EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Database Systems Research Group

Masterthesis Computer Science

Mapping GraphQL Queries to SQL

Lukas Sailer

28.09.2023

Examiner

Prof. Dr. Torsten Grust

Co-Examiner

Prof. Dr. Klaus Ostermann

Supervisors

Jakob Thiersch
Thomas Graf

Lukas Sailer:
Mapping GraphQL Queries to SQL
Masterthesis Computer Science
Eberhard Karls Universität
From 01.05.2023 to 28.09.2023

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterthesis selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterthesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum

Lukas Sailer

Acknowledgement

I am deeply grateful to all those who have played a role in helping me complete this thesis.
First and foremost, my heartfelt thanks go to Prof. Dr. Torsten Grust for his continued interest in my work and invaluable guidance, without which I would not have had the privilege to pursue this thesis.
I extend my appreciation to my supervisors Jakob Thiersch and Thomas Graf for their insightful feedback and constructive criticism, which significantly improved the quality of my work.
I am grateful to the entire team at itdesign for their support and encouragement throughout this journey. Special recognition goes to Team Core for their unwavering support.
Last but certainly not least, I am deeply indebted to my friends and family for their unwavering support and motivation. Without them, not only this thesis, but my entire academic journey would not have been possible.

Abstract

This thesis explores the challenges of addressing the N+1 problem in GraphQL Application Programming Interface (API) implementations using a generic mapping approach. GraphQL is a dynamic query language that allows API callers to issue queries on a graph. A common approach to query resolution is to use resolvers to individually resolve each requested field, which corresponds to an edge within the graph. Because the resolvers fetch data for each field from a data source, typically a Structured Query Language (SQL) database, this may lead to performance issues when resolving one-to-many relationships. The N+1 problem occurs when a single resolver is called multiple times, resulting in unnecessary SQL queries. Although the GraphQL DataLoader utility [1] can assist in alleviating this problem, it still creates multiple SQL queries. This thesis aims to address the N+1 problem by developing a mapping approach that maps any given GraphQL query to exactly one SQL query, thus reducing the number of SQL queries and improving performance.

This study evaluates existing solutions, which provide a generic mapping of GraphQL queries to SQL queries and explores their limitations with respect to independent schema modeling. The proposed approach aims to overcome these limitations and provide a more efficient and flexible solution for GraphQL API development, further reducing boilerplate code and maintenance effort.

This thesis introduces a generic GraphQL-to-SQL mapping approach that addresses the N+1 problem. The approach yields a single SQL query for GraphQL queries containing one root field. GraphQL queries with multiple root fields result in one SQL query per root field. Furthermore, an implementation for the approach is included, which highlights the benefits of using jOOQ Object Oriented Querying (jOOQ) [2]. The implementation enables the use of a mapping with minimal configuration and results in a significant performance improvement over the use of resolvers.

Contents

Acronyms

API Application Programming Interface
AST Abstract Syntax Tree
BYOS Bring Your Own Schema
CQRS Command Query Responsibility Segregation
CRUD Create, Read, Update and Delete
CTE Common Table Expression
DSL Domain-Specific Language
HTTP Hypertext Transfer Protocol
jOOQ jOOQ Object Oriented Querying
JSON JavaScript Object Notation
JVM Java Virtual Machine
RDBMS Relational Database Management Systems
REST Representational State Transfer
SDL Schema Definition Language
SQL Structured Query Language

1 Introduction

The purpose of this first chapter is to provide an overview of the objective and content of this thesis. Therefore, the motivation, the problem statement, and the objectives are presented. Finally, an overview of the structure of the thesis is outlined.

1.1 Motivation

Web Application Programming Interfaces (APIs) are a central part of modern software development, enabling data exchange between servers and client applications [3]. GraphQL serves as a modern approach to building APIs that introduces a different set of principles compared to traditional Representational State Transfer (REST) APIs [4]. While REST follows a fixed endpoint structure, GraphQL allows clients to dynamically explore a graph that represents the data. This dynamic nature of GraphQL presents challenges for software developers, including the N+1 problem. Furthermore, using the typical solution of resolving edges individually leads to similar or redundant code and consequently to substantial manual effort in maintaining a GraphQL API. These challenges underscore the need for an efficient solution. This thesis aims to provide a solution by offering a generic mapping of GraphQL queries to Structured Query Language (SQL) queries, effectively addressing both issues.

1.2 Problem statement

The N+1 problem is a well-known database query issue, which can have a negative impact on performance [5]. It occurs when N relations in a table are requested along with associated relations in another table. To illustrate this, consider the example of films and actors. Each film is associated with a group of actors who star in it. If N films are requested, along with their corresponding actors, the N+1 problem and its related performance issues may be encountered. The server would initially request N films in one query and then execute one query for the actors of each film, resulting in a total of N+1 queries.

The N+1 problem is also frequently encountered in GraphQL server implementations [6]. Listing 1.1 illustrates how a problematic query results in multiple SQL queries. As shown in the example, the related objects in the inner table are loaded separately for each of the N objects, instead of all together [5]. This occurs because a GraphQL server typically resolves queries employing resolvers. Thus, for each field, a resolver specifies how that field can be queried from the database.

```
query {
  allFilms {
    title                # 1x SELECT * FROM film;
    actors {
      last_name          # Nx SELECT * FROM actor JOIN film_actor ON ... ;
    }
  }
}
```

Listing 1.1: GraphQL query that requests all films with their actors. The comments represent the SQL queries that could result from this request.

A prevalent method of addressing this issue is to use the DataLoader pattern (see Section 2.3.1), which is a way to batch requests to the database [6]. This approach should ideally result in one request per object type (i.e., table) requested in the query. Although one query per table already scales better than the N+1 queries resulting from the naive approach, the data could theoretically be requested in only one single query. A single request reduces both network and database overhead. This is particularly significant in distributed systems with higher latency. There is also more potential for optimization by the database engine when using a single complex query.

Another issue that results from using resolvers is that their implementation leads to boilerplate code. Despite the fact that the data fetching logic may be identical for each field, it must be written out separately for each field's resolver. This redundancy can lead to difficulties in maintaining and scaling codebases for complex GraphQL schemas that have many connections and types.

1.3 Objectives

The main objective of this thesis is to propose a solution to the N+1 problem that constructs exactly one SQL query for any given GraphQL query. The proposed solution should operate in a manner that imposes minimal restrictions on both the GraphQL and SQL schemas.

The solution should not use a resolver approach and therefore minimize the redundancy and amount of code that has to be written when the schema is expanded.

A working prototype of the solution should be implemented and made available at `https://github.com/lukassailer/byos`.

1.4 Thesis structure

Chapter 1 Introduction

At the beginning of the thesis, the motivation, the problem statement and the objectives are described. Additionally, a brief summary of the chapters is provided.

Chapter 2 Background and Related Work

Chapter 2 is dedicated to the context of the thesis. A short overview of the GraphQL language is provided, as well as an introduction to the GraphQL Abstract Syntax Tree (AST). Different existing approaches for mapping GraphQL to SQL are considered. Furthermore, the drawbacks they bring with them are evaluated.

Chapter 3 Methodology

Chapter 3 explains the approach used for this thesis. First, the transpilation of a GraphQL AST into a SQL query is described. Subsequently, the challenges as well as the restrictions regarding this approach are examined. Lastly, the use cases for SQL views are listed with respect to this approach.

Chapter 4 Implementation

The fourth chapter explains how the logic outlined in Chapter 3 was implemented and used in a prototype. It details the technologies that were selected and how they were utilized in the construction and consumption of the internal query representation. It also provides an overview and explanation of the components of the prototype project.

Chapter 5 Discussion

In Chapter 5 the implementation from Chapter 4 is compared to the naive approach. Following that, limitations and constraints, as well as potential areas for improvement and future work, are discussed.

Chapter 6 Conclusion

The last chapter of the thesis summarizes the contents of all chapters. Finally, it is evaluated if the goals of the thesis were achieved.

2 Background and Related Work

This chapter provides an overview of the key concepts of GraphQL, explores the GraphQL AST and examines existing approaches for mapping GraphQL to SQL. The proposed solution, explained in the following chapter, is based on this.

2.1 Overview of GraphQL

GraphQL is an API query language originally developed by Meta Platforms Inc. (then Facebook Inc.) in 2012 [7]. It was open-sourced in 2015 and moved to the GraphQL Foundation in 2018 [8].

GraphQL is agnostic to programming languages and storage systems [9]. A GraphQL API is defined by a GraphQL schema, consisting of types and their respective fields. Each field is resolved by its own function. Such a resolver-function handles data fetching for its field. A client calling the GraphQL API can dynamically construct a query based on the schema that fetches exactly the data they require and nothing more.

The subsections of this section discuss specific aspects of the GraphQL language that are relevant throughout this paper. They are not and do not aim to be a complete introduction to GraphQL. In fact, they are strongly based on the GraphQL documentation, which offers in-depth explanations of all concepts in detail [10, 11].

2.1.1 Queries

As shown by the GraphQL query, shown in Listing 1.1, GraphQL allows the client to request exactly the information they need. In this example, the client uses the `allFilms` query, which returns a list of films (as specified on the `Query` type in the schema, see Section 2.1.2). The query is defined to include the titles of all films as well as the last names of all actors who starred in each film. The response is a JavaScript Object Notation (JSON) object called `data` that contains the requested data in the same structure as it appears in the query. A possible response to this exemplary query can be seen in Listing 2.1. In this case `data` would contain an array `allFilms` that in turn contains objects each with a field `title` and a field `actors`. The value of `actors` is an array in which each object only has the field `last_name`.

```
{
  "data": {
    "allFilms": [
      {
        "title": "ACADEMY DINOSAUR",
        "actors": [
          {
            "last_name": "GUINESS"
          },
          {
            "last_name": "GABLE"
          },
          ...
        ]
      },
      {
        "title": "ACE GOLDFINGER",
        "actors": [
          {
            "last_name": "FAWCETT"
          },
          {
            "last_name": "ZELLWEGER"
          },
          ...
        ]
      },
      ...
    ]
  }
}
```

Listing 2.1: Example of a JSON response to the query in Listing 1.1.

As seen in the example in Listing 2.2, GraphQL queries can also specify arguments. In this example, a `film_id` is provided, and only one film object is returned instead of an array. If there is no film with the provided id `filmById` is `null`. Arguments do not have to be passed to the query field, but are possible on every field that the schema defines arguments for.

```
query {
  filmById(film_id: 1) {
    title
    release_year
  }
}
```

Listing 2.2: GraphQL query that requests the title and release year for the film with id 1.

Queries can also use aliases for fields. These change the names of fields in the result. This is mainly useful when the same field is requested multiple times with different arguments, since it is not possible for multiple instances of the same field with different arguments to have the same name. Listing 2.3 shows an example in which films released in 1999 and 2000 are requested. The resulting data object contains one array `films1999` and one array `films2000` with the respective objects.

```
query {
  films1999: filmsByYear(release_year: 1999) {
    title
  }
  films2000: filmsByYear(release_year: 2000) {
    title
  }
}
```

Listing 2.3: Query containing the same field twice with different aliases. For each one a different argument value is used.

Although GraphQL also supports mutations for data manipulation and subscriptions for real-time updates, these are not the focus of this paper and are therefore not discussed in detail.

2.1.2 Schemas and types

The GraphQL schema describes a set of types and their fields, which explains what data can be requested and how it can be obtained. This schema can be used to validate and execute queries. A GraphQL schema can be defined in the Schema Definition Language (SDL), which serves as a human-readable way to declare types and their relationships.

A simple example of a GraphQL schema defined in SDL can be seen in Listing 2.4. The schema defines the fields of the `Query` type that were used in previous examples. Additionally, two object types and their fields are defined. One for films and one for actors. The fields of the `Query` type determine what can be requested at the top level. The field `allFilms` is an example. Its type is a list of `Film` objects. This permits requesting any field on the `Film` type on that field, some of these have scalar types, like `String`, and some have other object types, allowing for further nesting.

```
type Query {
  allFilms: [Film!]!
  filmById(film_id: ID!): Film
  filmsByYear(release_year: Int): [Film!]!
}

type Film {
  title: String!
  release_year: Int
  actors: [Actor!]!
}

type Actor {
  first_name: String!
  last_name: String!
}
```

Listing 2.4: Examplary GraphQL schema defining the `Query` type and two object types, one for films and another for actors.

In GraphQL, a type (like `String`) is nullable by default. Only if it is followed by an exclamation mark, it becomes non-nullable (`String!`). The same is true for lists. GraphQL can differentiate between nullable lists and non-nullable lists, as well as lists of non-nullable and nullable objects. Table 2.1 gives an overview of the permutations of the nullability of lists and their contents. Since this thesis focuses on GraphQL APIs with an SQL database as the data source, nullable lists and lists of nullable objects are not considered in greater detail. However, it should be mentioned that they can be useful for other data sources due to the bubble-up behavior of GraphQL [9].

Type declaration	Definition
`[Actor]`	A nullable list of nullable objects
`[Actor!]`	A nullable list of non-nullable objects
`[Actor]!`	A non-nullable list of nullable objects
`[Actor!]!`	A non-nullable list of non-nullable objects

Table 2.1: Different options for list nullability as described by Porcello and Banks [12].

It should also be noted that nullability plays a role not only in GraphQL responses but also in queries, since arguments can be nullable as well. The schema in Listing 2.4 defines the `release_year` for `filmsByYear` to be nullable. This means that the argument is optional, but also that `null` can be provided which would yield all films without a `release_year` (which is applicable since the schema defines the field `release_year` of the `Film` type to be nullable).

2.2 The GraphQL AST

An AST is a hierarchical structure used by compilers to represent code in a way that can be processed algorithmically [13]. In GraphQL, the AST serves as an intermediate stage during query processing, encapsulating the structure of a GraphQL query. After obtaining the AST from the GraphQL query, the query is first validated and then executed by traversing the AST.

The GraphQL AST, as explained by Porcello and Banks [14], has a GraphQL `Document` as the root node. It can define one or more operations, one of which is selected. An `OperationDefinition` has an `operation` field, indicating the type of operation (for this thesis, only the `QUERY` type is important), and a `SelectionSet`. The `SelectionSet` represents the fields that have been requested on this type and, therefore, consists of field nodes. Some of these field nodes may themselves have a nested `SelectionSet`. Listing 2.5 shows the GraphQL AST resulting from the query in Listing 1.1. This AST can be used to validate the query against the schema in Listing 2.4.

2.3 Existing approaches for mapping GraphQL to SQL

GraphQL is not dependent on any particular data source. Data can be stored in various formats or distributed across multiple sources. In a common scenario, the data is stored within a SQL database, requiring the derivation of essential SQL queries from GraphQL queries. This section introduces both the conventional method of mapping GraphQL to SQL and various generic approaches.

2.3.1 Resolvers and DataLoaders

GraphQL allows the dynamic construction of queries. A GraphQL server needs to be able to understand any query that adheres to the schema and return the expected result. The usual way to resolve a GraphQL query is to use resolvers, as defined in the GraphQL specification [9]. A resolver is a function that determines how to fetch the value of a field requested. If the data source is a SQL database, this means that the resolver fetches for its field via a SQL query.

```
Document {
  definitions = [
    OperationDefinition {
      name = 'null',
      operation = QUERY,
      variableDefinitions = [],
      directives = [],
      selectionSet = SelectionSet{
        selections = [
          Field {
            name = 'allFilms',
            alias = 'null',
            arguments = [],
            directives = [],
            selectionSet = SelectionSet{
              selections = [
                Field {
                  name = 'title',
                  alias = 'null',
                  arguments = [],
                  directives = [],
                  selectionSet = null
                },
                Field {
                  name = 'actors',
                  alias = 'null',
                  arguments = [],
                  directives = [],
                  selectionSet = SelectionSet{
                    selections = [
                      Field {
                        name = 'last_name',
                        alias = 'null',
                        arguments = [],
                        directives = [],
                        selectionSet = null
                      }
                    ]
                  }
                }
              ]
            }
          }
        ]
      }
    }
  ]
}
```

Listing 2.5: GraphQL AST derived from the query in Listing 1.1.

Resolving every field with its own SELECT statement, while being a clear and modular solution, results in the aforementioned N+1 problem. To address the N+1 problem in GraphQL APIs, DataLoader instances (or DataLoaders) are commonly used. An implementation of the DataLoader utility for Node.js is provided in the official GraphQL DataLoader repository [1]. Additionally, ports of this utility are available for other languages, such as java-dataloader [15] for Java. DataLoaders play a crucial role in optimizing data fetching by enabling the batching and caching of requests.

When using the DataLoader utility, resolvers use DataLoaders to bundle and process multiple database requests together within the same GraphQL query, instead of performing separate database queries. When multiple entities of the same type are requested, DataLoaders group these requests into a single SQL query per requested type. This approach results in the execution of only one SQL query for each requested type, which corresponds to a table in the example of films and actors. The use of batching in this approach can significantly reduce the number of individual SQL queries that are executed.

In addition to batching, the DataLoader utility provides caching capabilities. Once data is retrieved, it is stored in the cache for the context of the current query. When the same data is requested again, it can be efficiently served from the cache, eliminating the need for redundant database calls and improving query response times.

2.3.2 Generic approaches

The existing tools for translating GraphQL queries to SQL can be divided into two groups [16], servers with and libraries for GraphQL to SQL mapping.

2.3.2.1 GraphQL servers with generic GraphQL to SQL mapping

The first group consists of tools designed to generate a complete GraphQL API based on an existing SQL schema and some configuration. Examples for this group are Hasura [17] and PostGraphile [18]. These powerful engines act as a middle layer between the client and the database, not only generating the GraphQL schema, but also handling GraphQL queries, mutations, and subscriptions. Their aim is to speed up development by eliminating the effort of maintaining a GraphQL backend. Hasura acts as a standalone GraphQL server. PostGraphile can act as a GraphQL server or be used as a middleware to be integrated into an existing server.

2.3.2.2 Libraries for generic GraphQL to SQL mapping

The second group consists of tools that generate a SQL query from a GraphQL query. An example of one of those tools is Join Monster [19]. While the GraphQL servers with GraphQL to SQL mapping mentioned above include a similar tool, these libraries are different in that they do not act as a GraphQL server, but as a middle layer between an existing GraphQL server and a database. This allows these tools to provide more flexibility, as developers can design the GraphQL schema themselves (with some limitations or additional configuration).

Join Monster follows a code-only approach, which means that the schema, instead of being defined in its own file in SDL, it is defined in the code. In case of Join Monster this means that the schema is defined using graphql-js [20], a GraphQL implementation for JavaScript. This allows the addition of fields specific to Join Monster to the types of the schema, which can, for example, be strings for table names or functions that return a `String` for the SQL `JOIN` conditions given the required context. An example of the graphql-js definition for the type that represents the `film` table is provided in Listing 2.6. The example features the Join Monster specific fields required to resolve fields of the type in GraphQL queries.

2.4 Limitations of related work

While DataLoaders do solve the N+1 problem, they still lead to multiple queries and, because they work with resolvers, redundant code and maintenance effort. Additionally, DataLoaders are a more general solution and are not specifically tailored for SQL, thus not utilizing the full power of the Relational Database Management Systems (RDBMS).

GraphQL servers with GraphQL to SQL mapping, such as Hasura, generate the whole GraphQL schema, thus removing the ability to customize it almost entirely. These tools may reveal too much of the database to API users. Furthermore, their opinionated nature may clash with legacy projects that have specific demands, for example, in authentication/authorization.

Join Monster introduces information about the SQL schema directly into the GraphQL schema definition (see Listing 2.6). This means that SQL code is embedded within the GraphQL schema, making it dependent on the underlying SQL schema. However, this approach does not include validation checks to ensure that the SQL code is valid or compatible with the SQL schema. As a result, changes to the SQL schema could introduce errors in the GraphQL schema, which may go unnoticed until runtime. Furthermore, Join Monster is limited to Node.js applications and enforces a code-only approach, which may not align with projects that prioritize the schema-first approach using SDL documents to define the schema.

```
export const Film = new GraphQLObjectType({
  name: 'Film',
  extensions: {
    joinMonster: {
      sqlTable: 'film',
      uniqueKey: 'film_id',
    },
  },
  fields: () => ({
    title: {
      type: new GraphQLNonNull(GraphQLString),
      extensions: {
        joinMonster: {
          sqlColumn: 'title',
        },
      },
    },
    release_year: {
      type: GraphQLInt,
      extensions: {
        joinMonster: {
          sqlColumn: 'release_year',
        },
      },
    },
    actors: {
      type:
        new GraphQLNonNull(new GraphQLList(new GraphQLNonNull(Actor))),
      extensions: {
        joinMonster: {
          junction: {
            sqlTable: 'film_actor',
            sqlJoins: [
              (filmTable, junctionTable, args) =>
                `${filmTable}.film_id = ${junctionTable}.film_id`,
              (junctionTable, actorTable, args) =>
                `${junctionTable}.actor_id = ${actorTable}.actor_id`,
            ]
          },
        },
      },
    },
  }),
});
```

Listing 2.6: Example of an object type definition using Join Monster.

3 Methodology

This chapter provides an overview of the approach taken to address the objectives outlined in Section 1.3. After the basic concept is described, more specific issues and their proposed solutions are presented. Finally, the scope and limitations of the approach are described.

3.1 Basic concept

The GraphQL AST, which is derived from a query to validate it against a schema and execute it, is, as the name suggests, a tree. The tree structure of the GraphQL AST from Listing 2.5 is shown in Figure 3.1. A SQL query with subqueries can also be thought of as a tree structure. In this section, the process of transpiling the GraphQL AST into an equivalent SQL query are explored. In this context, transpiling describes the process of translating a GraphQL AST into an equivalent SQL query that retrieves the requested data from the underlying database.

To transpile a GraphQL AST into a nested SQL query in a comprehensible way, an internal tree representation is used. The AST is translated into the internal tree representation, and this in turn is translated into a SQL query. The AST depicted in Figure 3.1 reveals a pattern. All fields are either inner nodes, such as `allFilms` and `last_name`, or leaf nodes. Regarding the GraphQL schema, it can be seen that the inner nodes have object types, while the leaf nodes must have scalar types. It can also be deduced that, with respect to the underlying SQL database, the inner nodes correspond to tables, while the leaf nodes correspond to columns of the table. The internal tree representation is derived from this information. Figure 3.2 illustrates the internal tree obtained from the AST in Figure 3.1. `Document` and `OperationDefinition` are discarded and the fields are categorized into relations and attributes according to the described pattern.

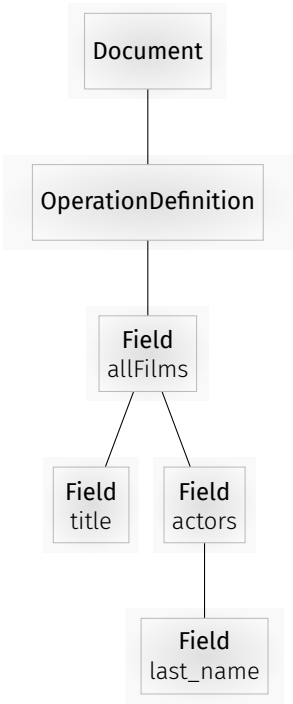

Figure 3.1: The GraphQL AST from Listing 2.5 represented as a tree.

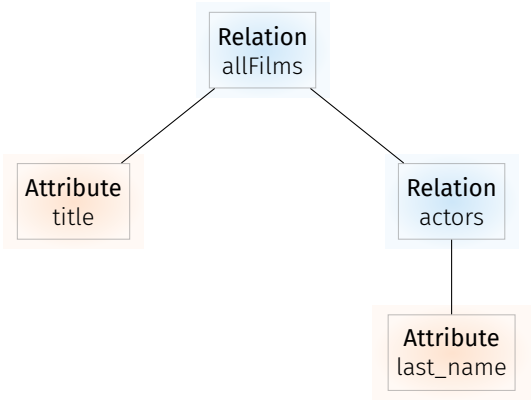

Figure 3.2: The internal tree representation derived from Figure 3.1.

As outlined in Section 1.3 the purpose of this thesis is to provide a solution to the N+1 problem that translates any GraphQL query into exactly one SQL query. With the described approach, each root field (meaning each field on the `Query` type that is requested in a query) leads to one internal tree, and thus to one SQL query. This is the same as for the generic approaches mentioned in Section 2.3.2. The `OperationDefinition` in the `AST` has one child node for each root field. When the `OperationDefinition` is removed, each child becomes a `Relation` and root node of a tree. Usually, but not necessarily, a GraphQL query includes only one root field. However, a GraphQL query can consist of multiple fields of the query type (such as in Listing 2.3), thus representing potentially very different requests. Since the relationship between these queries is unpredictable and may not be related at all, it is not possible to connect the resulting SQL queries in a generic way.

3.2 Challenges and proposed solutions

The GraphQL specification [9] defines how a GraphQL server should behave. It would be ideal to meet this specification. Additionally, common use cases for GraphQL APIs should be supported. This section looks at some features of the specification and common use cases in more detail.

3.2.1 Simple queries

Consider a query that requests all films and their titles (like Listing 1.1 without actors). This would result in the internal tree in Figure 3.2 without the nodes for `actor` and `last_name`. It is apparent that the SQL query obtained from this should be as in Listing 3.1.

```sql
SELECT film.title FROM film;
```

Listing 3.1: Simple SQL query requesting the titles of all films.

This internal tree uses the GraphQL field names as labels for the tree nodes. These must be used to construct a result in the expected JSON format. However, these names are usually not the names of the corresponding tables that are needed to construct the SQL query (like `allFilms` in Figure 3.2). There must be a way to derive table names from the corresponding GraphQL fields, either by a constraint on the GraphQL fields or by configuration. This approach employs the design pattern "Convention over Configuration", which aims to derive default settings from code instead of additional configuration [21]. Constraining the field names to match the names of SQL tables would be a significant restriction to the GraphQL schema, removing the ability of differently named fields in the GraphQL schema to refer to the same table (like `allFilms` and `filmById`). Since this restriction is not feasible, how can the name of the database table be derived? When constructing the internal tree, the GraphQL schema is introspected and the type of the requested `Relation` is retrieved. It is assumed that this type has the same name as its corresponding table.

In this example, this would be `film`, derived from the type `Film`, since it is assumed that table names are lowercase. For attributes, the name of the GraphQL field can be used. It is assumed that the field name matches the column name. Just like table names, column names are lowercase. In both cases, aliases can be handled because the GraphQL AST includes both the `name` and the `alias` (if there is one) for each requested field.

It would be possible to extend this behavior with some configuration like a lookup table linking type or field names to table or column names if having both match is not desired.

Since the end result should be a JSON structure, in the case of `allFilms` a JSON array, the JSON capability of the RDBMS can be utilized to directly accumulate the data in this form. The SQL query would then look as in Listing 3.2. Note that JSON methods may vary depending on the RDBMS used. `COALESCE` is used in combination with `JSON_ARRAY` to ensure that an empty JSON array is returned instead of `null` if there are no rows in the table.

```sql
SELECT
  COALESCE(
    JSON_ARRAYAGG(
      JSON_OBJECT(
        KEY 'title' VALUE film.title
      )
    ),
    JSON_ARRAY()
  )
FROM film;
```

Listing 3.2: The SQL query from Listing 3.1 but responding directly in JSON format.

3.2.2 Queries with increased depth

The GraphQL schema, as well as the SQL schema, should be minimally affected by the solution. Ideally, this would mean that any existing application that uses a GraphQL API and a SQL database could, with some configuration, use our mapping. Without any configuration, it is not possible for the mapping to work. This is because the relationships between tables in SQL cannot be derived from the GraphQL AST or the GraphQL schema.

Consider the internal tree in Figure 3.2. Deriving the SQL query from this tree introduces an additional challenge. The provided information does not make it evident how the tables `film` and `actor` are interconnected. To specify this, configuration is required. This configuration holds information on how to join tables for all relationships between tables that should be exposed.

Again, the JSON functionality of the RDBMS can be used to construct JSON arrays and objects reflecting the structure of the internal tree. For the tree in Figure 3.2, this is illustrated in Listing 3.3. Using this nested structure of SELECT statements can not only be recursively obtained from the internal tree but also prevents unintended Cartesian products when requesting deeper trees.

```sql
SELECT
  COALESCE(
    JSON_ARRAYAGG(
      JSON_OBJECT(
        KEY 'title' VALUE film.title,
        KEY 'actors' VALUE (
          SELECT
            COALESCE(
              JSON_ARRAYAGG(
                JSON_OBJECT(
                  KEY 'last_name' VALUE actor.last_name
                )
              ),
              JSON_ARRAY()
            )
          FROM actor, film_actor
          WHERE actor.actor_id = film_actor.actor_id
          AND film_actor.film_id = film.film_id
        )
      )
    ),
    JSON_ARRAY()
  )
FROM film;
```

Listing 3.3: SQL query using JSON functions to resolve internal tree in Figure 3.2.

3.2.3 Object and list types

GraphQL fields are defined within the GraphQL schema as either having an object or list type. Therefore, when a field is requested, the result is either a JSON object or a JSON array. If the field is nullable, the result may also be `null`. To determine whether an object or an array should be returned, the type information must be obtained by introspecting the GraphQL schema.

GraphQL fields, as mentioned in Section 2.1.2 can be nullable. For object types, this indicates that the value can be either an object or `null`. A `null` value would mean that the corresponding row in SQL was not found.

Lists, as well as objects in a list, can also be nullable (see Table 2.1). Nullable lists and lists of nullable objects cannot be represented in a relational database. Since a relational database is used as a data source, only non-nullable lists of non-nullable objects are considered for the proposed approach.

3.2.4 Filter arguments

For every field in a GraphQL schema, arguments can be defined. A common use case for arguments, which the proposed solution should support, is filtering. This could be used to retrieve one specific object like `filmById`. In this case, `null` is returned if no film with the provided `film_id` exists. However, it could also be used to filter multiple results as with `filmsByYear`. Given an integer, this would return zero or more films as an array. Nullable arguments, as mentioned in Section 2.1.2, should also be supported.

3.2.5 Pagination

Web applications often allow users to scroll through pages of content. This means that the client fetches one page after the other from the server. Cursor-based pagination makes this possible by providing both the option of retrieving only a limited number of results and the option of specifying a cursor after which this number is counted [22]. Therefore, the proposed solution should support cursor-based pagination as specified in the GraphQL Cursor Connections Specification [23].

Pagination is expressed in in GraphQL as a modification to the GraphQL schema. It is therefore an alternative to using list types directly. Fields that return multiple results can use list types, like `allFilms` or `actors` in Listing 2.4. A list indicates that the field returns multiple items of the wrapped type (`Film` or `Actor` in this case). However, it is generally recommended to paginate fields that return multiple types. Pagination is not mandatory in this solution. Both options can be utilized, as it can be advantageous to forgo pagination for specific fields, particularly if the results for that field are consistently small.

The GraphQL Cursor Connections Specification [23] establishes a uniform protocol for pagination within GraphQL. On the one hand, the specification specifies information about the connection, the page, and cursors that is accessible through the API. On the other hand, it defines the arguments `first` and `after`. The argument `first` accepts an integer specifying how many objects of the connection should be returned, while `after` accepts a cursor specifying after which object the first n elements should be selected.

A cursor is an opaque string that identifies each result entry, or edge. It can be used as an argument for the `after` argument in a field that implements cursor-based pagination to retrieve edges occurring after its own edge. By using the cursor value returned by the `endCursor` field, clients can obtain the next page and continue paging through content.

Listing 3.4 shows how the updated version of `allFilms` can be used to return the next ten nodes after a cursor and how additional information can be requested.

```
query {
  allFilms(first: 10, after: "bXlDdXJzb3I=") {
    edges {
      node {
        title
      }
      cursor
    }
    totalCount
    pageInfo {
      hasNextPage
      endCursor
    }
  }
}
```

Listing 3.4: GraphQL query featuring pagination.

For the GraphQL schema cursor-based pagination means that a field which returns multiple results, instead of having a list type, has a non-nullable object type, the connection type. This connection type has a field `edges` whose type is a list of the edge type. Additionally, the connection type includes the field `totalCount`, which indicates how many edges there are on the connection in total, as well as the field `pageInfo`, which provides the information if there is a next page and what the last cursor of the page is. Lastly, the edge type has a field `node` that has the original object type, as well as a field `cursor`. A cursor can be used for the `after` argument to fetch a page starting after that cursor. An example of the types mentioned with regard to the `allFilms` query can be seen in Listing 3.5 It should be noted that the original object type, in this case `Film`, remains unchanged when using cursor-based pagination.

Similarly to the forward pagination arguments `first` and `after`, the GraphQL Cursor Connections Specification [23] defines the backward pagination arguments `last` and `before`. These arguments enable data to be paginated in reverse. For the purpose of this thesis, the focus will be on forward pagination.

3.2.6 Sorting

Especially in combination with pagination, it can be useful to dictate the order of the requested elements. This can also be enabled by providing support for arguments that reflect the `ORDER BY` functionality of SQL. It should be possible to define what fields ordering should be allowed. It should also be possible to order by multiple fields, and order ascending as well as descending, since SQL offers this functionality.

```
type Query {
    # with cursor-based pagination:
    allFilms(first: Int, after: String): FilmConnection!
    # without:
    # allFilms: [Film!]!
}

type FilmConnection {
    edges: [FilmEdge!]!
    totalCount: Int!
    pageInfo: PageInfo!
}

type FilmEdge {
    node: Film!
    cursor: String!
}

type PageInfo {
    hasNextPage: Boolean!
    endCursor: String
}
```

Listing 3.5: Part of a GraphQL schema that defines `allFilms` using cursor-based pagination.

3.3 Scope and limitations

The approach, as outlined in this chapter, comes with some limitations. Some of these restrictions are a result of the effort to bridge the gap between dynamic GraphQL and structured SQL. Other restrictions inherent to the scope of the thesis.

3.3.1 Arguments

GraphQL arguments offer extensive flexibility, since their functionality depends on how they are implemented within the resolver function. Since a generic mapping cannot accommodate arbitrary functionalities, this solution focuses on common use cases for GraphQL arguments, as outlined by Porcello and Banks [12]. The prioritized functionalities are filtering and sorting based on fields of a table, as well as pagination. These selected functionalities serve as essential building blocks for GraphQL queries and exemplify how arguments can be supported, laying the groundwork for the inclusion of additional functionalities in the future.

3.3.2 Mutations and subscriptions

While GraphQL offers mutations for data manipulation and subscriptions for real-time updates, this solution focuses on generating one SQL query per GraphQL query. As a result, mutations and subscriptions are not supported. Although it is theoretically possible to extend the solution to include generic mappings for mutations and subscriptions, doing so would go beyond the scope of this thesis. Developers, in scenarios involving mutations, have the option of exploring alternative methods, like Command Query Responsibility Segregation (CQRS), as detailed by Martin Fowler [24]. CQRS describes the approach of decoupling read and write operations to improve performance and modularity.

3.3.3 Lists of lists and scalar values

GraphQL supports nested lists, for example `[[Actor]]`. If these lists describe relationships between types, such nested structures would be represented in relational databases as tables with relationships. Therefore, when using a SQL database as the data source, it is more appropriate to represent these nested structures in GraphQL as nested types that mirror the tables in the data source, rather than as nested lists. Since representing relationships as nested lists can be avoided and doing so leads to a more comprehensible GraphQL schema, they are not supported.

Similarly, GraphQL enables the definition of lists of scalar values, such as `[String!]!`. However, due to the lack of support for array types in some RDBMSs, it can be difficult to represent these types of lists in relational databases. While certain RDBMSs do offer support for arrays, effectively modeling lists of scalar values often involves alternative strategies, such as normalizing the data into a separate table and establishing a foreign-key relationship to the parent table. Due to the variability in SQL database support, because they can be represented in SQL using tables (and thus object types in GraphQL), and to prioritize the core implementation, this version does not support lists consisting of scalar values.

3.3.4 Not representable nullability

As explained in Section 3.2.3 nullable lists and lists of nullable types cannot occur and are therefore not supported.

Additionally, the bubble-up behavior described in the GraphQL Spec [9], which describes propagating `null` values up the GraphQL schema for non-nullable fields, are not supported. This is because this feature is mainly useful for other data sources. If a non-nullable field returns `null` in the generic GraphQL to SQL mapping, this means that the SQL and GraphQL schemas do not match. Propagating the `null` value up would not be useful.

3.4 Enhancing query flexibility with SQL views

Since the generic mapping approach maps GraphQL object types to SQL tables, it is not directly possible to have queries that fetch data that are not present in a table in the same form. Examples of this would be fields that represent a value that is computed from database columns. In other words, queries that involve some sort of business logic.

For this purpose, SQL views can be utilized. Since SQL views can be used for our approach in the same way as tables, they can be used to provide columns with values that are computed from columns of other tables. They can also be used to provide existing tables under a different name or with different column names. Providing a table and its columns under other names can be used to address the constraint of object type names matching table names and field names matching column names.

4 Implementation

This chapter explains how the approach described in Chapter 3 was implemented and utilized with example data. Currently, the implementation comprises a single repository that functions as a GraphQL server. This repository encapsulates both the logic described in Chapter 3 for handling GraphQL queries, as well as example data and server functionality to facilitate testing and demonstration purposes.

The complete implementation is available at `https://github.com/lukassailer/byos`. The project name BYOS stands for Bring Your Own Schema and expresses that no schema is generated, but one's own GraphQL and SQL schemas can be used. Version 0.1.0 is the current version at the time of this publication.

An outline of the selected technologies is presented first, followed by an explanation of the underlying database. Section 4.3 describes the structure of the repository, the details of which are described in Sections 4.4 and 4.5.

4.1 Overview of the chosen technologies

In order to accomplish the objectives of this study, it was necessary to make a choice of technologies to utilize. This section provides a summary of the main technologies that form the basis of the project.

4.1.1 jOOQ

The library jOOQ Object Oriented Querying (jOOQ) [2] provides a Domain-Specific Language (DSL) that allows SQL code to be written in Java. Using a DSL like jOOQ to build SQL queries offers an abstraction of SQL statements thus enabling modular assembly of valid statements. Furthermore, jOOQ features a code generator that creates Java classes for the underlying SQL database. This allows the correctness of SQL statements to be checked at compile time. These features make jOOQ ideal for dynamically constructing SQL queries. Furthermore, jOOQ supports various database systems, making it database-agnostic. This allows us to use functions such as `JSON_ARRAYAGG`, which may not be supported in the underlying RDBMS, but can be simulated by jOOQ in such cases.

The JSON capabilities of JOOQ are, in fact, so extensive that Lukas Eder, the creator of JOOQ, suggested that it could be used to implement a GraphQL to SQL translator [25], using a similar nesting of JSON aggregates as shown in Listing 3.3.

4.1.2 Kotlin

Since JOOQ is used for this project, Java would be a natural choice of programming language. However, Kotlin [26], with its full compatibility with the Java Virtual Machine (JVM), is an equally viable option.

In addition to its seamless integration with Java code and libraries, Kotlin offers unique features and advantages. One of the distinctive features of Kotlin is its support for functional programming paradigms. Kotlin also has robust null safety features that help prevent `NullpointerException`s.

For this project, Kotlin was chosen not just because of its technical merits but also as a personal preference of the author.

4.1.3 Jackson

Jackson [27] is a JSON library for Java (and thus the JVM platform). It provides tools for parsing and generating JSON. For our implementation, Jackson's `ObjectMapper` is used to parse the GraphQL AST's JSON structure.

4.1.4 GraphQL Java

GraphQL Java [28] is an implementation of the GraphQL specification. It includes tools for parsing, validating, and executing schemas using the resolver approach. This implementation only uses a small portion of these tools, since most of this is handled differently. However, the query validation and the type system of GraphQL Java are utilized.

4.1.5 Spring Boot

The Spring Boot framework [29] is a simple tool used to create Java applications. In this project, Spring Boot was used as the foundation for the GraphQL Server.

The project was initialized with Spring Initializr [30], a tool that generates a Spring Boot project. The Spring Initializr was used to construct a Spring Boot GraphQL server.

The server also offers a GraphiQL frontend, a web-based graphical user interface that facilitates interaction with GraphQL APIs. GraphiQL enables users to query the available GraphQL schema without using third-party tools or manual Hypertext Transfer Protocol (HTTP) requests. However, alternative methods for accessing the GraphQL API are also possible.

4.2 Database with sample data

PostgreSQL [31] was chosen as the RDBMS to store sample data for the application. However, an alternative RDBMS could also have been used, since JOOQ supports the utilized JOOQ functions in various RDBMS platforms.

For test data, the Sakila example database was used as specified by the JOOQ organization [32]. This database represents a video rental store and consists of movies, actors, and other related information, as used throughout this thesis. The schema and data for film categories have been expanded to include parent categories to allow categories to have a parent category as well as be the parent category for other categories. This change was made to add an example where a table has a foreign-key relationship with itself.

4.3 Project Structure

The BYOS repository is divided into two packages: the byos package (core package) for logic components that are independent of the application in which they are used, and the example package for application-specific elements. The latter is an example of a GraphQL Server, that makes use of the mapping logic of the core package. This section provides an overview of these packages. Section 4.4 describes the components of the example package, while Section 4.5 details those of the core package. The contents of Sections 4.4 and 4.5 is presented in the same order in which the code is executed.

4.3.1 Core package

The core package consists of the following Kotlin files:

- `Counter`
- `JooqHelpers`
- `QueryTranspiler`
- `SchemaParsing`
- `WhereCondition`

The heart of the core package and thus the query transpilation logic is the `QueryTranspiler` class. The file containing the `QueryTranspiler` class also contains the definition for the internal representation tree for queries (`InternalQueryNode`). The `QueryTranspiler` class itself contains functions for generating internal query trees from operation definitions and resolving internal query trees into JOOQ `Fields`.

To create an instance of the `QueryTranspiler` class, an object of the `WhereCondition` class is required. The `WhereCondition` class is also part of the core package. It includes functions to obtain conditions for `WHERE` clauses from relationships and arguments.

Furthermore, the package includes the `SchemaParsing` file that provides the `getFieldTypeInfo` function that inspects a GraphQL schema and obtains type information for a GraphQL field.

The `JooqHelpers` file consists of helper functions for executing and formatting JOOQ queries as well as a pretty printer for debugging.

Lastly, the `Counter` class offers a simple counter, an instance of which is also used to instantiate an `QueryTranspiler` object.

4.3.2 Example package

The example package includes these Kotlin files:

- `ByosApplication`
- `Config`
- `GraphQLRequestFilter`
- `GraphQLService`

The example package defines a spring boot server that offers a GraphQL API. The entry point for this application is the `main` method in the `ByosApplication` class. By running this method the server can be started.

Incoming GraphQL Queries are intercepted in order to resolve them using the logic of the core package. To intercept GraphQL Queries the class `GraphQLRequestFilter` overrides the function `doFilterInternal` of the `OncePerRequestFilter` class, which is provided by the Spring framework.

Another class of the example package, namely `GraphQLService`, defines functions that extract the relevant GraphQL operation definition from the HTTP request and execute it using an instance of the `QueryTranspiler` class.

The `GraphQLService` provides a function to the `WhereCondition` instance that determines how the relationships should be resolved. Since this is application-specific, this function, `getConditionForRelationship`, is located in the `Config` file in the example package.

4.4 Exemplary application

This section provides an overview of the exemplary GraphQL Server within the example package, which demonstrates the usage of the GraphQL to SQL mapping. The implementation of the logic for the mapping itself is described in Section 4.5.

4.4.1 Intercepting the request

The `GraphQLRequestFilter` serves as an entry point for processing GraphQL queries. It intercepts any incoming HTTP request and, if it is a GraphQL query, calls the `executeGraphQLQuery` method of the `GraphQLService` to execute it. The result of this method is then sent as a response to the client.

4.4.2 Handling the request

The `executeGraphQLQuery` method of the `GraphQLService` uses the `Validator` of GraphQL Java to validate the GraphQL `Document` against the GraphQL schema and return errors if they occur. If no errors occur, the GraphQL `OperationDefinition` for the selected operation is extracted from the AST. For this `OperationDefinition` the `buildInternalQueryTrees` method is invoked on an instance of the `QueryTranspiler` class. To instantiate the `QueryTranspiler` class both a GraphQL schema and a `WhereCondition` are provided. The latter is instantiated itself using the `getConditionForRelationship` function defined in the `Config` file. Afterwards, every resulting query tree is resolved and the respective query executed. Finally the JSON results are joined into a `data` object and returned.

4.5 Query resolution logic

In contrast to the application-specific context discussed in the previous section, this section explains the implementation of the GraphQL to SQL mapping, which is mostly independent of the application's specific use case.

4.5.1 Internal query tree

The internal tree representation described in Section 3.1 was implemented in Kotlin as the sealed class `InternalQueryNode`. The implementation of the class is depicted in Listing 4.1. A sealed class is an abstract class with a fixed set of subclasses. The subclasses of `InternalQueryNode` are `Relation` and `Attribute`. This means that each node can be either a `Relation`, an inner node with a list of children, or an `Attribute`, and thus a leaf node.

Every `InternalQueryNode` has a `graphQLFieldName` and a `graphQLAlias`. The `graphQLFieldName` is the name of the requested GraphQL field. For an `Attribute`, this matches a column name in the database, whereas for a `Relation`, it identifies a predefined relationship between tables. The `graphQLAlias` holds the alias for the GraphQL field, so that the field can be returned under that alias. Objects of the `Attribute` class do not need any additional information.

Objects of the `Relation` class also have a `sqlAlias`, an alias to be used in the SQL query. This unique alias is used to prevent conflicts if the same table occurs multiple times in the generated SQL query.

A `Relation` also contains information about the GraphQL type inside the `fieldTypeInfo` field. This field of type `FieldTypeInfo` consists of the `graphQLTypeName`, the Boolean value `isList`, and the `relationName`. The `graphQLTypeName` holds the name of the GraphQL object type of the field. If `isList` is true, it means that this object type is wrapped in a list, and therefore an JSON array should be returned, otherwise an object is returned. The `relationName` is simply shorthand for the lowercase value of the `graphQLTypeName`, which matches the name of the corresponding table.

In addition, a `Relation` stores the list arguments that were provided for its associated field. An `Argument` is a key value pair, where the key is a `String` and the value implements the GraphQL Java interface `Value`. Therefore, it represents a GraphQL argument consisting of a name and any GraphQL value.

Lastly, if the GraphQL type for a `Relation` uses a connection type, it has a `connectionInfo` (otherwise this is `null`). This field of type `ConnectionInfo` contains aliases for the fields `edges`, `node`, `cursor`, and `totalCount`. Additionally, it contains a list of `PageInfo` objects, each of which contains their alias and a list of aliases for the fields `hasNextPage` and `endCursor`. The aliases are lists because the same field can be requested multiple times under different aliases.

4.5.2 Building the internal query tree

The class `QueryTranspiler` defines functions for building and resolving internal query trees. An instance of the `QueryTranspiler` class must be initialized with an instance of the `WhereCondition` class (which will be explained in Section 4.5.5) and a GraphQL schema. The instance of the `WhereCondition` class is specific to the exemplary application.

The method `buildInternalQueryTrees` takes the `OperationDefinition` of a GraphQL AST and returns a list of `Relation` objects or, in other words, a forest. Internally `buildInternalQueryTrees` calls `getChildrenFromSelectionSet` on the `SelectionSet` of the `OperationDefinition` and casts the list of `InternalQueryNode`s into a list of `Relation`s. This cast is valid because a root node cannot be an `Attribute`, since an `Attribute` has to exist on a table.

```
sealed class InternalQueryNode(
  val graphQLFieldName: String,
  val graphQLAlias: String
) {
  class Relation(
    graphQLFieldName: String,
    graphQLAlias: String,
    val sqlAlias: String,
    val fieldTypeInfo: FieldTypeInfo,
    val children: List<InternalQueryNode>,
    val arguments: List<Argument>,
    val connectionInfo: ConnectionInfo?,
  ) : InternalQueryNode(graphQLFieldName, graphQLAlias)

  class Attribute(graphQLFieldName: String, graphQLAlias: String)
    : InternalQueryNode(graphQLFieldName, graphQLAlias)
}

data class FieldTypeInfo(
  val graphQLTypeName: String,
  val isList: Boolean
) {
  val relationName = graphQLTypeName.lowercase()
}

data class ConnectionInfo(
  val edgesGraphQLAlias: String,
  val nodeGraphQLAlias: String,
  val cursorGraphQLAliases: List<String>,
  val totalCountGraphQLAliases: List<String>,
  val pageInfos: List<PageInfo>
)

data class PageInfo(
  val graphQLAlias: String,
  val hasNextPageGraphQlAliases: List<String>,
  val endCursorGraphQlAliases: List<String>,
)
```

Listing 4.1: Implementation of the internal query tree.

The implementation of `getChildrenFromSelectionSet` can be seen in Listing 4.2. The initial call of `getChildrenFromSelectionSet` on the `SelectionSet` of the `OperationDefinition` constructs one `Relation` object for each root field in the GraphQL query. A field in the `SelectionSet` is a selection. For each selection, since these are object types representing tables, there are children. These children are located in a nested `SelectionSet` for which a recursive call of `getChildrenFromSelectionSet` is issued. The return type of `getChildrenFromSelectionSet` is a list of `InternalQueryNodes`, which means that its result can be a mixture of both `Relation` and `Attribute` objects. When there are no more nested `Relations`, only `Attributes` are returned, and the recursion stops. As long as nested `Relations` exist, the recursion continues until the forest is complete.

As seen in Listing 4.2, the type of the `InternalQueryNode` constructed for a selection depends on its `SelectionSet`, the `subSelectionSet`. If there is no `subSelectionSet`, an `Attribute` is created. Otherwise, a `Relation` is created. For the `graphQLFieldName` the `name` of the selection is used, since this holds the name of the requested field. Similarly, the `alias` of the selection is assigned to the `graphQLAlias`. If no alias is provided in the GraphQL query, the `graphQLAlias` is the same as the field name. The `sqlAlias` is derived from the `graphQLFieldName` and an increasing ID, making it unique while still being human-readable.

As mentioned above, the `children` of a `Relation` are created by calling `getChildrenFromSelectionSet` on the `subSelectionSet`. For `Relations` of a connection type, the `SelectionSet` of the node type is used instead. The `fieldTypeInfo` is retrieved by the `getFieldTypeInfo` function of the `SchemaParsing` module, since it requires introspection of the GraphQL schema. For `Relations` of a connection type, this function is called again on the `edge` and the `node` type, thus producing the information of the `node` type. Similarly, the `SelectionSets` of these nested types are followed to obtain the aliases of the special fields used for fields with a connection type. These aliases are lists, since the fields can be requested multiple times under different aliases or not at all resulting in an empty list.

4.5.3 Introspecting the schema

The function `getFieldTypeInfo` is called to get the `FieldTypeInfo` for a field of the GraphQL schema. To do this, a GraphQL schema, a field name, and a type name are provided. With these, it is possible to lookup the type of the field on the provided type inside the GraphQL schema. The result of this lookup is a `GraphQLType`. To turn this `GraphQLType` into a `FieldTypeInfo` the name of the object type and the `isList` Boolean have to be derived from it. This is the task of the `getTypeInfo` function. The `GraphQLType` can be a `GraphQLObjectType`, in which case its type name can be accessed and `isList` would be `false`. But the `GraphQLType` could also be a `GraphQLNonNull` or `GraphQLList`, both of which wrap an inner type. In these cases `getTypeInfo` is called recursively. If, while descending the wrapped types, a `GraphQLList` is passed `isList` is `true`.

```
private fun getChildrenFromSelectionSet(
  selectionSet: SelectionSet,
  counter: Counter,
  parentGraphQlTypeName: String = schema.queryType.name
): List<InternalQueryNode> =
  selectionSet.selections
    .filterIsInstance<Field>()
    .map { selection ->
      val subSelectionSet = selection.selectionSet
      when {
        subSelectionSet == null -> {
          InternalQueryNode.Attribute(
            graphQLFieldName = selection.name,
            graphQLAlias = selection.alias ?: selection.name
          )
        }

        subSelectionSet.selections.any {
          it is Field && it.name == "edges"
        } -> {
          // construct InternalQueryNode.Relation with connectionInfo...
        }

        else -> {
          val fieldTypeInfo = getFieldTypeInfo(
            schema,
            selection.name,
            parentGraphQlTypeName
          )
          InternalQueryNode.Relation(
            graphQLFieldName = selection.name,
            graphQLAlias = selection.alias ?: selection.name,
            sqlAlias =
              "${selection.name}-${counter.getIncrementingNumber()}",
            fieldTypeInfo = fieldTypeInfo,
            children = getChildrenFromSelectionSet(
              subSelectionSet,
              fieldTypeInfo.graphQLTypeName
            ),
            arguments = selection.arguments,
            connectionInfo = null
          )
        }
      }
    }
```

Listing 4.2: Shortened version of the method `getChildrenFromSelectionSet` of the `QueryTranspiler` class. The method transforms an AST into the internal tree representation.

4.5.4 Resolving the internal tree

The purpose of the `resolveInternalQueryTree` method, defined in the `QueryTranspiler` class, is to consume the internal tree representation, built using the logic of the preceding sections, and return a JOOQ statement. Specifically, it returns a JOOQ `Field`, which represents a SQL column. The `Field` returned by `resolveInternalQueryTree` is a `Field` of type `JSON`, indicating it encapsulates a single column containing a JSON structure. A `Field` can be used in the `select` function of JOOQ, which is analogous to a SQL `SELECT` statement. This return type was chosen to allow the function to call itself recursively for all children of type `Relation`.

Within the `resolveInternalQueryTree` method, the table with the `relationName` of the provided `Relation` is retrieved from the tables that were generated by the JOOQ code generator and assigned a unique alias. This aliased table is assigned to the `outerTable` variable.

After that, the children of the `Relation` are processed. The list of children of type `Attribute` is converted into a list of columns of the `outerTable` with their respective `graphQLFieldName` in lowercase as the field name. The list is called `attributeNames`. For each child of type `Relation` the `resolveInternalQueryTree` method is invoked again. The results form the list `subSelects`, a list of `JSON Fields`. For each recursive call, the `getForRelationship` method of the `WhereCondition` instance is used to generate a condition that is used to join each inner table to the outer table. Furthermore, the arguments `first`, `orderBy` and `after` are extracted, as well as the sorting criteria and filter arguments. A cursor is constructed from the fields of the order criteria. The condition realizing the offset and the conditions for the filter arguments are generated using the methods of `WhereCondition`.

Finally, the prepared data is used to construct a Common Table Expression (CTE) which provides the required data to construct the final JSON result. This CTE is depicted in Listing 4.3.

The argument `first` specifies how many results should be fetched. If none is provided, all results are fetched. To facilitate this, an SQL `LIMIT` with the provided value is applied, if one is provided.

If the field `hasNextPage` is requested, the CTE includes a column `count_after_cursor`. The `hasNextPage` field on the `PageInfo` object type indicates if a next page exists. One solution to obtain this information would be to always query for one more result from the database than requested via the `first` argument. In this way, it would be possible to infer that a next page only exists if this additional result has been obtained. Instead, a more elegant solution has been chosen. If the `hasNextPage` field is requested, the number of results after the `after` cursor is calculated. As seen in Listing 4.3 this column, `count_after_cursor`, is obtained by using a SQL window function. The value of `hasNextPage` is true if `count_after_cursor` is greater than the value of the argument `first`. The SQL function `MAX` must be used on `count_after_cursor`, because it exists, with the same value, in every row. In other words, it is true if not all remaining results fit on the current page.

```
fun resolveInternalQueryTree(
  relation: InternalQueryNode.Relation,
  joinCondition: Condition = DSL.noCondition()
): org.jooq.Field<JSON> {
// ...
  val cte =
    DSL.name("cte").`as`(
      DSL.select(attributeNames)
        .select(subSelects)
        .apply {
          if (cursorRequested || endCursorRequested) select(cursor)
        }
        .apply {
          if (hasNextPageRequested)
            select(DSL.count().over().`as`("count_after_cursor"))
        }
        .from(outerTable)
        .where(argumentConditions)
        .and(joinCondition)
        .and(afterCondition)
        .orderBy(orderByFieldsWithDirection)
        .apply { if (limitValue != null) limit(limitValue) }
    )
// ...
```

Listing 4.3: CTE used to prepare the data required for the current level of the GraphQL resonse.

If the `cursor` for an edge object is requested, it is also added to the CTE. This `cursor` column contains a JSON object, whose keys are all fields by which the results of the CTE are ordered by and their respective values. In a real-world application, this should be an opaque cursor (for example, Base64-encoded), as specified in the GraphQL Cursor Connections Specification [23]. But for debugging and demonstration purposes, it is not encoded for this application.

The fields for sorting the results are provided in the input object `orderBy`. In addition to these fields, the results are secondarily ordered by the fields of the primary key, if they are not already part of the provided fields. If no `orderBy` input object is provided, the results are ordered by the fields of the primary key. This ensures a deterministic order of results.

In addition to the CTE from Listing 4.3 subqueries to resolve the fields `totalCount` and `endCursor` are constructed. These subqueries are depicted in Listing 4.4. The `totalCountSubquery` uses the `selectCount` function of JOOQ, which constructs a `SELECT COUNT(*)` statement in SQL. The subquery is constructed in such a way that the count is the number of rows the CTE would have if it did not have an offset. The `endCursorSubquery` uses a SQL window function to obtain the cursor value of the last row of the CTE.

```
// ...
  val totalCountSubquery =
    DSL.selectCount()
      .from(outerTable)
      .where(argumentConditions)
      .and(joinCondition)

  val endCursorSubquery =
    DSL.select(DSL.lastValue(DSL.field(cursor.name)).over())
      .from(cte)
      .limit(1)
// ...
```

Listing 4.4: Subqueries used for `totalCount` and `endCursor`.

The CTE and, if necessary, the subqueries are used to construct the JOOQ `Field`. If the `Relation` is an object, the data is bundled into a JSON object. If it is a list, it is bundled into an array of objects. Since the `jsonObject` function of JOOQ accepts a variable number of arguments, the `attributeNames` and `subSelects` of the CTE are provided using the Kotlin spread operator (i.e., asterisk). Since these collections are lists and the spread operator only works on typed arrays, they need to be converted using `toTypedArray`. If the GraphQL field in the schema uses a connection type, the connection structure is created, and the special pagination field values are provided, using the subqueries for `totalCount` and `endCursor`, if requested. The return statement of `resolveInternalQueryTree`, and thus the construction of the resulting JSON structure, can be seen in Listing 4.5.

```
// ...
  return DSL.field(
    DSL.with(cte).select(
      when {
        relation.connectionInfo != null -> {
          // construct connection structure...
        }

        relation.fieldTypeInfo.isList -> {
          DSL.coalesce(
            DSL.jsonArrayAgg(
              DSL.jsonObject(
                *attributeNames.toTypedArray(),
                *subSelects.toTypedArray()
              )
            ),
            DSL.jsonArray()
          )
        }

        else -> {
          DSL.jsonObject(
            *attributeNames.toTypedArray(),
            *subSelects.toTypedArray()
          )
        }
      }
    ).from(cte)
  ).`as`(relation.graphQLAlias)
}
```

Listing 4.5: Construction of the jOOQ field. The data from the CTE is put into JSON format.

4.5.5 Constructing conditions

To construct the SQL query, different types of conditions are needed for the `WHERE` clause. These conditions serve three purposes: to facilitate table joins, to allow data filtering based on the arguments provided, and to support the `after` argument for pagination. The class `WhereCondition` defines methods to facilitate these use cases. To initialize an instance of the `WhereCondition` class, a function, that defines how conditions can be obtained given a relationship must be provided.

For joining a parent table with a child table, as they appear in the internal tree structure, a condition is created based on the specified relationship. Since this is specific to the application context, `getForRelationship` uses the function `getConditionForRelationship` that was passed when initializing the instance of the `WhereCondition` class (see Listing 4.6). The relationship name matches the field name. Both tables have to be of the correct type, which is generated by JOOQ. Using the type of tables allows the tables to use aliases. The relationship name is used together with the table types to allow multiple relationships to have the same name and multiple relationships to exist between the same tables. Matching the types of tables also ensures that the correctness of the defined condition can be checked at compile time. The custom definition of condition also allows relationships to spread over multiple tables, for example, the relationship between films and actors with the junction table `film_actor`. It would also be possible to have a relationship from one table only to some entries of the other table by filtering them inside the condition.

```
class WhereCondition(
  private val getConditionForRelationship:
    (String, Table<*>, Table<*>) -> Condition?
) {
  fun getForRelationship(
    relationshipName: String,
    left: Table<*>,
    right: Table<*>
  ): Condition =
    getConditionForRelationship(relationshipName, left, right)
      ?: error(
        "No relationship called $relationshipName" +
        "found for tables $left and $right"
      )
//...
```

Listing 4.6: Method that returns a condition that joins two tables.

Listing 4.7 shows the function passed when constructing a `WhereCondition` for the exemplary application. This defines how conditions can be obtained from relationships between tables in the Sakila example database. The first relationship shown in Listing 4.7 describes the field `actors` on the GraphQL type `Film`. Since the field occurs on the `Film` and has type `Actor`, the condition connects the tables `film` and `actor`. Since this is a many-to-many relationship, a subquery with the table `film_actor` is used to construct the condition. The second relationship describes the relationship the table `category` has with itself. This condition is a simple equality condition that joins the table `category` with itself where the `parent_category_id` matches the `category_id`. A condition like this would not be possible without using JOOQ's table types and thus allowing SQL aliases, since the instances of `category` need to have different aliases.

```
fun getConditionForRelationship(
  relationshipName: String, left: Table<*>, right: Table<*>
): Condition? =
  when {
    relationshipName == "actors" && left is Film && right is Actor ->
      DSL.exists(
        DSL.selectOne().from(FILM_ACTOR)
          .where(left.FILM_ID.eq(FILM_ACTOR.FILM_ID)
          .and(FILM_ACTOR.ACTOR_ID.eq(right.ACTOR_ID)))
      )
    relationshipName == "parent_category" && left is Category
      && right is Category ->
    left.PARENT_CATEGORY_ID.eq(right.CATEGORY_ID)
    //...
    else -> null
  }
```

Listing 4.7: Application-specific lookup function that defines conditions. This is used as an argument when initializing `WhereCondition`.

Moreover, the `WhereCondition` object handles data filtering using the `getForArgument` method. It dynamically interprets the argument provided which matches a field on the table, generating a condition using the `EQ` operator for its value. This also works for `null` values, generating the `IS NULL` operator, and for lists, generating the `IN` operator.

Lastly, in the context of pagination, the `WhereCondition` object constructs conditions that enable after-based pagination. This is done by the `getForAfterArgument` method. By leveraging the current sorting and after argument, it creates a condition that positions the query to retrieve data following a given point. This condition is an example of the seek method as described by Markus Winand [33]. To construct this condition, the key-value pairs of the JSON cursor provided as the `after` argument are processed in order. The cursor represents the exact position in the result set, after which the next page of data should begin. The `getForAfterArgument` method intelligently translates these cursor values into database conditions that implement the seek behavior.

The method iterates through each field-value pair in the cursor and, for each field, identifies the corresponding field used as an order criterion. Depending on whether the order is ascending or descending, it creates a condition that positions the query's result set either before or after the provided value. This process is recursively applied for each key-value pair in the cursor, allowing the condition to accurately seek data according to the desired pagination point.

Consider an example of sorting films in ascending order by release year and descending order by title. Furthermore, the page should start after a provided cursor. The cursor value provided for the `after` argument is `"{"release_year":2006, "title":"ZORRO ARK", "film_id":1000}"` and the value for the `orderBy` argument is `{release_year:ASC, title: DESC}`. Providing these values leads to the condition in Listing 4.8 being generated.

```
allFilms.release_year > 2006
OR (
  allFilms.release_year = 2006
  AND (
    allFilms.title < 'ZORRO ARK'
    OR (
      allFilms.title = 'ZORRO ARK'
      AND allFilms.film_id > 1000
    )
  )
)
```

Listing 4.8: Example for a generated seek condition.

5 Discussion

In this chapter, the performance of a SQL query generated by the implementation of Chapter 4 is compared with the queries that would be issued when using basic resolvers. This is followed by a comprehensive discussion of the limitations, constraints, and future possibilities of the GraphQL-to-SQL mapping approach developed in the course of this thesis. Exploring its limitations and potential for improvement provides a deeper understanding of the broader context in which this solution operates.

5.1 Improvement over naive approach

This section compares the performance of a SQL query generated by BYOS with the queries that would be executed by resolvers in the naive approach. This comparison serves as a demonstration of the performance gains achieved by addressing the N+1 problem and using a single SQL query. It is important to note that the purpose of this section is not an exhaustive comparison between approaches but rather a showcase of the potential optimization that can be achieved. Performance depends on many factors, such as the chosen GraphQL query, the underlying data, and the RDBMS. The example data used for this performance test consists of 1000 films and 200 actors.

Note that these times only describe the time consumed by the SQL query or queries. It does not include the time to obtain them from the GraphQL query. The result of the generated SQL query is already in JSON format, whereas the results of the queries in the naive approach would still need to be processed into a JSON result. It is anticipated that converting the data acquired through the naive approach into a JSON structure (and sorting it if, as in the example, the SQL query has not already applied an order) would take a significant amount of time.

The performance test is performed on the query from Listing 1.1, which requests all films with their title and the last names of their actors. The generated SQL query is shown in Listing 5.1. The query begins with a `WITH` clause creating the CTE as described in Section 4.5.4, which retrieves the titles of all films and defines a correlated subquery for the `actors` field. This nested query also defines a CTE, which obtains the actors associated with each film. The inner CTE is used to construct an array of objects containing the last names. Finally, the outer CTE is used to construct an array of objects containing the film titles and the actor arrays.

```sql
SELECT (
  WITH
    "cte" AS (
      SELECT
        "allFilms-0"."title" AS "title",
        (
          WITH
            "cte" AS (
              SELECT "actors-1"."last_name" AS "last_name"
              FROM "public"."actor" AS "actors-1"
              WHERE exists (
                SELECT 1 AS "one"
                FROM "public"."film_actor"
                WHERE (
                  "allFilms-0"."film_id" =
                    "public"."film_actor"."film_id"
                  AND "public"."film_actor"."actor_id" =
                    "actors-1"."actor_id"
                )
              )
              ORDER BY "actors-1"."actor_id" ASC
            )
          SELECT COALESCE(
            JSON_AGG(JSON_BUILD_OBJECT('last_name', "last_name")),
            JSON_BUILD_ARRAY()
          )
          FROM "cte"
        ) AS "actors"
      FROM "public"."film" AS "allFilms-0"
      ORDER BY "allFilms-0"."film_id" ASC
    )
  SELECT COALESCE(
    JSON_AGG(JSON_BUILD_OBJECT(
      'title', "title",
      'actors', "actors"
    )),
    JSON_BUILD_ARRAY()
  )
  FROM "cte"
) AS "allFilms";
```

Listing 5.1: Generated PostgreSQL query.

The performance of the SQL query generated from the GraphQL query in Listing 5.1 was evaluated, and its execution plan was retrieved using `EXPLAIN ANALYZE`. A graph depicting the execution plan is shown in Figure 5.1. It is important to note that this plan is generated by PostgreSQL and other RDBMSs may create a more efficient execution plan. The plan shows that the foreign-key join between the `actor` table and the `film_actor` table is executed via a Hash Join. The result-set of the Hash Join is sorted by its `actor_id` and then aggregated. This aggregate is then combined with the `film` table to produce the final result. The planning time was 3 ms and the execution time was 46.5 ms. Performance did not show any noticeable difference when using a field with a connection type instead (i.e., a type with an `edges` field).

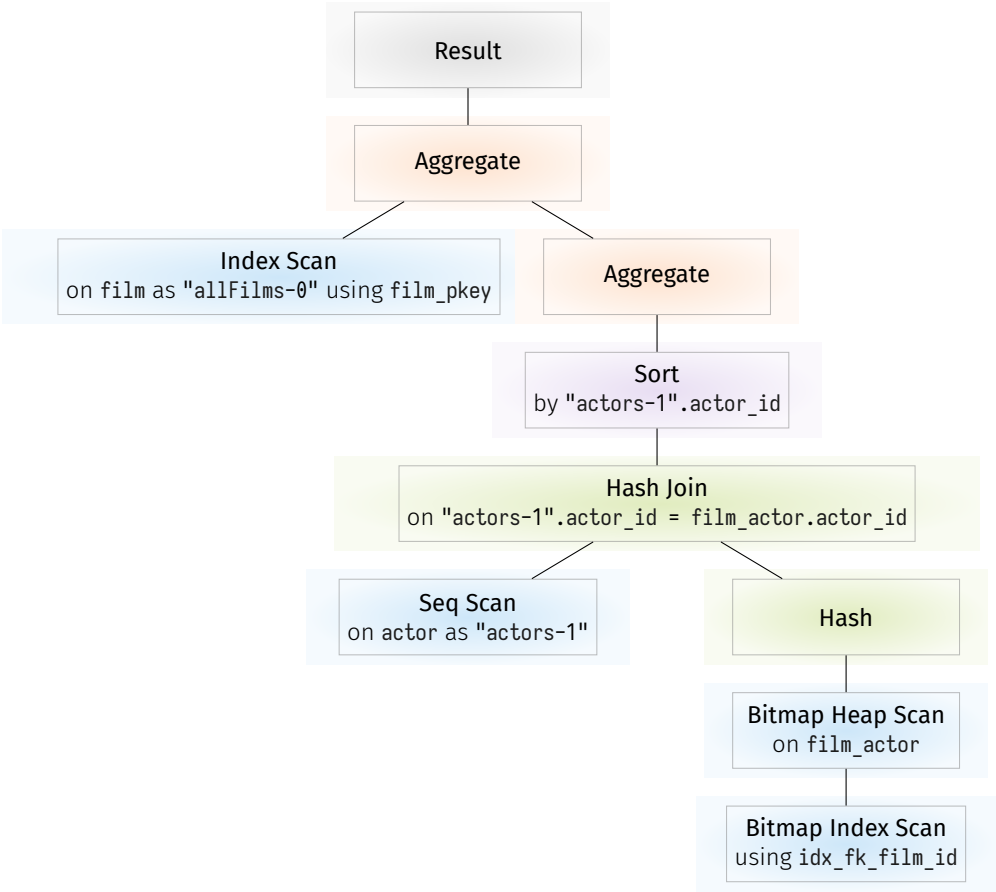

Figure 5.1: Execution plan for the generated SQL query from Listing 5.1.

The naive approach would be to request all films in one query (Listing 5.2) and after that request the respective films for each film (Listing 5.3).

The first query, shown in Listing 5.2, results in a Sequential Scan, since the entire table is selected. The measured planning time was 1.4 ms and the execution time was 0.7 ms.

```sql
SELECT film.title, film.film_id
FROM film;
```

Listing 5.2: SQL query to request all films.

The second query (shown in Listing 5.3) is executed for every film, 1000 times in total, resulting in the execution plan depicted in Figure 5.2 for each execution. Since it implements the same foreign-key join as the generated query's execution plan (Figure 5.1), it is a subtree of it. This query required 1.8 ms of planning time and 0.4 ms of execution time during its initial execution. Further executions reduced these times to 0.6 and 0.2 ms.

```sql
SELECT actor.last_name
FROM film_actor
JOIN actor ON film_actor.actor_id = actor.actor_id
WHERE film_actor.film_id = <film_id>;
```

Listing 5.3: SQL query to request all actors for a film given its `film_id`

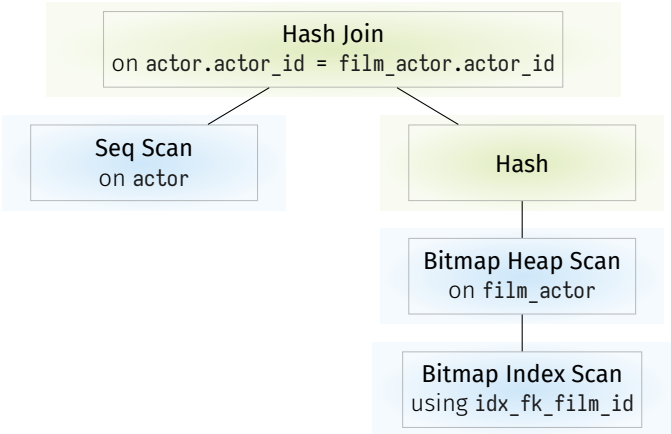

Figure 5.2: Execution plan for SQL query from Listing 5.3.

To conclude, the generated query took 49.5 ms. The naive approach resulted in 1001 queries. These queries would require 803.5 ms in total, assuming that they are issued in sequence.

This example demonstrates the potential performance enhancements of the approach employed by BYOS. It could be argued that a well-designed API, for most use cases, would not require or allow the retrieval of thousands of data entries, even when resolving deeper GraphQL queries. However, subsequent tests have shown that even if only the first ten films (ordered by their `film_id`) are requested, noticeable improvements can be achieved. The query generated by BYOS required 3.7 ms (and already serves JSON in the structure using **edges** and **node**), whereas the queries created by the naive approach required a total of 9.7 ms.

5.2 Limitations and constraints

This section delves into the inherent challenges that emerge when adopting a generic approach to mapping GraphQL queries to SQL. These challenges result from the combination of the generic strategy with extensive GraphQL schemas and the dynamic nature of user-defined queries. Exploring these challenges allows for a deeper understanding of the intricate interplay between the adaptability of GraphQL and the structured domain of SQL, as well as the trade-offs that arise when combining the two domains.

5.2.1 Schema constraints

Since the GraphQL schema, as well as the SQL schema, should be controlled by the user, instead of generating the GraphQL schema, one of them imposes some restrictions on the other. The direction in which this dependency acts depends on which is designed first.

The GraphQL object type names must match the SQL table names. In the same way, GraphQL field names with scalar types must match SQL column names.

If these restrictions are not feasible, a simple solution for the current implementation would be to use SQL views that use the required names as mentioned in Section 3.4.

5.2.2 Configuration

To allow nested GraphQL queries, which are crucial for a GraphQL API, the server requires information on how to join tables. This information about relationships between tables must be configured by hand.

The configuration using JOOQ is a unique feature and an advantage of the provided implementation. It provides both type safety and guarantees that the SQL code written is compliant with the SQL schema, both at compile time.

However, setting up relationships with JOOQ code requires that the person writing the code has a general understanding of JOOQ and SQL.

5.2.3 Business logic

A generic approach naturally represents Create, Read, Update and Delete (CRUD) operations on the database. The approach in this thesis is limited to queries and therefore read operations, but a similar approach could be used to generate simple mutations and thus create, update, and delete operations. Certainly, it would also be possible to use a CQRS approach and handle commands separately from queries [24].

In any case, the generic approach struggles with queries that involve custom business logic. Such queries cannot be simply mapped to an SQL query and would have to be handled differently.

As mentioned in Section 3.4 SQL views could be used to solve this problem but, while possible, may not be viable for complex business logic. If queries with complex business logic are required, the current solution provides the option to filter these queries and handle them differently. Some queries may, for example, be filtered by name and resolved using an ordinary resolver approach.

5.3 Potential areas for improvement and future work

While the current implementation of the generic GraphQL to SQL mapping supports the core features of GraphQL as well as additional features and use cases there is still more that could be explored and implemented in the future. This section lists areas and specific aspects that could be promising for future research.

5.3.1 Handling change

Modern web applications are constantly changing, and so their APIs have to handle change. One thing generated schemas (as generated by GraphQL server tools from Section 2.3.2.1) struggle with is that the GraphQL schema has to change every time the underlying SQL schema changes [34]. For BYOS the GraphQL schema does not change automatically but has to be changed by hand (or SQL views have to be employed). To prevent this pitfall and remove the schema constraint from Section 5.2.1 for our approach, it might be useful to add a configuration file which maps GraphQL object types to JOOQ tables. Using JOOQ tables ensures that the table exists on the database. If this configuration is enforced for every object type and field (other than paging-specific types and fields), it would be impossible for the names to be mismatched. If a table did not exist, a compile time error would occur.

Additionally, this would support migration by preventing subtle schema mismatches that could lead to runtime errors. For migration, SQL views could be used to ensure that the old and new API versions find their expected tables or views in the database. The view could resemble the old table until the migration to the new data representation is complete.

Further research could be conducted to explore the difficulties of migrating schemas with the suggested solution, which is a significant factor in real-world applications.

5.3.2 GraphQL features

While this thesis implemented core features of GraphQL, there remain several useful aspects of the GraphQL specification [9] that were beyond the scope and could be explored in future work.

An aspect of GraphQL that could be explored further is its extensive type system, as described in the GraphQL documentation [11].

GraphQL has a small set of predefined scalar types (`Int`, `Float`, `String`, `Boolean`, and `ID`). However, it is possible to define custom scalar types such as `Date`. Improvements of the generic mapping approach may include allowing the use of custom scalar types.

As stated in Section 3.3.3, lists of scalar values are presently not supported. In the future, if a scenario arises where a BYOS use case involves SQL arrays, it may prove beneficial to permit their representation as scalar value lists in the GraphQL schema.

Similarly to custom scalar types, GraphQL allows developers to define enumeration types and input types. The latter describes objects that can be used as arguments to allow for more complex arguments than scalar values. The processing of such arguments, and how they could be utilized in the generic mapping approach, could also be investigated in future work.

Interfaces can also be implemented in GraphQL similar to object types. An object type can implement an interface, in which case it has to include all fields of the interface. Inline fragments can be used to check for the type of an object and retrieve its specific fields. Union types act in a similar way. Instead of having all object types implement common fields, it allows different object types to be grouped under one union type. Again, inline fragments can be used to access fields of a specific type when dealing with union types. Interfaces, union types, and fragments are currently not supported and may be of interest when extending the implementation.

In addition to the features of the GraphQL schema mentioned so far, some query features, which are explained in the GraphQL documentation [10], are also yet to be implemented. These include fragments, variables, and directives.

Similarly to the inline fragments that can be used in conjunction with interfaces and union type, GraphQL offers fragments that can be defined similar to object types in the query document or the schema. These act as sets of fields that can be reused. Currently, queries that use fragments are not supported, but future endeavors could make them possible.

Sophisticated client applications would most likely not opt to manipulate the entire query string when an input is changed and instead utilize GraphQL's variable feature. A variable can be used as an argument, thus changing the query depending on the value bound to the variable. A future version of the implementation could handle such variables, for example, by flattening the AST document with variable definitions into one without, by inlining their values.

Lastly, directives are similar to variables as they allow changes to the query while keeping the query string unchanged. The GraphQL specification [9] defines the two directives `@include(if: Boolean)` and `@skip(if: Boolean)`, which makes it possible to specify if a field should be included in the response based on a variable. Some GraphQL server implementations also allow for the definition of custom directives. Future work could enable the two directives of the specification and explore common use cases for custom directives and how these could be supported.

5.3.3 Error handling

The GraphQL specification [9] defines that if errors occur, the value of the `data` entry should be `null` and the `errors` entry should be an array of errors. In its current state, the implementation supports this behavior for validation errors. However, custom errors are not possible, as they are an example of business logic. Currently, if internal errors occur due to the GraphQL query or schema not meeting the implementation constraints, the caller receives an internal server error with error code 500. An improvement would be to make these errors more verbose by adding a description to the `message` field and adjusting the error code where appropriate. However, the error messages should be chosen carefully, as reporting that a table was not found or that a relationship between two tables could not be found can reveal information about the database schema.

5.3.4 Supporting additional use cases

As mentioned in Section 3.3.1, the functionality of GraphQL arguments depends completely on their individual implementation.

A generic approach cannot offer the limitless potential that a developer could access if they were to create their own resolver functions. As mentioned in Section 5.2.3 queries that exceed data reading are challenging. The same applies to arguments that perform tasks other than what is currently provided (filtering on equality, sorting based on fields, and pagination).

It is possible to achieve all of these functionalities with the current implementation using SQL views, as previously mentioned in Section 3.4.

However, it might be a requirement to allow queries to access data that is not directly present in the database without using views. For example, if not all developers have access to the database. One possibility to achieve this would be to add the possibility to define virtual fields or tables. This approach would be similar to what Join Monster [19] provides in the form of computed columns [35] and derived tables [36]. The former denotes providing a JavaScript or SQL function to compute a virtual column, and thus a field based on other columns. The latter means that SQL table expressions can be used instead of table names, which works as a virtual table. In BYOS, such specific configurations would be made in JOOQ.

Since backward pagination, as discussed in Section 3.2.5, is presently not supported but operates similarly to forward pagination, the functionality of the `last` and `before` arguments may be implemented in the future.

Although the implementation discussed in this thesis supports several conventional use cases of GraphQL arguments, offering support for more would most certainly be useful. Currently filtering is only possible based on equality, since arguments are translated into conditions, that are then used in `WHERE` clauses. To provide more elaborate filtering, it might be useful to expose more of SQLs comparison operators. This could be done, similar to Hasura [17], by using input objects instead of scalars for arguments. In this way, in addition to the `EQ` operator, operators such as `NE`, `LT`, `LE`, `LTE`, `GT`, `GE`, and `GTE` could be exposed. In a similar way string matching operators such as `LIKE` and many more `SQL` operators could be exposed. These would have to be specified in the schema and are not automatically added to queries and fields as they would be when using a GraphQL server tool that automatically generates the GraphQL schema (see Section 2.3.2.1). This would allow developers to choose what filter options should be made available on which queries and fields.

One powerful aspect of SQL databases that is, except for the `totalCount` field, only usable when using SQL views, is aggregation. Views can be used to provide fields that hold aggregates over a table.

Adding some form of generic aggregation over tables in the future should be easy to implement and might prove useful. If a built-in way to do this would be required, one way would be to, like Join Monster [19], allow virtual fields that compute a value with provided SQL code [35]. Again, differentiating it from Join Monster, BYOS would have users define type-safe JOOQ code instead of having SQL code in strings. A more elaborate approach to offering aggregation would be to allow the definition of aggregate top level fields, similar to those generated by Hasura [37]. The fields on the aggregate type of such a field represent the different aggregate functions (such as `MAX`, `COUNT`, `SUM`, etc.) on which the fields to be aggregated can be chosen. It would be equally possible to flip the order, having the fields representing aggregate functions on the fields that represent columns.

5.3.5 Security

This section investigates the security of the implementation as well as security-related topics that could arise when applying it to a real-world application.

An aspect that has to be considered when using user input in SQL queries is the option of SQL injections. Since JOOQ is used for all SQL operations and no plain SQL operations are used, the implementation provides an API that prevents SQL injections.

A potential issue with other tools is that the GraphQL schema is generated instead of created by developers, possibly exposing tables and columns that should not be made public. Since only the tables and columns that are represented in the GraphQL schema, are exposed, everything else is inaccessible.

Malicious users could request large amounts of data, thus overloading the server. One way to prevent this, as well as to prevent users from accidentally request too large amounts of data, is to limit the page size. A common practice is to use pagination, making the `first` argument required, and returning a custom error when its value extends a certain threshold. Therefore, this would be another case where more error functionality (see Section 5.3.3) would be useful. Additionally, it might prove helpful to limit the depth of GraphQL queries since, even with a page limit, malicious users could issue a query of arbitrary depth. While the improved scalability of the mapping approach reduces the effectiveness of such an attack, very large SQL queries might still slow the database. Further versions of the implementation may include such features and utilize custom errors in their realization.

Since most APIs are not public, future research may also investigate how authentication and authorization are used in GraphQL, whether the implementation needs to be extended to support them, and if so, how. Regarding authentication, authenticating users via the HTTP Authorization header within application-specific code may be sufficient. As for authorization, numerous methods are available. Especially when seeking to replicate the row-level security features of the RDBMS, certain authentication methods may be better suited than others.

6 Conclusion

In this thesis, we set out to conceptualize a method for transpiling GraphQL queries into SQL queries. Our primary goal was to bridge the gap between the flexible nature of GraphQL and the structured querying capabilities of SQL, while avoiding the N+1 problem. This should additionally result in less boilerplate code, and thus less effort for developers.

To achieve this, we first introduced key concepts of GraphQL and the GraphQL AST. Furthermore, existing approaches for mapping GraphQL to SQL were presented and their limitations discussed.

Subsequently, the approach for our generic mapping, with its challenges and limitations, was explained.

In the next part, the implementation of the prototypical application utilizing the previously explained approach was presented.

The improvement in performance, when comparing our implementation to the naive approach, was shown based on an example.

Finally, it was discussed how the prototype could be improved and extended to an extensive library.

According to the author's assessment, the objectives were achieved. The N+1 problem does not occur for the presented solution. However, it should be noted that a GraphQL query does not necessarily result in exactly one SQL query, but rather each root field does. This means that if a GraphQL query consists of multiple root fields, one SQL query is issued for each one. Since the resolver approach is not used, the redundancy and the amount of code required when the schema is expanded are dramatically reduced. When the SQL schema is expanded, the GraphQL schema has to be expanded accordingly, and relationships between the new and existing tables have to be registered.

Bibliography

[1] Graphql. *GitHub - graphql/dataloader*. URL: https://github.com/graphql/dataloader (visited on July 18, 2023).

[2] Data Geekery GmbH. *jOOQ: The easiest way to write SQL in Java*. URL: https://www.jooq.org/ (visited on May 24, 2023).

[3] Brenda Jin, Saurabh Sahni, and Amir Shevat. *Designing Web APIs. Building APIs That Developers Love*. O'Reilly Media, Inc., Sept. 20, 2018. Chap. 1. What's an API?

[4] Eve Porcello and Alex Banks. *Learning GraphQL. Declarative Data Fetching for Modern Web Apps*. O'Reilly Media, Inc., Jan. 1, 2018. Chap. 1. Welcome to GraphQL.

[5] Junwen Yang et al. "How not to structure your database-backed web applications". In: May 27, 2018. DOI: 10.1145/3180155.3180194. URL: https://doi.org/10.1145/3180155.3180194.

[6] Mike Cronin. "What is the N+1 Problem in GraphQL? - The Marcy Lab School - Medium". In: (Dec. 10, 2021). URL: https://medium.com/the-marcy-lab-school/what-is-the-n-1-problem-in-graphql-dd4921cb3c1a (visited on Apr. 30, 2023).

[7] Lee Byron. *GraphQL: A data query language*. Sept. 14, 2015. URL: https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/ (visited on Apr. 30, 2023).

[8] Emily Olin. *The Linux Foundation Announces Intent to Form New Foundation to Support GraphQL*. Nov. 6, 2018. URL: https://www.linuxfoundation.org/press/press-release/intent_to_form_graphql (visited on Apr. 30, 2023).

[9] *GraphQL*. Oct. 2021. URL: https://spec.graphql.org/October2021/#sec-Document (visited on Apr. 30, 2023).

[10] *Queries and Mutations | GraphQL*. URL: https://graphql.org/learn/queries/ (visited on July 11, 2023).

[11] *Schemas and Types | GraphQL*. URL: https://graphql.org/learn/schema/ (visited on July 11, 2023).

[12] Eve Porcello and Alex Banks. *Learning GraphQL. Declarative Data Fetching for Modern Web Apps*. O'Reilly Media, Inc., Jan. 1, 2018. Chap. 4. Designing a Schema.

[13] Adam Hannigan. "Understanding the GraphQL AST. - Adam Hannigan - Medium". In: (Feb. 19, 2019). URL: https://adamhannigan81.medium.com/understanding-the-graphql-ast-f7f7b8e62aa4 (visited on July 21, 2023).

[14] Eve Porcello and Alex Banks. *Learning GraphQL. Declarative Data Fetching for Modern Web Apps*. O'Reilly Media, Inc., Jan. 1, 2018. Chap. 3. The GraphQL Query Language.

[15] Graphql-Java. *GitHub - graphql-java/java-dataloader*. URL: https://github.com/graphql-java/java-dataloader (visited on July 25, 2023).

[16] *On GraphQL-to-SQL*. May 21, 2020. URL: https://productionreadygraphql.com/blog/2020-05-21-graphql-to-sql (visited on May 24, 2023).

[17] *Instant GraphQL APIs on your data | Built-in Authz & Caching*. URL: https://hasura.io/ (visited on May 24, 2023).

[18] *Graphile | Powerful, Extensible and performant GraphQL APIs Rapidly*. URL: https://www.graphile.org/postgraphile/ (visited on May 24, 2023).

[19] *Introduction - Join Monster*. URL: https://join-monster.readthedocs.io/ (visited on May 24, 2023).

[20] Graphql. *GitHub - graphql/graphql-js*. URL: https://github.com/graphql/graphql-js (visited on Aug. 2, 2023).

[21] Kexugit. *Patterns in practice - convention over configuration*. Aug. 17, 2015. URL: https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-in-practice-convention-over-configuration (visited on Sept. 1, 2023).

[22] *Pagination | GraphQL*. URL: https://graphql.org/learn/pagination/ (visited on Aug. 4, 2023).

[23] *GraphQL Cursor Connections Specification*. URL: https://relay.dev/graphql/connections.htm (visited on Aug. 4, 2023).

[24] Martin Fowler. *bliki: CQRS*. URL: https://martinfowler.com/bliki/CQRS.html (visited on Sept. 25, 2023).

[25] Lukas Eder. *Implement a GraphQL to SQL translator · Issue 10122 · jOOQ/jOOQ*. Apr. 25, 2020. URL: https://github.com/jOOQ/jOOQ/issues/10122 (visited on Aug. 16, 2023).

[26] *Kotlin Programming Language*. URL: https://kotlinlang.org/ (visited on July 11, 2023).

[27] FasterXML. *GitHub - FasterXML/jackson*. URL: https://github.com/FasterXML/jackson (visited on July 25, 2023).

[28] Graphql-Java. *GitHub - graphql-java/graphql-java*. URL: https://github.com/graphql-java/graphql-java (visited on July 25, 2023).

[29] *Spring boot*. URL: https://spring.io/projects/spring-boot (visited on Aug. 17, 2023).

[30] *Spring initializr*. URL: https://start.spring.io/ (visited on Aug. 16, 2023).

[31] *PostgreSQL*. URL: https://www.postgresql.org/ (visited on Aug. 10, 2023).

[32] jOOQ. *sakila/postgres-sakila-db at main · jOOQ/sakila*. URL: https://github.com/jOOQ/sakila/tree/main/postgres-sakila-db (visited on Aug. 10, 2023).

[33] *OFFSET is bad for skipping previous rows*. URL: https://use-the-index-luke.com/sql/partial-results/fetch-next-page (visited on Aug. 25, 2023).

[34] *Generated GraphQL schemas and schema design*. Jan. 9, 2022. URL: https://www.apollographql.com/blog/tooling/apollo-codegen/generated-graphql-schemas-and-schema-design/ (visited on Aug. 26, 2023).

[35] *3. Add metadata to fields - Join Monster*. URL: https://join-monster.readthedocs.io/en/latest/field-metadata/ (visited on Sept. 4, 2023).

[36] *2. Map objects to tables - Join Monster*. URL: https://join-monster.readthedocs.io/en/latest/map-to-table/ (visited on Sept. 4, 2023).

[37] *Postgres: Aggregation Queries | Hasura GraphQL Docs*. URL: https://hasura.io/docs/latest/queries/postgres/aggregation-queries/ (visited on Sept. 4, 2023).