: "VARCHAR(255)",
    ↪ Student_birthday: "DATE", Student_id: "INT", Tutor_id: "INT",
    ↪ Student_age: "SMALLINT", Student_name: "VARCHAR(255)"})
CREATE (Student:Student {name: "Student", Student_birthday: "DATE",
    ↪ Student_id: "INT", Student_age: "SMALLINT", Student_name: "
    ↪ VARCHAR(255)"})
CREATE (Course:Course {name: "Course", Course_course_nr: "INT",
    ↪ Course_credits: "SMALLINT", Course_course_name: "VARCHAR(100)"
    ↪ })
CREATE (Lecture:Lecture {name: "Lecture", Lecture_title: "VARCHAR
    ↪ (255)"})
CREATE (Instructor:Instructor {name: "Instructor",
    ↪ Instructor_instructor_id: "INT", Instructor_name: "VARCHAR(255)
    ↪ "})
CREATE (Department:Department {name: "Department", Department_name: "
    ↪ VARCHAR(100)", Department_abbreviation: "CHAR(5)",
    ↪ Department_dept_nr: "INT"})
CREATE (Room:Room {name: "Room", Room_room_nr: "INT"})
CREATE (Building:Building {name: "Building", Building_address: "
    ↪ VARCHAR(255)", Building_building_id: "CHAR(8)"})
// relationships
CREATE (Exam:Exam{name: "Exam", Exam_points: "DOUBLE"})
CREATE (Exam)-[Exam_Student:Exam_Student {points: "DOUBLE" }]->(
    ↪ Student)
CREATE (Exam)-[Exam_Course:Exam_Course {points: "DOUBLE" }]->(Course)
CREATE (Exam)-[Exam_Instructor:Exam_Instructor {points: "DOUBLE"
    ↪ }]->(Instructor)
CREATE (Building)-[has:has{name: "has" }]->(Room)
CREATE (Room)-[Office:Office{name: "Office" }]->(Instructor)
CREATE (Department)-[Work:Work{name: "Work" }]->(Instructor)
CREATE (Course)-[include:include{name: "include" }]->(Lecture)
CREATE (Department)-[Location:Location{name: "Location" }]->(Building
    ↪ )
// extends relationships (IS_A)
CREATE (Tutor)-[Tutor_IS_A:Tutor_IS_A]->(Student)
// constraints
CREATE CONSTRAINT IF NOT EXISTS FOR (x:Tutor) REQUIRE x.
    ↪ Tutor_Tutor_id IS UNIQUE;
CREATE CONSTRAINT IF NOT EXISTS FOR (x:Student) REQUIRE x.
    ↪ Student_Student_id IS UNIQUE;
CREATE CONSTRAINT IF NOT EXISTS FOR (x:Course) REQUIRE x.
    ↪ Course_Course_course_nr IS UNIQUE;
CREATE CONSTRAINT IF NOT EXISTS FOR (x:Lecture) REQUIRE x.
    ↪ Lecture_Lecture_title IS UNIQUE;
```

```
27 CREATE CONSTRAINT IF NOT EXISTS FOR (x:Instructor) REQUIRE x.
    ↪ Instructor_Instructor_instructor_id IS UNIQUE;
28 CREATE CONSTRAINT IF NOT EXISTS FOR (x:Department) REQUIRE x.
    ↪ Department_Department_dept_nr IS UNIQUE;
29 CREATE CONSTRAINT IF NOT EXISTS FOR (x:Room) REQUIRE x.
    ↪ Room_Room_room_nr IS UNIQUE;
30 CREATE CONSTRAINT IF NOT EXISTS FOR (x:Building) REQUIRE x.
    ↪ Building_Building_building_id IS UNIQUE;
```

5.2.3 Results in Neo4j AuraDB

We need to keep in mind that the creation statements of nodes and relationships can be all run together in the Neo4j, although in that order. The constraints however, need to be run **after** the nodes and relationships are already created.

First we import all the nodes and relationships. Once this is success full, we can query graph, that contains all of the existing nodes and relationships like so:

```
1 MATCH (n)-[r]->(m)
2 RETURN n, r, m
```

The output yields the following graph:

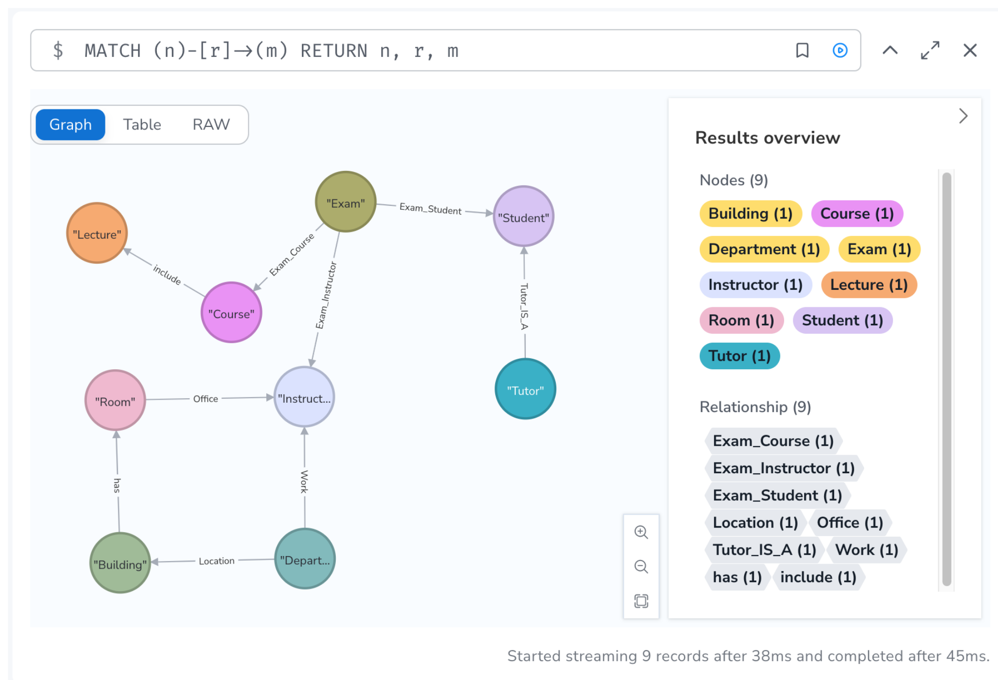


Figure 5.8: Neo4j resulting graph after import of er diagram export.

If we then run the creation statements for the constraints as well, we see the following success message for each constraint added:

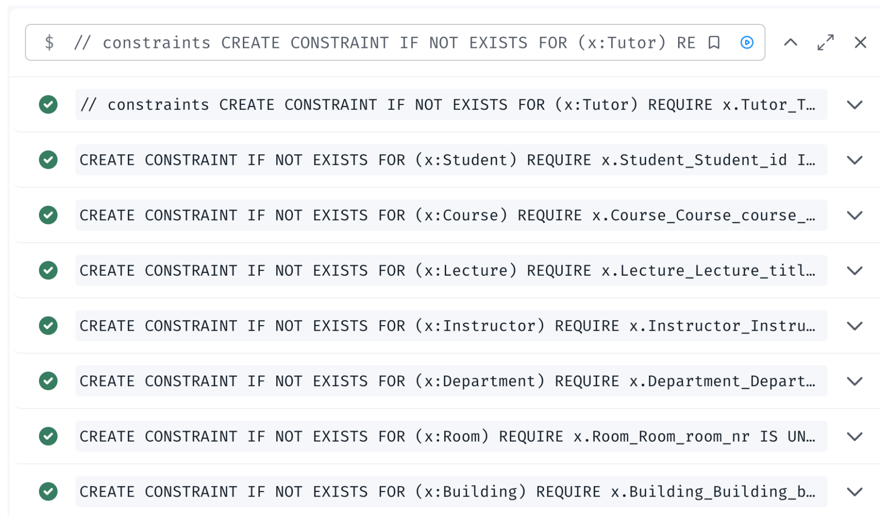


Figure 5.9: Neo4j resulting success message after constraints import.

5.3 Cassandra Transformation

To interact with Cassandra, the language CQL is used similar to the usual SQL scripts.

5.3.1 Cassandra output from export

We will first look at the generated output of BIGER of our university example:

```
CREATE KEYSPACE University WITH REPLICATION = {'class': '
    ↪ SimpleStrategy', 'replication_factor': 2};
USE University;
CREATE TABLE Student(
  name TEXT,
  age VARINT,
  birthday DATE,
  id INT,
  PRIMARY KEY (id)
);

CREATE TABLE Course(
  course_name TEXT,
  course_nr INT,
  credits VARINT,
  PRIMARY KEY (course_nr)
);

CREATE TABLE Lecture(
  title TEXT,
  PRIMARY KEY (title)
);

CREATE TABLE Instructor(
  name TEXT,
  instructor_id INT,
  PRIMARY KEY (instructor_id)
);

CREATE TABLE Department(
  abbreviation TEXT,
  dept_nr INT,
  name TEXT,
  PRIMARY KEY (dept_nr)
);

CREATE TABLE Room(
  room_nr INT,
  PRIMARY KEY (room_nr)
);

```

```
41 CREATE TABLE Building(
42     building_id TEXT,
43     address TEXT,
44    PRIMARY KEY (building_id)
45 );
46
47 CREATE TABLE Tutor(
48     Student_name TEXT,
49     Student_age VARINT,
50     Student_birthday DATE,
51     subject TEXT,
52     id INT,
53     Student_id INT,
54    PRIMARY KEY (id)
55 );
56
57 CREATE TABLE has(
58     room_nr INT,
59     building_id TEXT,
60    PRIMARY KEY (room_nr, building_id)
61 ) WITH comment='relationship';
62
63 CREATE TABLE include(
64     title TEXT,
65     course_nr INT,
66    PRIMARY KEY (title, course_nr)
67 ) WITH comment='relationship';
68
69 CREATE TABLE Exam(
70     Student_id INT,
71     course_nr INT,
72     instructor_id INT,
73     points DOUBLE,
74    PRIMARY KEY (Student_id, course_nr, instructor_id)
75 ) WITH comment='relationship';
76
77 CREATE TABLE Office(
78     room_nr INT,
79     instructor_id INT,
80    PRIMARY KEY (room_nr, instructor_id)
81 ) WITH comment='relationship';
82
83 CREATE TABLE Work(
84     instructor_id INT,
85     dept_nr INT,
86    PRIMARY KEY (instructor_id, dept_nr)
87 ) WITH comment='relationship';
88
89 CREATE TABLE Location(
```

```
90   building_id TEXT,
91   dept_nr INT,
92   PRIMARY KEY (building_id, dept_nr)
93 ) WITH comment='relationship';
```

5.3.2 Cassandra Database Setup

We again use Docker, in order to test out our generated creation script of the university example. For this, we can use the command `Docker pull Cassandra` from the terminal, which will give us access to the current Cassandra image, which we can run like so:

```
1 Docker run -d -p 9042:9042 --name my-Cassandra-container Cassandra:
    ↪ latest
```

This starts up a new Docker container using our Cassandra image.

- `Docker run`: starts the new Docker container.
- `-d`: Detached mode, meaning the container runs in the background.
- `-p 9042:9042`: This maps port 9042 on the host to port 9042 in the container (`-p host_port:container_port`).
- `-name my-Cassandra-container`: Names the container `my-Cassandra-container`, which is also used to refer to this container in other Docker commands.
- `Cassandra:latest`: The Docker image to use for the container, in this case, it's the latest version of the Cassandra image, which is (4.1).

We can run this script in Cassandra's own shell client called "cqlsh", which we can use from Docker desktop, the graphical user interface for Docker. This is conveniently installed already in our Cassandra image, so we can access it right away.

We will create a file in the home directory called **university.cql** which will contain our generated script. Then we can call **cqlsh** and type the following command: `SOURCE '/home/university.cql'`. This will tell cqlsh to use this script and run it.

Figure 5.10: Cassandra cqlsh in Docker desktop terminal interface, with SOURCE command.

5.3.3 Results in Cassandra

After executing the *SOURCE* command, the output looks as follows:

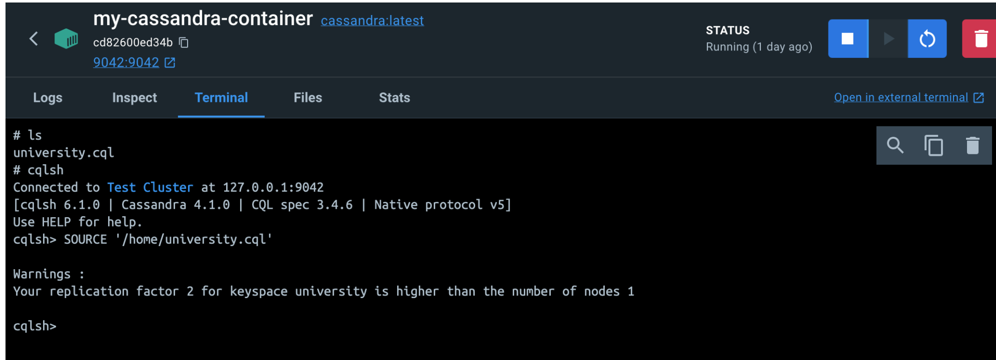


Figure 5.11: Cassandra cqlsh success message after running script.

In order to check, if everything got created, we can look at all the keyspaces in our cassandra instance with `cqlsh> DESCRIBE KEYSPACES;`, with the following result:

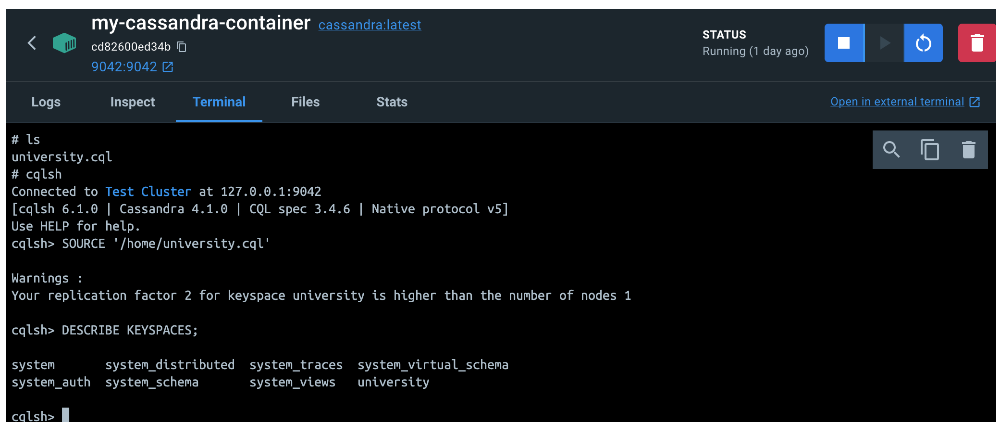


Figure 5.12: cqlsh showing all keyspaces.

As we can see, the keyspace university exists. Now we can use this keyspace to display all its tables:

```
1 cqlsh> USE university;
2 cqlsh:university> DESCRIBE TABLES;
```

This yields the following output:

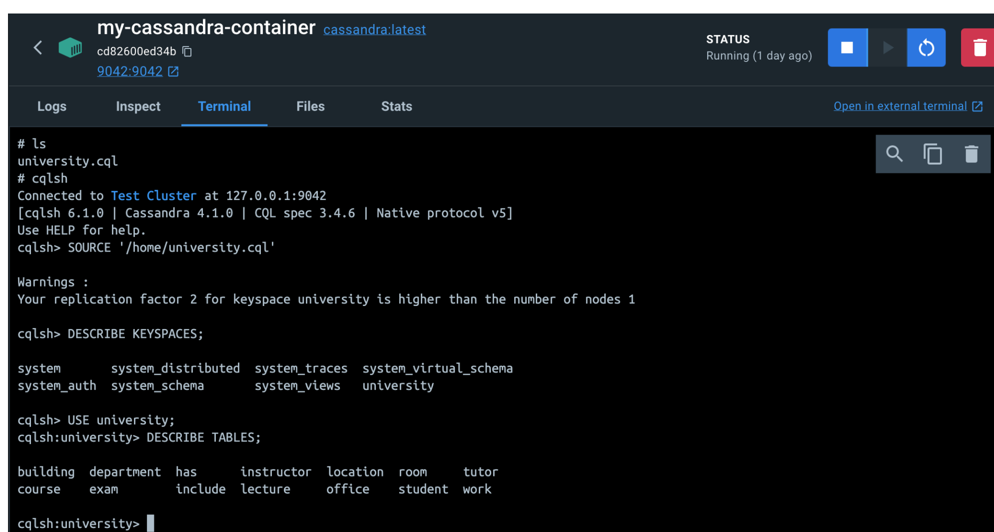


Figure 5.13: Cqlsh showing all tables of the university keyspace.

As we can see, all the tables do exist, which means our creation statements were run successfully. For further information about the current schema of the university keyspace, one can check it out with `DESCRIBE KEYSPACE university`, which will show the schema in form of a more detailed creation script.

5.4 Summary

We wanted to test our solution in each of the databases. To do that, we looked at the setup of each database, using docker containers or a cloud solution. We took our university example and exported it to all three database scripts. Using the databases we setup earlier, we imported and executed these creation scripts in MongoDB, Neo4j and Cassandra. We then evaluated the results and checked for correctness.

CHAPTER 6

Conclusion

In this thesis, we looked at the transformation of an ER model into several different NoSQL physical models. We achieved this, by using techniques to define, guide or force some sort of schema in inherently non-schema databases. We defined generators in BIGER for every one of our three supported NoSQL databases, which has access to the ER model and generates the NoSQL generation code and puts that output into the appropriate file.

Let's remind ourselves on what the objectives of this thesis were:

1. *Exploring the challenges and limitations of transforming ER diagrams into NoSQL data models, as well as importing NoSQL code back into BIGER.* We have seen how all our NoSQL databases work and how it's syntax looks like. All the limitations of our transformation had to do with the potential loss of information that results when exporting or importing code. For this we looked closely at the structure of the NoSQL databases and what features allow us to restrict and express database schemas. Using mapping tables, we have examined a base case and how the transformation should behave. In the implementation, we handled special cases, like for example the handling of ternary relationships for our graph database example Neo4j.

2. *How to design NoSQL schema code, even though some of those databases are schema-less by nature?* For MongoDB, we used the schema validation feature in order to enforce our structure for the entities and relationships and their fields. Here we dealt with the lacking foreign keys by adding them all as required fields, so as to ensure there existence. This can be extended further in future work, with additional validation functionality. In Neo4j, we replicated the ER model using nodes and edges, the result is a graph which represents the model visually. We also added all the fields into the nodes and edges, so as to more precisely describe the model. And finally, we used Cassandra's already familiar, SQL like creation

statements to add all entities and relationships as tables. However, as there is no support for foreign keys, so here we defined them just as keys. Future work may be directed into denormalizing the model so as to more precisely reflect queries being for performance optimizations.

3. *Contribution to bigER of Xtend based Code Generators to generate code for document based, column-family and graph based NoSQL databases.* Using the language Xtend, we have created Code Generators for each of the three different NoSQL database types. All three have support for the extends relationships with folding attributes to the extended entities and thus represent them in this way in the output code. We also used regular expressions to parse back the NoSQL code and produce back the .erd file.

Some consideration for future work:

1. Exporting NoSQL code with more information and constraints in order to more precisely express the schema. Not only will it benefit the import functionality by reducing information loss, but also the user will have better representation of the schema.
2. NoSQL optimizations could be made by introducing denormalization for Cassandra. By generating tables to represent actual queries made by the user, read operation will gain significant performance boosts. This could be done by introducing new UI elements into bigER to allow them for adding redundancies into the model's exported code.
3. Utilization of artificial intelligence for aiding design process for data modeling itself. Similarly to the generation of models into .erd syntax, as can be seen in my example with ChatGPT in the appendix 6, a feature could implement an API to such a service, allowing the user to directly interact with the Model via a chat interface.

Bibliography

[1] *Apache cassandra is an open source, distributed, nosql database. it presents a partitioned wide column storage model with eventually consistent semantics.*, https://cassandra.apache.org/doc/4.1/cassandra/architecture/overview.html.

[2] *Build applications and work with your data in mongodb directly from your vs code environment.*, https://www.mongodb.com/products/vs-code.

[3] *Cassandra provides the cassandra query language (cql), an sql-like language, to create and update database schema and access data.*, https://cassandra.apache.org/doc/4.1/cassandra/architecture/overview.html#features.

[4] *Chatgpt get instant answers, find creative inspiration, and learn something new.*, https://openai.com/product/chatgpt.

[5] *Data modeling: Conceptual vs logical vs physical data model*, https://online.visual-paradigm.com/knowledge/visual-modeling/conceptual-vs-logical-vs-physical-data-model/.

[6] *Dbschema is an advanced database diagram designer and query tool that allows developers, analysts, and database administrators to visually design, manage, document, and understand their databases.*, https://dbschema.com/.

[7] *Docker takes away repetitive, mundane configuration tasks and is used throughout the development lifecycle for fast, easy and portable application development – desktop and cloud.*, https://www.docker.com/.

[8] *Eclipse rcp (rich client platform) applications, use the eclipse framework to create feature-rich stand-alone desktop applications.*, https://www.vogella.com/tutorials/EclipseRCP/article.html.

[9] *Eclipse xtext is a framework for development of programming languages and domain-specific languages.*, https://www.eclipse.org/Xtext/.

[10] *language-server-protocol.*, https://microsoft.github.io/language-server-protocol/.

[11] *List of data types in cassandra.*, https://cassandra.apache.org/doc/4.1/cassandra/cql/types.html.

[12] *Luna modeler is a powerful data modeling tool for relational databases for sql server, postgresql, mariadb, mysql and sqlite database design.*, https://www.datensen.com/.

[13] *The metamodel of the er language is generated out of the xtext grammar and is located in org.big.erd/model/generated.*, https://github.com/borkdominik/bigER/wiki/Metamodel.

[14] *Mongodb document databases provide high availability and easy scalability.*, https://hub.docker.com/_/mongo/.

[15] *Mongodb helps remove barriers to innovation for organizations in any industry.*, https://www.mongodb.com/.

[16] *Mongodb is a general-purpose document database designed for modern application development and for the cloud.*, https://www.mongodb.com/basics.

[17] *Neo4j etl tool - interactive relational database data import*, https://neo4j.com/developer/neo4j-etl/.

[18] *A neo4j graph database stores nodes and relationships instead of tables or documents.*, https://neo4j.com/docs/getting-started/get-started-with-neo4j/graph-database/.

[19] *An open, flexible and extensible cloud & desktop ide platform.*, https://theia-ide.org/.

[20] *Open-source er modeling tool for vs code supporting hybrid, textual- and graphical editing, multiple notations, and sql code generation!*, https://github.com/borkdominik/bigER.

[21] *Polyglot data modeling for nosql databases, storage formats, rest apis, and json in rdbms.*, https://hackolade.com/.

[22] *Schema validation lets you create validation rules for your fields, such as allowed data types and value ranges.*, https://www.mongodb.com/docs/manual/core/schema-validation/.

[23] *A simple strategy that defines a replication factor for data to be spread across the entire cluster.*, https://cassandra.apache.org/doc/4.1/cassandra/cql/ddl.html#replication-strategy.

[24] *This is the client part of sprotty, a next-generation, open-source diagramming framework built with web technologies.*, https://github.com/eclipse-sprotty/sprotty.

[25] *This is the client part of sprotty, a next-generation, open-source diagramming framework built with web technologies.*, https://projects.eclipse.org/projects/ecd.sprotty.

[26] *This page describes how the tool can be extended to include a new code generator.*, https://github.com/borkdominik/bigER/wiki/New-Code-Generator.

[27] *Visual-studio-code marketplace: biger modeling tool*, https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.erdiagram.

[28] *The world's leading graph database as a fully-managed cloud service — zero-admin, globally available and always-on.*, https://neo4j.com/cloud/platform/aura-graph-database/?ref=docs-nav-get-started.

[29] *Xtend is a flexible and expressive dialect of java, which compiles into readable java 8 compatible source code.*, https://www.eclipse.org/Xtend/.

[30] Fatma Abdelhedi, Amal Ait Brahim, Faten Atigui, and Gilles Zurfluh, *Mda-based approach for nosql databases modelling*, Big Data Analytics and Knowledge Discovery (Cham) (Ladjel Bellatreche and Sharma Chakravarthy, eds.), Springer International Publishing, 2017, pp. 88–102.

[31] Dominik Bork, Philip Langer, and Tobias Ortmayr, *A vision for flexibile GLSP-based web modeling tools*, CoRR (2023).

[32] Giuliano De Carlo, Philip Langer, and Dominik Bork, *Advanced visualization and interaction in glsp-based web modeling: realizing semantic zoom and off-screen elements*, Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, ACM, 2022, pp. 221–231.

[33] Peter Pin-Shan Chen, *The entity-relationship model—toward a unified view of data*, ACM Trans. Database Syst. **1** (1976), no. 1, 9–36.

[34] Philipp-Lorenz Glaser, *Developing sprotty-based modeling tools for vs code*, 2022, https://model-engineering.info/publications/theses/thesis-glaser.pdf.

[35] Philipp-Lorenz Glaser and Dominik Bork, *The bigER Tool – Hybrid Textual and Graphical Modeling of Entity Relationships in VS Code*, 25th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2021, IEEE International Enterprise Distributed Object Computing Conference (EDOC), 10 2021.

[36] Esther Guerra and Juan de Lara, *On the quest for flexible modelling*, Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (2018).

[37] Statista Inc., *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025 (in zettabytes).*, https://www.statista.com/statistics/871513/worldwide-data-created/.

[38] Haydar Metin and Dominik Bork, *On developing and operating glsp-based web modeling tools: Lessons learned from bigUML*, Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, IEEE, 2023.

[39] Louis M. Rose, Dimitrios S. Kolovos, and Richard F. Paige, *Eugenia live: a flexible graphical modelling tool*, XM '12, 2012.

[40] K. Shin, C. Hwang, and H. Jung, *Nosql database design using uml conceptual data model based on peter chen's framework*, International Journal of Applied Engineering Research **12** (2017), 632–636.

List of Tables

4.1 Mapping from ER diagram to MongoDB 22
4.2 Example Mapping from ER Diagram to MongoDB 22
4.3 Mapping of MongoDB Data Types to SQL Data Types 26
4.4 Mapping from ER Diagram to Neo4j 34
4.5 Example Mapping from ER Diagram to Neo4j 34
4.6 Mapping from ER Diagram to Cassandra 43
4.7 Example Mapping from ER Diagram to Cassandra 44

List of Figures

2.1 Visual Top-Level View on the Model Driven Architecture 4
2.2 Basic example of a simple er diagram showing its components. 5
2.3 Building blocks of the property graph model as in official documentation [18]. 9

3.1 The Metamodel of bigER [13] 16
3.2 The Metamodel of bigER [13] 17

4.1 The simple ER-diagram example for Transformation. 22
4.2 Preview of the neo4j-etl transformation tool shows schema as Neo4j graph. 32
4.3 The extends relationship mapped in neo4j graph as IS_A relationship. . . 33
4.4 Ternary relationship in university example in bigER. 33
4.5 Ternary relationship in university example as Neo4j graph. 33
4.6 The simple ER-diagram example for Transformation. 35
4.7 The resulting graph from the building-has-room example. 36

5.1 University example diagram. 57
5.2 University example diagram. 57
5.3 Docker MongoDb image in dashboard. 58
5.4 MongoDb database after the `js` output is run for University example. . . 64
5.5 MongoDB Tutor collection after we insert correct test data. 66
5.6 MongoDB error message after we insert false test data. 66
5.7 Neo4j instance for testing the transformation in aura db. 66
5.8 Neo4j resulting graph after import of er diagram export. 68
5.9 Neo4j resulting success message after constraints import. 69
5.10 Cassandra cqlsh in Docker desktop terminal interface, with SOURCE command. 72
5.11 Cassandra cqlsh success message after running script. 73
5.12 cqlsh showing all keyspaces. 73
5.13 Cqlsh showing all tables of the university keyspace. 74

A.1 ER-diagram generated by ChatGPT. 86

APPENDIX A

Generating ER-Diagrams with ChatGPT

As a bonus chapter, we will see the effectiveness of Open Ai's **ChatGPT** [4] when learning the required syntax of the .erd files and with it generating completely new ER-diagrams.

A.1 What is ChatGPT?

ChatGPT is a model of language that was made by OpenAI. It is run by a machine learning model called GPT-3 (Generative Pretrained Transformer 3), which is meant to make writing that sounds like it was written by a person based on what it is told to write.

The prototype was launched on November 30, 2022. Since then, it has gained massive traction because of it's accurate and human-like text generation. It's output is generated by predicting which word most likely should come next, based on it's trained model. But it is not limited to human-like texts, it also can generate computer code, and even poetry, among many other things.

The generated examples int this thesis where performed in version **ChatGPT 4**.

A.2 The Prompt

A **prompt** is the text input that is given to ChatGPT, which instructs it on what output to generate. Results may widely vary, but in general, the more precise the prompt, the better the result could be.

Here is the instruction given to CHATGPT as of 27th of May, 2023:

```
take a look at the following syntax of the bigER open source tool,
    ↪ which describes an er diagram with entities,
relationships and even extends relationships. Learn this pattern and
    ↪ make me a new er diagram based on that syntax,
but i want one with 10 entities and minimum 2 extends statements, and
    ↪ minimum 8 relationships. Make the theme of it Start Wars.
here is an example of an er diagram:
// ER Model
erdiagram University

// Options
notation=default

// Entities
entity Tutor extends Student {
    id: INT key
    subject: VARCHAR(255)
}
entity Student {
    id: INT key
    name: VARCHAR(255)
    birthday: DATE
    age: SMALLINT
}
entity Course {
    course_nr: INT key
    course_name: VARCHAR(100)
    credits: SMALLINT
}
weak entity Lecture {
    title: VARCHAR(255) partial-key
}
entity Instructor {
    instructor_id: INT key
    name: VARCHAR(255)
}
entity Department {
    dept_nr: INT key
    name: VARCHAR(100)
    abbreviation: CHAR(5)
}
weak entity Room {
    room_nr: INT partial-key
}
entity Building {
    building_id: CHAR(8) key
    address: VARCHAR(255)
}

```

```
47 // Relationships
48 relationship Exam {
49     Student[1]  -> Course[N] -> Instructor[N]
50     points: DOUBLE
51 }
52 weak relationship has {
53     Room[N] -> Building[1]
54 }
55 relationship Office {
56     Room[1] -> Instructor[1]
57 }
58 relationship Work {
59     Instructor[N] -> Department[1]
60 }
61 weak relationship include {
62     Course[1] -> Lecture[N]
63 }
64 relationship Location {
65     Building[N] -> Department[1]
66 }
```

Using the above command, and our already fammiliar university.erd example, **chatGPT** generated a diagram, but this lacked the second extends statement. This could be mitigated by the following command: `but there is only 1 extends statement, i wanted 2` By providing minimal instructions, as well as the overall theme of the new diagram to be generated, CHATGPT was able to understand the syntax and semantically interpret our instructions.

A.3 Results

In our example, the theme of **Star Wars** was chosen, and the resulting diagram, after importing it into BIGER, looks like this:

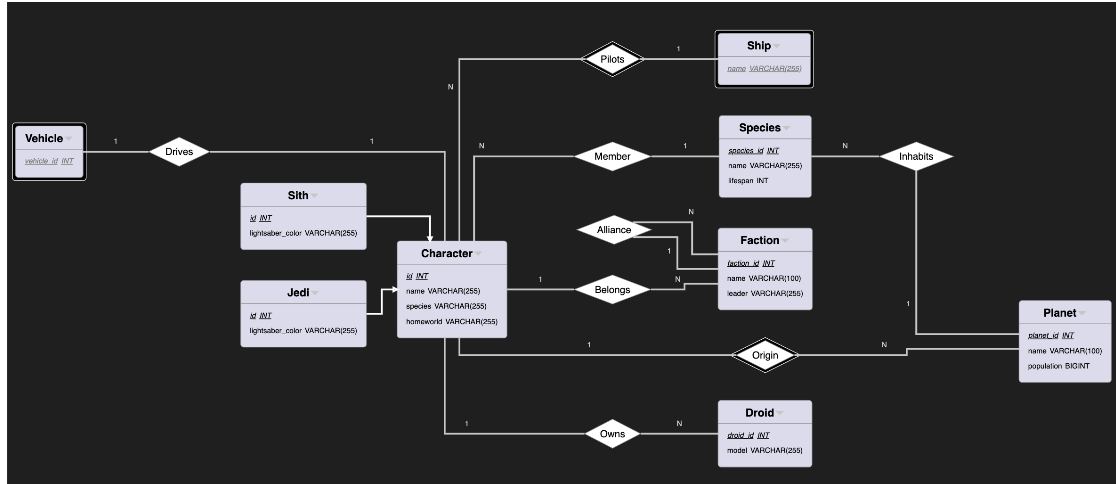


Figure A.1: ER-diagram generated by ChatGPT.

The result of that is the following star wars themed ER-diagram:

```
1  // ER Model
2  erdiagram StarWars
3
4  // Options
5  notation=default
6
7  // Entities
8  entity Jedi extends Character {
9      id: INT key
10     lightsaber_color: VARCHAR(255)
11 }
12 entity Sith extends Character {
13     id: INT key
14     lightsaber_color: VARCHAR(255)
15 }
16 entity Character {
17     id: INT key
18     name: VARCHAR(255)
19     species: VARCHAR(255)
20     homeworld: VARCHAR(255)
21 }
22 entity Planet {
23     planet_id: INT key
24     name: VARCHAR(100)
25     population: BIGINT
```

```
26 }
27 weak entity Ship {
28     name: VARCHAR(255) partial-key
29 }
30 entity Droid {
31     droid_id: INT key
32     model: VARCHAR(255)
33 }
34 entity Faction {
35     faction_id: INT key
36     name: VARCHAR(100)
37     leader: VARCHAR(255)
38 }
39 weak entity Vehicle {
40     vehicle_id: INT partial-key
41 }
42 entity Species {
43     species_id: INT key
44     name: VARCHAR(255)
45     lifespan: INT
46 }
47
48 // Relationships
49 relationship Belongs {
50     Character[1]  -> Faction[N]
51     rank: VARCHAR(255)
52 }
53 weak relationship Pilots {
54     Character[N] -> Ship[1]
55 }
56 relationship Drives {
57     Vehicle[1] -> Character[1]
58 }
59 relationship Member {
60     Character[N] -> Species[1]
61 }
62 weak relationship Origin {
63     Character[1] -> Planet[N]
64 }
65 relationship Alliance {
66     Faction[N] -> Faction[1]
67 }
68 relationship Inhabits {
69     Species[N] -> Planet[1]
70 }
71 relationship Owns {
72     Character[1] -> Droid[N]
73 }
```

We do now have two extends statements, 8 relationships, like we instructed. We do find only 9 entities, but this could be resolved like we did with the lacking extend statement.

As we see, this result is a fascinating display of the state of artificial intelligence. This could be used to aid developers, teams and even teachers, to more quickly design er diagrams and perhaps aid in the reconfiguration of existing er diagrams e.g. if some big changes occur in the data model, ChatGPT could perhaps rearrange the .erd file much more proficiently and quickly. Of course, oversight is still needed, as we saw in our example, simple things like generating the right amount of entities is still challenging to the large language model.