Implementation and performance analysis of GraphQL schema delegation

Samuel Barnholdt
Abstract

Implementation and performance analysis of GraphQL schema delegation

Samuel Barnholdt

GraphQL schema delegation is the name of the architectural pattern of delegating queries from one GraphQL server to another. This project aimed to find different methods and techniques for implementing schema delegation, implement one of them and evaluate the performance. Few methods were found, and the npm library graphql-tools was seemingly the only dedicated tool available to implement schema delegation. One GraphQL server was automatically generated with a tool called Hasura. graphql-tools was used to implement another server, delegating to the automatically generated server. Tests were run with the help of the resources from the Linköping GraphQL benchmark. The automatically generated server served as a baseline. Test results found that, overall, the delegating server had significantly worse performance than the generated server. There were, however, some test results where the servers had similar performance. We conclude that there seem to be situations where graphql-tools perform well and suggest methods to increase the performance in areas where graphql-tools do not perform well. We also suggest implementing GraphQL schema delegation without the use of any tools.
Contents

1 Introduction 4

2 Background 5
  2.1 GraphQL schema delegation ................................................... 8
  2.2 Schema stitching ................................................................. 8
  2.3 Aligned schemas ................................................................. 9
  2.4 PostgreSQL ................................................................. 9
  2.5 Hasura GraphQL Engine ......................................................... 9
  2.6 LinGBM Benchmark .............................................................. 10
    2.6.1 Choke points and query templates ........................................ 10
    2.6.2 Performance metrics ....................................................... 12
    2.6.3 Dataset generator and schema .......................................... 13
    2.6.4 Test driver throughput .................................................... 14
    2.6.5 Test driver QET/QRT ....................................................... 15
    2.6.6 Query generator ............................................................. 15
  2.7 Search engines ............................................................... 15
    2.7.1 Google Search .............................................................. 15
    2.7.2 SearX ................................................................. 15
    2.7.3 DuckDuckGo ............................................................... 16
    2.7.4 Microsoft Bing ............................................................ 16

Phase 1

3 Exploring methods of schema delegation 17
  3.1 Keyword: "graphql schema delegation" ..................................... 17
  3.2 Keyword: "combining graphql schemas" .................................... 18
  3.3 Keyword: "graphql delegating queries" ..................................... 18
  3.4 Keyword: "graphql different types of schema delegation" ............. 19
  3.5 Keyword: "graphql querying other servers" ............................... 19

Phase 2

4 Results 20
1 Introduction

GraphQL \cite{9} is a new approach to build Web-based data access interfaces. Web APIs built with GraphQL allow the client, not the server, to determine exactly what data it needs and when. Since it first published in July 2015 it has become tremendously popular, and lists tech giants like Airbnb, Facebook and Twitter among its users \cite{10}. As GraphQL has increased in popularity, so has the number of resources describing approaches of building GraphQL servers, optimization techniques and performance pitfalls (some examples \cite{38, 37, 11, 13}). One of these techniques is what is referred to as \textbf{schema delegation}. Schema delegation is the name for the architectural pattern of delegating the execution of queries from one server to another. It can be used in many different scenarios, with the obvious case being when one simply wants to delegate a query to another server.

Studies reviewing and comparing the performance of different GraphQL implementations and techniques, like schema delegation, are rare. Moving forward, if we want to be able to understand the pros and cons of different GraphQL implementations and build efficient GraphQL servers with optimal performance for our desired solution; we can not continue relying on anecdotal evidence. If not for our understanding of GraphQL implementations, at the very least for our ability to verify it. To solve the problem of relying on anecdotal evidence, Linköping University is developing a performance benchmark in order to get more reliable information when testing the performance of different GraphQL implementations. \cite{18}.

This degree project aims to find and describe different methods of schema delegation, implement one of them and analyze its performance with the help of the LinGBM benchmark. This is done with the purpose of providing information about the performance and the limitations of GraphQL schema delegation.

This will be done in four phases:

- \textbf{Phase 1}
  In Phase 1, possible methods of schema delegation will be explored. At the end of this phase the one method will be chosen to be implemented.
- **Phase 2**  
  In this phase the method of schema delegation found in Phase 1 will be implemented.

- **Phase 3**  
  In this phase the performance of the implementation will be tested.

- **Phase 4**  
  In this phase the results of Phase 4 will be documented and evaluated.

# 2 Background

GraphQL [9] is a tremendously popular new approach to build Web-based data access interfaces. Hartig et al. [42] found that as of June 2018, there were more than 20,000 repositories on GitHub matching the keyword “graphql” and 2,000. They also found over 37,000 repositories dependent on GraphQL reference implementations.

GraphQL is an alternative to Representational State Transfer (REST) [5]. One key component and likely a factor in GraphQL’s increase in popularity as an alternative to REST is that GraphQL allows the client to determine what data it retrieves. In consequence, GraphQL API’s do not need multiple endpoints accessing the same resource; as the client can query the data it needs for the situation at hand. Giving the control to the client also reduces overfetching. A practical study by Brito et al. [2] on migrating API clients from REST to GraphQL found that GraphQL could reduce the size of JSON documents returned by REST APIs by 94% (in terms of fields). Brito et al. [3] also found in an controlled experiment that the participants had to spend less time implementing service queries when using GraphQL as compared to REST. It should also be stated that a study by Mikula et al. [29] suggest that the benefits of GraphQL do not come with a decline in performance as compared to REST, as their results found that GraphQL and REST have similar performance in terms of adding, editing, deleting and retrieving data.

Building a GraphQL server involves building a schema and so-called resolvers for that schema. The former can be done as is showed in Listing 1. In the example, we define a schema for a simple database, with students and professors. We also define a query to get a professor.
1 type Student {
2   id: ID
3   name: String
4   courses: [String]
5 }
6
7 type Professor {
8   id: ID
9   name: String
10  student: Student
11 }
12
13 type Query {
14   getProfessorById(id: String): Professor
15 }
16
Listing 1: Simple GraphQL schema

In the above schema we have defined three object-types, Student, Professor, and Query. Each object-type contains declarations of its fields and their corresponding types. As an example, the Student type contains two fields, name and id, with predefined scalar types ID and String. The Student type also contains the field courses of type List containing the scalar type String.

GraphQL object-types can also contain fields that are object-types. The Professor type contains the field student of type Student.

One special type that has to exist in every GraphQL schema is the Query type. It is the ”root”, the ”main entrance”, to the GraphQL schema. If one wants to access the database through the GraphQL endpoint one has to do so through the Query type. In our Query type we have the query getProfessorById of type Professor. getProfessorById also receives the argument id of type String. Arguments in GraphQL work like arguments in functions of regular programming languages. They can be scalar typed, but there also input-types that are constructed similarly to object-types that can be used as arguments, but can not be queried (i.e. object-types can not contain input types). Input-types can only contain scalar types or other input types.

Should we want to use the getProfessorById query we would first have to
build a GraphQL query. Listing 2 contains an example of a GraphQL query. GraphQL queries are nested expressions, in which the innermost are scalar-typed fields (i.e. the getProfessorById expression could not contain only the student field, all object-typed fields have to contain subexpressions). All types in a GraphQL schema act as ”pointers” to other types, the first expression of every GraphQL query always begin at the Query type as it is the only ”pointer” we have access to initially.

Listing 2: Example of a GraphQL query

```graphql
query {
  getProfessorById (id: "1"){
    name
    student {
      name
    }
  }
}
```

Listing 3: Example of a possible result of the GraphQL query in Listing 2 could look like.

```json
{
  "data": {
    "name": "Olaf",
    "student": {
      "name": "Samuel"
    }
  }
}
```

Listing 3: Example of a possible result of the GraphQL query in Listing 2

How the queries are resolved, i.e. the retrieval of the actual data, is not solved by GraphQL. GraphQL provides a language for describing databases and an interface which one can use to access the data contained in them. But one has to implement the ”actual” retrieval of the data. As an example of how an implementation of a resolver could look like, look at Listing 4.

```javascript
function getProfessorById(obj, args, context, info){
  const professor = getProfessorFromDataBase(args.id) // This is not a real function
  return professor;
}
```

Listing 4: Example of a possible implementation of a resolver function for the getProfessorByld query
GraphQL resolvers receive four arguments:

1. **parent** (also referred to as **obj**). The previous object in the resolver chain. A resolver chain is created when a query asks for an object type. It has to also ask for a field in the asked for object type, thus creating a "chain" (query → type → field). The resolver for field has access to the return value of the resolver of type.

2. **args** Arguments for the GraphQL query.

3. **context** Value provided to every resolver, containing important contextual information. Could be the current logged in user as an example.

4. **info** Metadata containing information about current query as well as schema information.

The important take away is that you can do anything in your resolvers as long as what you return follows the format specified in the GraphQL schema.

### 2.1 GraphQL schema delegation

Schema delegation is the name for the architectural pattern of delegating the execution of queries from one GraphQL server to another. One server, as the name suggests, resolves part of/the entirety of its schema through delegating the queries to another server. This can be useful in different scenarios, an example being when one has a GraphQL server, and wants to implement another more specific server based on the same data set and schema. One can then write a schema, and resolve it through delegating queries to the other server. Another scenario when one wants to split up a codebase into smaller, more maintainable sections. Each section can implement their own specific GraphQL server and schema, while delegating the shared parts to a shared GraphQL server.

### 2.2 Schema stitching

If one wants to combine multiple GraphQL schemas one can use a technique called **schema stitching**[37]. Schema stitching combines GraphQL schemas and creates a proxy layer delegating requests to the appropriate schema. Note that this is different from **schema merging**, where one simply merges multiple schemas (useful for building a local service schema from many individually-managed parts)[38].

8
2.3 Aligned schemas

We will refer to two schemas being aligned when the fields of all types representing the same entity, bar the Query type, have the same name. For clarification, view Listing 5. Even though the name of the professor type and the fields of Query type differ, the fields of the professor type do not. Had the emailAddress field in schema 2 been named ”emailaddress”, we would consider the schemas as unaligned.

```graphql
// Schema 1
type Query {
  professor : professor
}

// Schema 2
type Query {
  getProfessor : Professor
}

// Schema 1

type professor {
  id: ID!
  emailAddress : String
}

// Schema 2

type Professor {
  id: ID!
  emailAddress : String
}
```

Listing 5: Aligned Schema

2.4 PostgreSQL

The GraphQL servers in this project use PostgreSQL databases to store and retrieve data. PostgreSQL is an open source object-relational database system created in 1986 and has been actively developed since. PostgreSQL has millions of active users.

2.5 Hasura GraphQL Engine

This project uses the Hasura GraphQL engine to automatically generate a GraphQL server to delegate to and compare performance against. The
Hasura GraphQL Engine is a tool that can be used to automatically create a GraphQL server from an existing PostgreSQL database [11].

### 2.6 LinGBM Benchmark

This project uses the tools provided by the LinGBM benchmark to evaluate the performance of the GraphQL servers in the project. The LinGBM benchmark is a performance benchmark under development by Linköping University. It is developed with the purpose of getting more reliable information when testing the performance of different GraphQL implementations [18]. Datasets and schemas are provided to use when testing. The benchmark focuses on the following two scenarios:

1. The scenario in which data from a legacy database is exposed to a read-only GraphQL API, where the user has defined the GraphQL schema.

2. The scenario in which data from a legacy database is exposed to a read-only GraphQL API, where the GraphQL schema is generated automatically.

#### 2.6.1 Choke points and query templates

LinGBM Benchmark have identified the primary technical challenges for the systems facing above scenarios. These challenges are called ”choke points”. They are described in full at the LinGBM github page [19]. Some examples are:

- **CP 1.1 Multi-attribute retrieval**
  Being able to efficiently support queries that retrieve multiple attributes of the fetched data objects.

- **CP 2.4 Traversal of relationships that form cycles**
  Being able to efficiently fetch data containing directed cycles, i.e. traversal of the data objects potentially revisiting the same data object multiple times. Efficiency in this chokepoint would be refraining from fetching the same data twice, and instead caching data that has been visited already.

- **CP 4.5 Multiple filter conditions**
  Being able to efficiently fetch data using multiple filter conditions. In
addition to the challenge of filtering the data, each filter may not be equally selective. Therefore, performance may be effected by the order in which the filters are evaluated.

These chokepoints are used as a basis for the so called ”Query Templates” \cite{lin2019gbm} of the LinGBM Benchmark. The query templates are definitions of GraphQL queries used for testing a GraphQL server, and they cover all of the identified chokepoints. As an example, Listing \ref{lst:query_template_5} contains query template 5. It covers four chokepoints, one of those being \textbf{CP 2.4}. The query retrieves a department, the university it belongs to, the graduate students of said university, the email address and the department of each student. For each department of each graduate student, the same data is retrieved; resulting in directed cycles and therefore covering the challenge described by \textbf{CP 2.4}. Brief descriptions of all chokepoints as well as query template coverage can be viewed in Table \ref{table:chokepoints} (last page).
2.6.2 Performance metrics

LinGBM considers the following notions for their performance metrics \cite{22}.

1. **Throughput (TP)**
   Throughput is the number of queries that are processed completely by a GraphQL server within a set time interval. A query is considered processed completely when the entire result is received by the client. \(aTPt\) is the average throughput measured when running the same query workload multiple times.

2. **Query execution time (QET)**
   Query execution time is the time it takes between sending a query and receiving the complete result.
3. **Query response time (QRT)** Query response time is the time it takes between sending a query and the begin of receiving the query result.

The following performance metrics are based on above notions.

1. \( aQETt \) (average query execution time per template) is the average QET of a set of executions with the same template. This measurement is reported together with the standard deviation of the set.

2. \( aQRTt \) (average query response time per template) is the average QRT of a set of executions with the same template. This measurement is reported together with the standard deviation of the set.

3. \( aQETq \) (average query execution time) is the average QET of a set of executions with the same query. This measurement is reported together with the standard deviation of the set.

4. \( aQRTq \) (average query response time) is the average QRT of a set of executions with the same query. This measurement is reported together with the standard deviation of the set.

### 2.6.3 Dataset generator and schema

LinGBM provides a dataset generator [14] that can be used for generating a SQL dump file, which one can then import to a MySQL/PostGreSQL server. The dataset generated represent corresponds to a University scenario according to Figure [1] with students, professors, departments etc [15]. The data can be generated at different scales, with different scale factors.
Figure 1: Entity-Relationship diagram of the LinGBM database schema.

The GraphQL schema is written based on the diagram, and contains a object for each entity [20]. The entire schema can be viewed at the LinGBM github page [17]. As an example, view Listing 7. The GraphQL type Department describes the entity Department as it is described in the entity-relationship diagram.

```graphql
1 type Department
2 {
3   id: ID!
4   subOrganizationOf: University
5   head: Professor
6   researchGroups: [ResearchGroup]
7   faculties: [Faculty]
8   professors: [Professor]
9   lecturers: [Lecturer]
10  graduateStudents: [GraduateStudent]
11  undergraduateStudents: [UndergraduateStudent]
12 }
```

Listing 7: GraphQL schema for Department

2.6.4 Test driver throughput

LinGBM developed a test driver for throughput experiments [26]. The test driver simulates multiple clients concurrently sending queries to the GraphQL server. It consists of threads for a ”master” and clients. The master controls the clients and sends queries to them which they then send to the server. The
clients process their queries sequentially. When a client receives a response from the server, it sends log data to the master containing the identifier of the query, the execution time, and an error code (0 for success, 1 for failure). If the client has more queries in queue, it continues by processing the next one. After a given time period, the master stops all clients and outputs a directory containing processed statistics from the log data it received from the clients during the run.

2.6.5 Test driver QET/QRT

LinGBM developed a test driver for execution and response time experiments [25]. The test driver queries a GraphQL server and records the execution and response time.

2.6.6 Query generator

LinGBM developed a query generator [24]. It generates queries based on the LinGBM query templates [23], as well as values based on the data generated from the dataset generator [14]. The values are used to replace the placeholders in the query templates (i.e. they are used as arguments).

2.7 Search engines

In the research phase of this project, below search engines are used to find information on the World Wide Web.

2.7.1 Google Search

Google Search is the most popular web search engine in the world [44]. It is used to search for information on the World Wide Web (www).

2.7.2 SearX

SearX is a metasearch engine [47], i.e. a search engine that does uses other search engines and aggregates their results. It is used to search for information on the World Wide Web (www).
2.7.3 DuckDuckGo

DuckDuckGo is a web search engine that emphasizes user privacy [13]. DuckDuckGo seeks to avoid personalized search [16][13] through not using additional information about the user to provide results that it believes the user is interested in. DuckDuckGo is used to search for information on the World Wide Web (www).

2.7.4 Microsoft Bing

Microsoft Bing is web search engine [15]. It is used to search for information on the World Wide Web (www).
Phase 1

In this phase we explore methods of schema delegation. At the end of the phase we select one implementation to implement in Phase 3.

3 Exploring methods of schema delegation

The internet is a vast space, and it is hard if not impossible to make sure one has found every source of information for any specific topic. With the intent of finding as many possible methods of schema delegation we will use multiple search engines. These engines are Google [44], Bing [45], SearX [47], and DuckDuckGo [43]. Different keywords, seeking to describe schema delegation, will be searched for and approximately 40-50 results of each keyword will be explored and summarized. The research phase assumes that should it not find multiple methods then at the very least it will have found all “relevant” methods.

3.1 Keyword: ”graphql schema delegation”

- Bing
  Relevant results found here was the graphql-tools website [39]. graphql-tools is library with tools for building graphql servers [40] developed by a group of open-source developers called The Guild[36]. It contains dedicated tools for graphql schema delegation [41]. Other results were irrelevant stack overflow questions, github issues and blog posts about schema delegation, all using graphql-tools.

- Google
  Google found similar results to Bing. It also found documentation about how to implement schema stitching in C# [4]. The implementation described uses a framework called HotChocolate [27] to stitch schemas. Schema stitching is not schema delegation but it can be used as a tool for schema delegation.

- DuckDuckGo
  DuckDuckGo found similar results to Bing.
• **SearX**
  SearX found similar results to Bing. It also found GraphQL tutorials and frameworks not relevant to schema delegation, as well as tools for GraphQL schema generation (like Hasura).

Relevant findings: HotChocolate [27], graphql-tools [41].

### 3.2 Keyword: ”combining graphql schemas”

• **Bing**
  Bing found multiple blog posts and issues on schema stitching and expanding/splitting GraphQL schemas with the help of schema stitching. All posts used the graphql-tools library [37].

• **Google**
  Google found similar results to Bing.

• **DuckDuckGo**
  DuckDuckGo found similar results to Bing. It also found the Javascript library **graphql-weaver** [8]. graphql-weaver is a library used to combine multiple GraphQL endpoints. It is seemingly not intended for schema delegation but one should be able to use the library to resolve the fields of one schema through delegating to another.

• **SearX**
  SearX found similar results to Bing. It also found a small Javascript library called graphql-combine [6], used to modularise GraphQL schemas.

Relevant findings: graphql-tools [41], graphql-weaver [8].

### 3.3 Keyword: ”graphql delegating queries”

• **Bing**
  Bing found primarily tutorials and documentation about GraphQL queries.

• **Google**
  Google found GraphQL documentation as well as various graphql-tools documentation, blog posts and issues.
• **DuckDuckGo**
  DuckDuckgo found similar results to Google.

• **SearX**
  SearX found similar results to Bing, as well as some additional advanced usage of GraphQL.

Relevant findings: graphql-tools [41]

3.4 **Keyword: ”graphql different types of schema delegation”**

• **Bing**
  Bing found primarily various tutorials and documentation about GraphQL schema delegation, all using graphql-tools.

• **Google**
  Google found similar results to Bing.

• **DuckDuckGo**
  DuckDuckGo found similar results to Bing, as well as some basic GraphQL tutorials.

• **SearX**
  SearX found similar results to DuckDuckGo.

Relevant findings: graphql-tools [41]

3.5 **Keyword: ”graphql querying other servers”**

• **Bing**
  Bing found primarily various tutorials and documentation about GraphQL and implementing GraphQL servers.

• **Google**
  Google found similar results to Bing.

• **DuckDuckGo**
  DuckDuckGo found similar results to Bing.

• **SearX**
  SearX found similar results to Bing.
4 Results

It is uncertain whether this approach of searching is optimal, and if the results of this phase found all possible ways of schema delegation. What at the very least seems certain is that **graphql-tools** is the by far most popular library (and perhaps also the only dedicated tool) used for schema delegation, which is also suggested by the fact that it has over one million weekly downloads [32] as opposed to **graphql-weaver** with 62 weekly downloads [33].

The results of the research phase suggest that there are three options to implement schema delegation:

1. Using graphql-tools
   Using the graphql-tools library would possibly provide valuable information regarding complexity and performance about a popular library used by a multitude of developers. It also comes with the benefit of dedicated tools for the task at hand.

2. Using graphql-weaver
   Using graphql-weaver would provide an interesting challenge. It would provide tools one could use to implement schema delegation, but with the disadvantage of not having as many dedicated tools for schema delegation as graphql-tools.

3. Creating a custom solution
   The advantage of creating a custom solution would be complete control over the source code, as well as being able to create tools that are not only dedicated but specific to the task at hand. It would however be difficult to determine a suitable scope for the project.

There are times when one should refrain from using libraries, and I believe developers should always question whether they are using a sledgehammer to kill a fly before turning to a library for assistance. At the same time, I also believe that abstractions are the foundation of computer science. Using a popular and well-documented library like graphql-tools seems to be an obvious choice over going out of your way to implement schema delegation by yourself, and over one million weekly downloads [32] suggest that there are a vast number of developers agreeing with that statement.
Phase 2

In this phase we implement the schema delegation method found in Phase 1.

5 Setting up

This project was developed on a Macbook Pro running on macOS Catalina. The following walkthrough assumes the reader is using a computer with the same OS.

All tools were installed using homebrew [12], to install homebrew input the following command in the terminal:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

Note that commands might be outdated, and it is highly recommended to confirm that the commands are up to date on the homebrew website [12].

5.1 Postgres

Install postgres through the following command:

```
brew cask install postgresql
```

Start postgres through the following command:

```
brew services start postgresql
```

5.2 Docker

Install docker through the following command:

```
brew cask install docker
```

5.3 Maven

Install maven through the following command:

```
brew install maven
```
5.4 Generating database

Clone the LinGBM repository through the following command:

```bash
git clone https://github.com/LiUGraphQL/LinGBM.git
```

Run the following command to generate the database. This command will produce the sql files `schema_university_sql.sql`, `university0.sql`, `university1.sql`.

```bash
cd LinGBM/tools/datasetgen && mvn clean install && ./generate.sh --format PostgreSQL --consolidate Maximal -u 2
```

Login to postgres with the following command

```bash
sudo psql postgres
```

Then proceed to install the generated files. Note that you should have your current directory being `/LinGBM/tools/datasetgen`. Import the database to postgres through the following commands.

```bash
postgres=# \i schema_university_sql.sql
postgres=# \i university0.sql
postgres=# \i university1.sql
```

5.5 Set up Hasura

Get the dockerfile through the following command:

```bash
```

The below listing contains the content of docker-run.sh. Replace username with your postgres user, dbname with "linbenchmark" and port with 5432 (the default PostgreSQL port).

```bash
docker run -d --name hasura_engine -p 8080:8080 -- e HASURA_GRAPHQL_DATABASE_URL=postgres://username@host.docker.internal:port/dbname \
-e HASURA_GRAPHQL_ENABLE_CONSOLE=true \
hasura/graphql-engine:v1.2.1
```

Finally, run the file through the following command:

```bash
./docker-run.sh
```

Hasura now runs on localhost:8080. Browse to http://localhost:8080, go to the tab "Data" and select the option to track all tables to finish setting up Hasura.
5.6 Node and npm

Install node and npm through the following command:

1. `brew install node`

5.7 Project

The directory "src" was created, and the package.json found on [21] was added. Finally to finish all setup, the following command was run:

1. `npm install`
6 Implementation

The source code of this project can be found at the LinGBM Github page\[19\]. As the project is too large to explore in its entirety, we will look at how a part of the delegating server was implemented and resolved. Note that the delegating server only delegates the fields of its GraphQL schema required to perform the query templates later used for testing.

```
1 type Query {
2   graduateStudents(where: GraduateStudentWhereInput, limit: Int, order:[GraduateStudentField]) : [GraduateStudent]
3 }
4
5 interface Author
6 {
7   id: ID!
8   telephone: String
9   emailAddress: String
10 }
11
type GraduateStudent implements Author
12 {
13   id: ID!
14   emailAddress: String
15   advisor: Professor
16 }
17
```

Listing 8: A part of the LinGBM GraphQL Schema
In Listing 8 we can observe the root query `graduateStudents`. It returns a list of `GraduateStudent`, which can be observed in the same listing. In Listing 9 we can observe part of the schema generated through Hasura. The schema in its entirety can be viewed at [16]. Until otherwise stated we will refer to the schema in Listing 8 as Schema 1 (the schema of the delegating server) and the schema in Listing 9 as Schema 2 (the schema of the Hasura server). The goal is to delegate the fields id, emailAddress and advisor from Schema 1 to Schema 2. The mapping of the fields is as follows, with the fields from Schema 2 to the left, and Schema 1 to the right.

\[
\begin{align*}
nr & \rightarrow id \\
\text{emailaddress} & \rightarrow \text{emailAddress} \\
\text{professor} & \rightarrow \text{advisor}
\end{align*}
\]
6.1 delegateToSchema

Listing[11] shows the function declaration of delegateToSchema, sourced from graphql-tools’s website [41]. The declaration is written in Typescript, a typed superset of Javascript [28]. The function is used to delegate the resolution of a field to another GraphQL schema. It returns a Promise containing the type any, which as the name implies is any type. A Promise is, as described by MDN Web Docs [31], ”a proxy for a value not necessarily known when the promise is created.”. It is primarily used for asynchronous functions. The delegateToSchema function receives the below arguments.

- schema: The graphql schema to delegate to, in our case Schema 2.
- operation: The operation we want to perform on the schema.
- fieldName: The name of the field we want to retrieve.
- args: Arguments that are passed to the schema delegated to. This field is optional.
- context: Any data that is to be shared among resolvers. As an example, it can be used for authentication by including a user.
- info: Metadata used by GraphQL. This is passed down by default from GraphQL resolvers, and one does not have to worry about specifying it.
- transforms: Transforms, i.e. object manipulation or any function the developer finds suitable, to apply to the query and its results. Can be used to get fields that only are found in the schema delegated to.

Ideally, the schema we delegate to should return a type that is a superset of the type we want (see Listing[10]. We want the schemas to be aligned. When that is the case, delegateToSchema is able to resolve the sought after type with one function call; without being required to perform any transforms.
```typescript
1 type SuperSetType {
2   id: Int!
3   students: Int!
4   department: Int!
5 }
6

7 type SubsetType {
8   id: Int!
9   students: Int!
10 }
```

Listing 10: Example of superset type

What happens otherwise is shown in below sections.

```typescript
1 delegateToSchema(options: {
2   schema: GraphQLSchema;
3   operation: 'query' | 'mutation' | 'subscription';
4   fieldName: string;
5   args?: { [key: string]: any };
6   context: { [key: string]: any };
7   info: GraphQLResolveInfo;
8   transforms?: Array<Transform>;
9 }): Promise<any>
```

Listing 11: delegateToSchema

### 6.2 Schema transforms

Schema transforms are used to transform a query and its results. One transform that is used in the implementation is `WrapQuery`. `WrapQuery` can be used to modify the `SelectionSetNode` (object containing the fields asked for and metadata) and the result at a desired "path". Path in this case could be the name of a query. As an example, look at Listing 9 the path could be `[‘graduatestudent’]` or `[‘graduatestudent’, ‘nr’].

```typescript
1 new WrapQuery(
2   // path at which to apply wrapping and extracting
3   [path],
4   // modify the SelectionSetNode
5   (subtree: SelectionSetNode) => subtree,
6   // how to process the data result at path
7   result => result,
8 ),
```

Listing 12: WrapQuery
There are two common use-cases of WrapQuery in this project. One is covered by the function **WrapFields**. WrapFields is implemented to add fields to a path, of type SelectionSetNode. An example usecase can be found in Listing 16 where we only want to add the nr field. WrapFields receives the following arguments:

- **path**: Path from where to retrieve the fields.
- **fields**: Fields to add.

It adds fields to the SelectionSetNode that would otherwise not have been queried.

```javascript
export const WrapFields = (path, fields) => {
  return new WrapQuery(
    // path at which to apply wrapping and extracting
    [path],
    (subtree) => {
      if (!subtree)
        return {
          kind: "SelectionSet",
          selections: [...fields],
        };
      subtree.selections = [...subtree.selections, ...fields ];
      return subtree;
    },
    // how to process the data result at path
    (result) => {
      return result;
    }
  );
}
```

Listing 13: WrapFields

Another usecase is covered by the function **GetField**, used to retrieve a specific field at a path and also return it. An example usecase could be found in Listing 20 where we want to return the emailaddress and nothing else.
export const GetField = (path, field) => {
    return new WrapQuery(
        // path at which to apply wrapping and extracting
        [path],
        (subtree) => {
            if (!subtree)
                return {
                    kind: "SelectionSet",
                    selections: [createField(field)],
                };
            subtree.selections = [...subtree.selections,
                createField(field)];
            return subtree;
        },
        // how to process the data result at path
        (result) => {
            return result && result[field];
        }
    );
}

Listing 14: GetField

GetField uses the "helper" function createField, Listing 15, that creates a SelectionSetNode from a string. As an example GetField asks for a field in the form of a string, and then uses createField to create a SelectionSetNode. If one wants to retrieve the "emailaddress" field, they would not have to create a SelectionSetNode; and can instead just pass a string. The function createTwoLayeredField can be observed in the same Listing. As implied by the name, it creates a field with a selection set, containing more fields. It is used for the same reason as createField, but in this case we want to create a SelectionSetNode for a field and its subfields.
export const createField = (name) => {
    return {
        kind: "Field",
        name: {
            kind: "Name",
            value: name,
        },
        arguments: [],
        directives: [],
    };
};

export const createTwoLayeredField = (name, fields) => {
    const field = createField(name);
    field.selectionSet = {
        kind: "SelectionSet",
        selections: fields,
    };
    return field;
};

Listing 15: Helper functions

6.3 Resolving graduateStudents

To resolve, or rather help resolve, the query graduateStudents, WrapFields (Listing 13) is used to retrieve the field nr from Schema 2. The reason we retrieve the nr (the primary key of graduatestudent) field even though it is not queried is to be able to use it to then resolve the fields of GraduateStudent with it. One could analyse the argument info and seek to be more adaptive with which tools should be used when certain fields are asked for, but that approach was disregarded to avoid increasing the code complexity further.
graduateStudents: async (parent, args, context, info) => {
  const selectionSet = info.fieldNodes[0].selectionSet;
  const store = storeAndRemoveAdvisor(selectionSet);
  if (args.where) modifyWhereClause(args.where);
  if (args.order) modifyOrderClause(args);

  const results = await delegateToSchema({
    schema,
    operation: "query",
    fieldName: "graduatestudent",
    args,
    context,
    info,
    transforms: [WrapFields("graduatestudent", [createField("nr")])],

    if (store) selectionSet.selections.push(store);
    return results;
  },

  Listing 16: Resolver for graduateStudents
The graduateStudents query can receive the arguments where and order. The functions modifyWhereClause and modifyOrderByClause, Listing 17, are used to modify the arguments to fit according to the implementation in Schema 2. If for example we want to order by the field id we need to translate it to order by the field nr instead.

```
const modifyWhereClause = (where) => {
  if (where.university) {
    if (where.university.nr) {
      where.university.nr = { _in: [parseInt(where.university.nr)] };  
    }
  }
};
```

```
const modifyOrderByClause = (args) => {
  const clauses = args.order;
  let order_by = {};
  clauses.map((item) => {
    if (item === "id") order_by.nr = "asc";
    else order_by[item.toLowerCase()] = "asc";
  });
  args.order_by = [order_by];
};
```

Listing 17: Functions applied to the arguments of graduateStudents

One problem that was encountered was that in Schema 1 the field advisor is of type Professor, while in Schema 2 it is of type Int. This created a type error every time the advisor was queried for, even though the method of resolving the field was defined in GraduateStudent. This is because the fields we query in Schema 1 are also queried in Schema 2. If the field emailAddress is asked for, it is ignored since it does not exist on Schema 2. When the advisor is asked for, the advisor of type Int is returned, and GraphQL then throws an error since it expected a return value of type Professor. The function `storeAndRemoveAdvisor` (Listing 18) is used as a not so elegant fix, in consequence of not being able to find a better/more suitable solution. It removes the advisor from the selection set, to make sure that it is not queried in Schema 2. When the schema delegation is finished it is added to the selection set. The reason it is added on is for GraphQL to be able to identify that it is queried, and later on return it after it is resolved. If we did not add it back GraphQL would not recognize that it had been queried, and
would not return it.

```javascript
export const storeAndRemoveAdvisor = (selectionSet) => {
    let store = null;
    selectionSet.selections.map((item, i) => {
        if (item.name.value === "advisor") {
            store = selectionSet.selections[i];
            delete selectionSet.selections[i];
            selectionSet.selections = selectionSet.selections.filter((item) => {
                return item != null;
            });
        }
    });
    return store;
};
```

Listing 18: Function declaration of storeAndRemoveAdvisor

### 6.4 Resolving id

As observed in the previous section, we query the field `nr` from Schema 2. To resolve the id we simply get the field from the parent (root). The parent contains the return value of the previous resolver in the resolver chain. A resolver chain is created when a query asks for a object type. It has to also ask for a field in the asked for object type, thus creating a "chain" (query \(\rightarrow\) type \(\rightarrow\) field). The resolver for field has access to the return value of the resolver of type.

```javascript
GraduateStudent: {
    id: async (parent, args, context, info) => {
        return parent.nr;
    },
}
```

Listing 19: Resolving id

### 6.5 Resolving emailAddress

To resolve the field emailAddress we make use of the field `nr`, passed down by the parent. If the parent does not have the field `emailAddress`, i.e it was already resolved, the nr field is used to query `graduastudent_by_pk` in Schema 2. The query retrieves a graduate student with the same nr as the
given argument. `GetField` is then used to retrieve the field `emailaddress` and return it.

Presumably, this is not an optimal solution, since both `emailaddress` and `emailAddress` are of the same type. The intended approach was to try to either rename the field or map the fields to each other, but the attempt to find a solution was unsuccessful.

The field `emailAddress` can be queried similarly as `nr`, i.e. always add the field `emailaddress` whenever a type of GraduateStudent is queried. The problem with that solution is that it makes it difficult to preserve the recursive property of GraphQL. It was disregarded to avoid further increasing the code complexity. Note that this is the case for every field. The entire implementation relies on all resolvers for all fields in Schema 1 being able to retrieve `nr` from the parent. If the fields do not have the same name and type as the corresponding field in Schema 2, the fields are retrieved by querying Schema 2 with the help of the `nr`.

```javascript
1   emailAddress: async (parent, args, context, info) => {
2     if (!parent.emailAddress) {
3         const student = await delegateToSchema({
4             schema,
5             operation: "query",
6             fieldName: "graduatestudent_by_pk",
7             args: {
8                 nr: parseInt(parent.nr),
9             },
10             context,
11             info,
12             transforms: [GetField("graduatestudent_by_pk", "emailaddress")],
13         });
14         return student;
15     }
16     return parent.emailAddress;
17 },
```

Listing 20: Resolving id
6.6 Resolving advisor

To resolve the advisor we follow the same procedure as in Section 6.5. We use the helper function `graduateStudentByPk`, found in Listing 22. Note that the function `createTwoLayeredField` is used here. The reason for that is that the type Professor in Schema 1 contains fields that are not scalar-typed, i.e. they might have their own selection of subfields. The field nr is retrieved to make it possible to resolve those subfields.

```javascript
advisor: async (parent, args, context, info) => {
    if (!parent.professor) {
        const student = await graduateStudentByPk(
            parent.nr,
            [createTwoLayeredField("professor", [createField("nr")])],
            schema,
            parent,
            context,
            info
        );
        if (!student || !student.professor) return null;
        return student.professor;
    }
    return parent.professor;
},
```

Listing 21: Delegating fetch of professor
export const graduateStudentByPk = async (pk, fields, schema, parent, context, info) => {
  return await delegateToSchema({
    schema,
    operation: "query",
    fieldName: "graduatestudent_by_pk",
    args: {
      nr: pk,
    },
    context,
    info,
    transforms: [
      WrapFields("graduatestudent_by_pk", [createField("nr"), ...fields]),
    ],
  });
};

Listing 22: Delegating fetch of professor
Phase 3
In this phase we test the performance of the implementation.

7 Method

To evaluate the performance of the delegating server we will look at the performance metrics described in Section 2.6.2: aQET, aQRT, aTP. These metrics will be evaluated with the help of the LinGBM testdrivers. All experiments will be run on both the generated server and the delegated server. The generated server will serve as a baseline to which we can compare the performance of the delegating server. The data generated by the data set generator [14] will be generated with a scale factor of 2. All experiments will use queries generated by the query generator [24]. 1000 queries per template will be generated.

7.1 aQET and aQRT

The questions we want to answer are the following:

1. How does aQET and aQRT-performance differ between the delegating server and the generated server?

2. How do multiple queries in succession effect QET and QRT?

3. If there was an effect found in question 2, does it differ between the delegating and the generated server?

These metrics will be evaluated with the test driver described in section 2.6.6. For each query template, and a given number $n$, we will execute $n$ queries per template with the test driver; after which we will report the averages of the measurements returned by the test driver. Every query template will be tested with $n = 10$, $n = 50$, and $n = 100$. Question 1 will be answered by comparing the results of the tests run on the generated server with the results of the tests run on the delegating server. Question 2 will be answered by, for each server individually, comparing the results of the tests with differing $n$ values. Question 3 will be answered by comparing the results of question 2.
7.2 aTPt

The questions we want to answer are the following:

1. How does aTPt differ between the delegating server and the generated server?

2. How does an increase in load (i.e. an increase in the number of simulated clients) affect TP?

3. If there was an effect found in question 2, does it differ between the delegating and the generated server?

This metric will be evaluated with the test driver described in section 2.6.4. For each query template, and a given number $n$, we will repeatedly query the test driver for a time interval of 30 seconds with $n$ simulated clients; after which we will report the measurements returned by the test driver. Every query template will be tested with $n = 1$, $n = 2$, $n = 5$ and $n = 10$. The tests will be ran 5 times per $n$. Question 1 will be answered by comparing the results of the tests run on the generated server with the results of the tests run on the delegating server. Question 2 will be answered by, for each server individually, comparing the results of the tests with differing $n$ values. Question 3 will be answered by comparing the results of question 2.

7.3 Running the tests

The first step to run the experiments was generating the data with the dataset generator [10]. Afterward, the query generator [21] was used to generate queries. Both test drivers come with installation guides and instructions on how to use them, these were followed to run the tests according to the specification in previous sections. To attempt to make sure that the state of the computer (available memory etc.) was, at the very least, similar for all runs; all programs but the test drivers and servers were shut down during the test runs.
Phase 4

In this phase we document and evaluate the results of Phase 3.

8 Results

8.1 aQTPt

Figure 2: Average throughput (aTQt) of the delegating server and the Hasura server with 1 client at scale factor 2.

Figure 3: Average throughput (aTQt) of the delegating server and the Hasura server with 2 clients at scale factor 2.
Figure 4: Average throughput (aTPt) of the delegating server and the Hasura server with 5 clients at scale factor 2.

Figure 5: Average throughput (aTPt) of the delegating server and the Hasura server with 10 clients at scale factor 2.

The throughput of the delegating server was in the majority of cases significantly worse than the throughput of the Hasura server. The weakest points of the delegating server were queries with potentially large results, where the asked for subfields had a different name in the Hasura server, QT1 (Listing 23) as an example.
The above query returns a faculty member, the university they got their degree from, and graduate students from said university. The Hasura server is, presumably, able to query all students with one single query and instantly return them. The delegating server has to resolve the emailAddress field, for each graduate student, through delegating to the Hasura server as covered in section 4.5. That is where the steep decline in performance occurs. What the Hasura server manages to do in one query, the delegating server has to do in several. This problem is known as the N+1 problem\cite{35} (as there are N queries in addition to the first query). It is shared among the worst performing query templates (QT1, QT4, QT5, QT8, QT9).

The throughput of QT2 (Listing \ref{26}) and QT10 (Listing \ref{24}) indicate that the overhead of schema delegation might not be as steep as the other results suggest. QT10 performs well, in relation to the other query templates, because the schemas are aligned. The code modifies the arguments as they differ in the Hasura server, but thereafter the results of the query do not have to be processed further. In consequence, the only major difference in throughput should be from the overhead of schema delegation.

QT2 performs well, in relation to the other query templates, because of two factors. First, the code modifies the selections of the query, should it ask for
doctoralDegreeObtainers, see Listing 25. Second, the queried fields are aligned. In consequence, the delegating server is only required to query the Hasura server once.

```typescript
let fields = [createField("nr")];

if (selections.includes("doctoralDegreeObtainers")) {
    const s = getSelections(selectionSet, "
        doctoralDegreeObtainers");
    const a = getArguments(selectionSet, "
        doctoralDegreeObtainers");
    let field = createTwoLayeredField("facultiesByDoctoraldegreefrom", [
        createField("nr"),
        ...s,
    ]); modifyArguments(a);
    field.arguments = a;
    fields.push(field);
}
```

Listing 25: Modifying the selectionset before delegation

```typescript
query university_faculty_publications($universityID:ID)
{
    university(nr:$universityID){
        doctoralDegreeObtainers{
            publications{title}
        }
    }
}
```

Listing 26: QT2

QT8 (Listing 27) more or less crashed the delegating server every time. That is due to the limit argument. The limit argument defines the maximum number of graduate students returned by the query. When the limit was high, the server would eventually time out and then stop responding.
The throughput increased quite significantly with the addition of extra clients. The results in Figure 4 and Figure 5 indicate that the throughput will converge after a certain number of clients (QT1, QT2, QT3), or perhaps reach a "sweet spot" and then decline (QT15, QT16). There seems to be no difference in how the number of clients affect the throughput of both servers. Presumably, the increase in throughput of the Hasura server carries over to the delegating server.

8.2 aQRTt and aQETt

QT8 had to be omitted completely from the results due to its exceptionally poor performance, as it rendered the observations unreadable.

Figure 6: Average execution time (aQETt) of 10 consecutive queries per template at scale factor 2.
Figure 7: Average response time (aQRTt) of 10 consecutive queries per template at scale factor 2.

Figure 8: Average execution time (aQETt) of 50 consecutive queries per template at scale factor 2.

Figure 9: Average response time (aQRTt) of 50 consecutive queries per template at scale factor 2.
In regards to execution time, the delegating server seemed to have decent performance. Some query templates had significantly worse performance in the delegating server, but the majority seems to perform similarly to their counterpart in the Hasura server. Looking closely at the observations it seems as though there is a causal relationship in the difference between aQET/aQRT and aQTPt of the two servers, with small differences in execution/response time yielding large differences in throughput. Assuming that is the case, the query templates performing well in the delegating server in terms of aQET/aQRT should do so due to the same reasons as they perform well in terms of aQTPt.

It is unclear whether an increased number of consecutive queries had an impact on aQETt and aQRTt. If it did, it did not differ between the servers. Figure 8 and 9 show better performance than Figure 10 and 11 but considering the possibility of difference in computer state, and that the test results
from Figure 10 and 11 cover more variations of each query template, we refrain from drawing any conclusions.

9 Discussion

Schema delegation in its essence is querying another schema to resolve your own. `graphql-tools` provides resources to help with that, but schema delegation is not synonymous with using `graphql-tools`. There are cases where it might require less knowledge to implement the resolvers through writing manual requests based on what fields are asked for. Perhaps there are cases where this would result in fewer lines of code. Look at the code snippet in Listing 21. The field we are after is the `professor`. If we have not fetched it previously we delegate it to the other server. This is done through several helper functions, the primary one being `graduateStudentByPk` (Listing 22), that in turn use other functions to transform the fields.

A lot of work is done to perform the task of fetching the advisor of a student. This is because the schemas were not ”aligned”, i.e. the names and types of fields were not equal on the different schemas. Because of this, transforms are added to get fields with differing names. Had the schemas been aligned it would be sufficient to delegate to the other schema at the root, but since that is not the case it is required that one resolves the subfields through `delegateToSchema` as well. An alternative without `graphql-tools` is presented in Listing 28. The function `getAdvisorByNr` is not implemented but imagine it being an ordinary GET request [30]. It is not dependent on the `graphql-tools` library, it gives you the benefits of `graphql`, it improves readability, and results in less code.

```javascript
1  advisor: async (parent, args, context, info) => {
2      const advisor = await getAdvisorByNr(parent.nr)
3      /*
4         here you could perform transforms of the advisor
5            field, e.g.
6         return {
7              nr: advisor.id,
8              name: advisor.firstName
9         }
10     */
11     return advisor;
```
One shortcoming is that it is hard to know if there are other schema delegation implementations that would perform better than the one in this project. It is probably safe to say that there are other ways to solve the problems faced in this project, and it would have been of interest to measure the variance in performance of different implementations.

It should be stated that there were limitations due to the assignment at hand, those limitations being unable to rename fields or types contained in the schemas. Had it been allowed to rename the fields, a simple problem as the field email address being written in lower case in one schema and in "camel case" in the other could be solved by simply making them uniform.

It might seem as Listing 25 provides a viable solution to the performance issues of the delegating server, and while the performance increases; it comes with the cost of increased code complexity. The method could possibly be refactored and improved upon, but it would still be significantly more complex than querying the other server as discussed previously in this section. As stated on the graphql-tools website [41], "Delegation is useful when the parent schema shares a significant part of its data model with the subschema". If anything, the results of this project should verify that statement.

All tests were run with a large set of available queries per template. It is worth considering whether one should instead run tests with a smaller set of queries, each chosen based on whether it would actually be similar to queries that are used in "real-world scenarios". To avoid large variance in query performance it is good practice to hand out large sets of data in smaller chunks. If, for example, the client wants to view 1000 students, the server could send them out 10 at a time; allowing the client to go through the set of students 10 at a time. This, presumably, reduces the load on the server while at the same time providing the client with the desired data. With this in mind, one could question the value in evaluating server performance for queries asking for large sets of data, as one could use the previously mentioned technique to avoiding ever serving queries of the kind.

Another reason to use a smaller set of queries is to be able to draw conclusions
regarding the effect that consecutively executed queries have on aQETt and aQRTt. If all aQETt/aQRTt experiments cover a different set of queries it is hard to compare them to each other as, for some query templates individual queries have a large variance in terms of QET and QRT. It might also be the case that it is a better idea to measure aQETq/aQRTq instead if one wants to draw conclusions regarding the effect of consecutive queries.

10 Further research

This project did not use any profiling tools to determine where possible code improvements could be made. It might be of interest to also investigate if there could be improvements in the source code of graphql-tools. Potential improvements could also be found through reaching out to the developers working with graphql-tools, and ask for their feedback.

The cause behind the poor performance of QT1 (Listing 23), described in section 8.1, where the query has to query $N$ additional graduate students if the university has $N$ graduate students; is known as the N+1 problem (as there are $N$ queries in addition to the first query). This problem is shared among the worst performing query templates (QT1, QT4, QT5, QT8, QT9). There are tools available to solve this problem [7] that would instead fetch all graduate students in one single query, reducing the number of queries queried from $N + 1$ to 2. It would be of interest to evaluate how such tools would impact the performance of the delegating server.

When should one use graphql-tools? Using libraries poses the risk of deprecation and an increase in code complexity. Therefore, one should make sure that they know when and why one should use libraries. Researching the performance difference of simply querying the other server "manually", as discussed in Section 9, could help developers to determine whether they believe it is necessary to use a library or not.

As there are a multitude of companies that use GraphQL [10], there might be more techniques available to implement schema delegation. Instead of searching the web for one could ask users about if they use schema delegation and, if that is the case, what techniques they used.
11 Final remarks

This project aimed to find and describe different methods of schema delegation, implement one of them and analyze its performance with the help of the LinGBM benchmark. The purpose was to provide information about the performance and limitations of schema delegation. All things considered, I believe that this project has managed to do so. Hopefully this information could provide value to further research regarding schema delegation.

References


Table 1: Choke points of the LinGBM benchmark and their coverage by the 16 LinGBM query templates.

<table>
<thead>
<tr>
<th></th>
<th>QT: 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attribute Retrieval</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Multi-attribute retrieval</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Relationship Traversal</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Traversal of multiple 1:N relationship types</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Efficient traversal of 1:1 relationship types</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Traversal w/ and w/o retrieval of intermediate object data</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Traversal of relationships that form cycles</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Acyclic relationship traversal that visits objects repeatedly</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ordering and Paging</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Paging without offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Paging with offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Ordering</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Searching and Filtering</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP String matching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Date matching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Subquery-based filtering</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Subquery-based search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Multiple filter conditions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><strong>Aggregation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP Calculation-based aggregation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CP Counting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>