BACHELOR'S THESIS 2022

Performance in Apollo Federation — A Controlled Experiment Evaluating the Effects of Execution Strategies and Number of Subgraphs

Anna Bergvall

ISSN 1651-2197
LU-CS/HBG-EX: 2022-11

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

LTH
FACULTY OF ENGINEERING

Bachelor's Program in Computer Science and Engineering

Performance in Apollo Federation — A Controlled Experiment Evaluating the Effects of Execution Strategies and Number of Subgraphs

A Java Implementation

by

Anna Bergvall, an7151be-s@student.lu.se

EDAL05
Bachelor Thesis
June, 2022
Supervisors: Christian Söderberg (LTH), Anna Karlsson (Capgemini) and Fanny Ejlertsson (Capgemini)
Examiner: Roger Henriksson (LTH)

Abstract

Apollo Federation is an open architecture that allows connecting several different GraphQL APIs to a gateway server, and by doing so creating a unified supergraph API where all data defined in the connected subgraphs are available through one single port. The goal of this thesis was to study performance in Apollo Federation, and more specifically, to what extent performance is affected by, first, the choice of execution strategy in the registered subgraphs, and second, the number of subgraphs connected to the gateway server. To answer these questions, a test application was developed in Java, consisting of three microservices, each of which implemented its own GraphQL schema. The three services where connected to an Apollo Gateway server written in JavaScript.

Thirteen test queries were chosen, based on aspects like breadth, depth, parallelism, and cyclicity. The tests were conducted in Postman on a 2021 MacBook Pro M1 Pro (32 GB) with ten cores with MacOS Monterey 12.3.

The results suggest that the default execution strategy in graphql-java shows the best overall performance. However, the concurrent strategy fetched most bytes per milliseconds in a federated graph with three subgraphs. Moreover, the speed for federated queries were higher than for non-federated queries, which might suggest that distributing entities over several micro services could have positive effects on performance.

No conclusions can be made regarding a relationship between performance in Apollo Federation and the number of subgraphs connected to the gateway server. The results show a peak in latency when querying a subgraph through the gateway instead of querying the GraphQL server directly. However, further research is needed to reach any conclusion on whether the size of the Apollo Federation supergraph affects performance.

Keywords: Declarative Data Fetching, GraphQL, Apollo Federation, Federated Query Processing, Execution Strategies

Acknowledgements

I would like to extend my deepest gratitude to my supervisors at Capgemini, Anna Karlsson and Fanny Ejlertsson, for supporting me throughout this project with their knowledge, experience, and kindness.

I am also extremely grateful to Christian Söderberg at LTH for taking the time to supervise me during this thesis, even though there was no time, and for being encouraging and generous with good advice.

Finally, I would like to give a big thanks to Oskar Prändtare and the entire Java and API team at Capgemini who have helped me a lot during this semester just by being so open and welcoming.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Modern web applications call for a more flexible and rapid development. A monolithic architecture could make it challenging to meet these demands. As a result, the microservice architecture has gained momentum, due to its scalability capabilities. Microservices are "highly maintainable and testable, loosely coupled, independently deployable, organized around business capabilities, and owned by a small team" [4]. In traditional RESTful APIs [5], the type, and thereby the amount of data that is to be returned to the client is decided by the endpoint [6]. As a result, the application tends to become rigid and difficult to maintain over time as requirements change and dependencies grows in number. Moreover, the use of multiple endpoints in RESTful APIs lead to a larger number of server requests [7]. Recently, an alternative to the REST architecture has emerged - GraphQL.

GraphQL is a declarative query language for APIs that is strongly typed and enables inheritance. It is a relatively new technology developed by Facebook and has been used as a data fetching layer in Facebook's mobile applications since 2012 to solve performance issues in mobile clients. In 2015, the GraphQL specification was released open source [8]. In a GraphQL schema, each field is connected to a resolver function, which specifies how the data be fetched. The data is then delivered to the clients as JSON objects.

A benefit of GraphQL over RESTful APIs is that it is possible to fetch all data needed for a web page in one single round trip to the server [9]. Another objective of GraphQL is to solve the problem of over/under fetching, that is, the HTTP response contains either a smaller or larger amount of data than the API consumer needs. However, using a monolithic GraphQL schema takes away some of the flexibility which comes with a microservice architecture. To solve this problem, schema stitching was introduced to, as the name suggests, stitch together several GraphQL schemas. With schema stitching, the subgraphs are unaware of each other and are instead loaded by the gateway which stitch their schemas together [10].

Apollo's implementation of schema stitching is now deprecated and replaced with the open source project Apollo Federation. In Apollo Federation, as opposed to schema stitching, the subgraphs are aware of each other's schemas and the gateway

server is able to stitch them together automatically. The result is that an entity defined in Subgraph A can be referenced from Subgraph B [11]. Apollo Federation enables each team to own and maintain only the GraphQL schema connected to the microservice that they are responsible for. As a result, a higher degree of separation of concerns is possible.

The thesis project is conducted at Capgemini in Malmö. Capgemini offers consulting within IT and management and was founded as Sogeti in 1967 by Serge Kampf in France. In the seventies Capgemini acquired C.A.P. — also a French consulting company — and American Gemini, and the name was changed to Cap Gemini Sogeti. Capgemini has more than 300,000 employees worldwide [12].

1.2 Purpose

There is a body of work [13, 14, 15, 16] written on GraphQL from a performance standpoint. However, no academic work has been published on how performance is affected as subgraphs are composed into a federated architecture with Apollo Federation. The purpose of this study is to evaluate the performance in Apollo Federation with respect to two variables: First — the most fine-grained variable — the execution strategies used in the resolvers, and second, the number of subgraphs making up the supergraph.

1.3 Research Questions

This thesis aims to answer the following research questions:

- **RQ1:** How is the performance in Apollo Federation affected by the choice of execution strategy?
- **RQ2**: To what extent does the number of subgraphs affect performance in Apollo Federation?

1.4 Reasons for Writing the Thesis

As mentioned in the background section, a large share of clients consists of mobile devices. Users expect high responsiveness and speed. In order for companies to make informed decisions as to what API architecture to use, it is important to study the performance of new solutions like Apollo Federation, where multiple GraphQL APIs are composed into a supergraph. This supergraph schema provides looser coupling between front-end and back-end teams as well as increased separation of concerns as each team can own the graph they implement.

The Capgemini team where this thesis project is being conducted is called Java & API and the purpose of the study is to make a contribution to both the academic community but also to provide useful insights that can benefit Capgemini in their continuous effort to provide fast and responsive software to their clients. Furthermore, studying a relatively immature solution like Apollo Federation is interesting since there is a lack of published works on it at the time of writing.

Another reason why it is also important to study GraphQL performance is that it is getting widely adopted by large companies, like Netflix [17], GitHub[18]. It is important to ensure that computing resources are not wasted, given the environmental crisis.

1.5 Method

In order to investigate how the performance in Apollo Federation is affected by the choice of execution strategy and number of subgraphs, a Spring Boot application containing a number of microservices will be developed in Java, each owning their own GraphQL implementation. The Apollo gateway server that will join the subgraphs together will be implemented in JavaScript. The thesis will draw on an example from Apollo Federation's website [19] and call the subgraphs Subgraph A to Subgraph C. The main reason for doing so is that a real life example could contain hundreds of micro services (Netflix alone have more than 1,000 [20]) However, the scope of this project is limited and will not contain entities that reflect real world data, like *User*, or *Review*, for example. Instead, the different GraphQL schemas will be called Subgraph A, Subgraph B, and Subgraph C. This lack of a real world connection will hopefully emphasize the purpose of this study: to examine the performance of a federated graph by varying the above mentioned parameters in a controlled environment. Moreover, there exists a natural mapping between letters and numbers. We know that A is the first letter in the Latin alphabet, B is the second one, and so on. To prevent confusion related to the different parameters, this natural mapping may help clarify the number of subgraphs that make up the supergraph at any given point in the discussion.

When designing the federated architecture, schema analysis and construction will be conducted seamlessly in order to identify connections of varying breadth and depth that will constitute critical items in the experiments. Control items will also be distinguished in this process. To provide GraphQL support in Java, the third-party libraries graphql-java [21] and graphql-java-federation [22] will be used. Furthermore, since it is the performance of Apollo Federation that is being evaluated, no database will be used but the data will be stored in a static list in memory.

1.6 Delimitation

Only performance in an application developed in Java will be investigated. Since it is the performance of the API that is being tested in this thesis, no database will be used. Any latency caused by remote calls would be of no interest for the purpose of studying the performance of the API. Furthermore, only the performance of Query operations will be studied in this thesis.

As mentioned above, the performance aspect that will be measured is that of latency. The reason behind this choice is the high expectations application users have on speed and responsiveness.

1.7 Outline of the Thesis

In the second chapter, relevant theoretical background is presented, including a literary review. Following, in the Method chapter, the methods for conducting this research are revealed, together with an introduction to the developed test application. After that, the results are presented, followed by a short discussion on the results. Finally, in **Chapter 6**, the conclusions will be given.

2

Theory

This section will introduce important concepts and theories that are central to this study. First, an overview of GraphQL will be given, followed by an introduction of Apollo Federation. Finally, relevant works related to this study will be reviewed.

2.1 GraphQL

GraphQL was developed in 2012 by Facebook to solve issues with poor performance in their increasingly complex native mobile applications [8]. While the RESTful view on data focused on resource URLs and SQL statements, the GraphQL specification [23] describe Facebook's need to represent data as graph object. Another argument for moving away from REST was the large amount of code that had to be written to facilitate the data transport between client and server [8].

GraphQL is not a programming languate like Java or Python, but a declarative data-fetching language. *Declarative*, in this context, refers to the fact that developers will specify what data they need to fetch from the backend without having to think about how to get it [24]. Requests are sent in the form of GraphQL queries which structurally resemble the JSON objects that are sent from the GraphQL server in response [8].

Another aspect of GraphQL is that it is not a framework tied to a specific programming language or implementation, but "a specification for client-server communication" [24]. However, the GraphQL specification provides some guidelines [8, 24]:

1. GraphQL's declarative nature results in query responses that have the same shape as their respective requests [8], see **Listing 2.1** and **Listing 2.2**. In many REST APIs the content and form of the response will depend on how various endpoints are implemented [23]. In GraphQL, on the other hand, the client recieves exactly the data it asks for - no more and no less [8, 24]. That is, GraphQL is a solution to the issues of overfetching [6] and underfetching associated with RESTful services discussed in the previous section [15].
2. GraphQL has a **hierarchical nature** and its queries may consist of nested

data fields [24], as illustrated by **Listing 2.1** below. Each field has its own type, e.g., Integer or an Object type.

3. GraphQL is **strongly typed**, which enables descriptive error messages as each query is validated against the GraphQL schema – also the GraphQL type system – before execution. Each level in the GraphQL query corresponds to one type [8]. The notion of type will be further discussed below.
4. **GraphQL does not dictate data storage solution** [8]. GraphQL is "not directly connected to the database" [7], but is merely a specification for data transportation in a client-server solution. Any SQL or NoSQL database can be used to store data in a GraphQL API, as well as a static list in memory (which is the case in this study). [7].
5. It should be possible for the client to query the GraphQL server for what types it supports. This concept is called **Introspection** [8, 24].

Below is an example query that requests a type called `DataType`. In order to fetch the correct instance of DataType, a unique identifier ("primary key" in SQL syntax) has to be attached to the query. In this example, the unique identifier has the value '1'.

Listing 2.1: GraphQL used as a query language

```
Query DataType ("id": 1){
    FirstField
    SecondField
    ThirdFieldWhichReturnsAnotherObjectType {
        NestedField
    }
}
```

A possible response, written in JSON, is available in **Listing 2.2** below. The request and response look similar in their hierarchical and nested structure.

Listing 2.2: An example response

```
"data": {
    "DataType": {
        "FirstField": Some attribute,
        "SecondField": Another attribute,
        "ThirdFieldWhichReturnsAnotherObjectType": {
            "NestedField": Attribute belonging to another
                           Object Type.
        }
    }
}
```

It is up to the designer to decide how to implement a GraphQL API. For example, it is possible to choose a different transport protocol than HTTP. However, HTTP remains the most widely used protocol [24].

Both a Schema Definition Language and a Query Language

There are two types of GraphQL documents – either it is executable or representing a GraphQL type system [23]. Only an `ExecutableDocument` which contains at

least one operation definition, e.g., the query in **Listing 2.1**, can be executed. A `TypeSystemDefinitionOrExtension` document, on the other hand, is parsed by the GraphQL client in order to define the GraphQL schema that queries will be validated against [25]. **Listing 2.3**, which contains the type definition for one of the schemas used in this study, will serve as an introduction to GraphQL as a type definition language. First, the root queries and their return types are defined. Some interesting observations regarding the query definitions can be made. In case the return type is not a list, the query takes an id as parameter. This parameter cannot be null. In fact, no values followed by an exclamation mark can be null. Take the return type for `fetchAllBTypes` for example, "`[BType!]!`" (on line 3): This query must return a list (marked by the bracket parentheses), or it will return an empty list in case there is no data to be fetched. The point is that the query is not allowed to return null. Moreover, the exclamation mark after `BType` on line 3 in **Listing 2.3** adds the constraint that no null objects are allowed in the returned list. The annotation `@key` is an example of a GraphQL *directive*, which provides "additional information to the executor" [23].

Listing 2.3: GraphQL used as a type definition language

```
1  type Query{
2      lookupBType(id: ID!): BType
3      fetchAllBTypes: [BType!]!
4      lookupUnrelatedTypeInSubgraphB(id: ID!):
5          UnrelatedTypeInSubgraphB
6      getAllUnrelatedTypesInSubgraphB: [UnrelatedTypeInSubgraphB!]!
7  }
8
9  type Mutation{
10     createBType(appearsFirstIn: String!, isDefinedIn: String!,
11     relatedATypeId: Int!): BType!
12 }
13
14 type BType @key(fields: "id") {
15     id: ID!
16     appearsFirstIn: String!
17     isDefinedIn: String!
18     relatedObjectInSubgraphA: AType!
19 }
20
21 type UnrelatedTypeInSubgraphB @key(fields: "id") {
22     id: ID!
23     randomWord: String!
24     relatedObjectsOfSameType: [UnrelatedTypeInSubgraphB!]!
25 }
26
27 type AType @key(fields: "id") @extends {
28     id: ID! @external
29     isExtendedIn: String!
30     relatedObjectsInSubgraphB: [BType!]!
31 }
```

Below the query definitions, on line 9, a mutation operation type is defined. It takes three parameters and returns a newly instantiated data object. In SQL syntax, this mutation operation would correspond to an `insert` statement. Then, the next two

type definitions in **Listing 2.3** define the types `BType` and `UnrelatedTypeInSubgraphB`, that is, we define what attributes the two types will have. This operation would in a relational database correspond to the `create table` statement. The *type extension* of `AType` will be discussed in **Chapter 3**.

GraphQL Types

A central concept of GraphQL is that of *types*, which can be divided into two categories: *scalar* and *object* types. Both types populate fields in the GraphQL query and therefore also nodes in the data graph abstraction. However, object types, like `BType` in the schema in **Listing 2.3**, corresponds to intermediate level nodes [23], while scalar types are childless, also known as leaf nodes, in this abstraction.

The built-in scalar types provided by the GraphQL specification are *Int, Float, String, Boolean, and ID*. These types, *ID* excluded, behave similar to built-in types in for example Java and will not be further elaborated on in this section. However, the *ID* type deserves an introduction. It represents a unique identifier and is serialized in the same way as a String, although it is often numeric in practice [23].

When building an application, scalar types do not suffice to represent the complex connections and entities that make up the system. Object types, also called *custom objects*, are closely connected to the application domain [24]. These types often correspond to tables in a relational database.

Another set of types defined in the specification are *introspection types* [23]. These fields begin with two underscores, e.g., `__typename`, and they allow the GraphQL client to query the internal type system of the GraphQL schema. Apart from the types already described in this section, GraphQL also specifies *enumeration types*, *union types*, and *Interface* types. These types, however, are not going to be further explained in this thesis.

Fields and Fragments

An object type definition contains of a list of fields, each one ascribing to the object a distinct quality. A field can consist of scalar types, enumeration types, Interfaces, Unions, or other object types.

GraphQL fragments will not be used in this thesis. However, an introduction to GraphQL should at least mention this possibility. Fragments help to eliminate duplicated code. These are named procedures, or "reusable units" [23]. Instead of repeating sets of fields, this selection set can be used to define a fragment. Then, in the query body, it is possible to write the name of the fragment instead. Fragments loosely correspond to *views* in a relational database.

Operation Types

Three operation types can be defined in a GraphQL schema: Query, Mutation, and Subscription. These operations are also called *root types* [24] since they represent the entry points of execution. The Query operation is used to fetch data from an API – recall the query definitions in **Listing 2.3**.

The declarative nature of GraphQL becomes visible in the Query operation: the client specifies exactly what types and fields it wishes to fetch and receive these fields only, avoiding both over-fetching and under-fetching [23]. A block of fields surrounded by curly brackets are referred to as *selection sets* [24]. A query involving two nested selection sets is given in the following listing:

```
query fetchAllBTypes {
     id
     appearsFirstIn
     relatedObjectInSubgraphA {
         id
         appearsFirstIn
     }
 }
```

Since the field `relatedObjectsInSubgraphA` will return another object type, that field must be followed by another selection set specifying what fields to fetch.

The response will mirror the query and the fields will be fetched with the help of GraphQL resolvers. Again, it is worth mentioning that the GraphQL specification does not dictate what type of data store to use.

2.1.1 Query Execution

An important building block of the GraphQL run-time engine is the *directives*. A common directive is the `@key(id = $id)` marking the uniquely identifying field of a specific type. Its counterpart in a relational database would be the primary key. According to the GraphQL specification, directives can be used to specify "alternate run-time execution and type validation of a GraphQL document" as it provides additional information to the executor. An example of such directives is the `@CacheControl` which affect how caching is performed in the GraphQL service.

Before a request is executed it is validated against the type system, like the one in **Listing 2.3**. However, if a request is known to have been previously validated and has not been changed since, it will be executed without a new validation process taking place. After validation, input arguments like for example `$bTypeID` in **Figure 2.1** are coerced according to the coercion rules of the specific argument type. After all input variables are coerced, the operation is executed.

2.1.2 An Introduction to Execution Strategies in GraphQL Java

The result of executing a `Query` or `Mutation` operation is the same as executing the top selection set of that query. However, the top selection set of the latter should be executed serially to avoid race conditions [23].

Execution of `Query` and `Mutation` types return an unordered map containing data and errors [23]. The default execution strategies in GraphQL-Java are `AsyncExecution-Strategey` for queries and `AsyncSerialExecutionStrategy` for mutations [21].

GraphQL execution strategies can be differentiated based on three characteristics [15]:

Figure 2.1: Example of input arguments with placeholder variable [1]

1. Parallelism – A query execution strategy can be either serial or parallel.
2. The data fields of a GraphQL query can be resolved either synchronously or asynchronously.
3. If batching and/or caching are being used for data access.

Returning Futures

An important concept in asynchronous processing is that of "promises", also known as "futures". A class implementing the `java.util.concurrent.Future` interface represents "a future result of an asynchronous computation" [26]. An implementation of the `Future` interface, including the possibility to handle errors and add logic to the computation, was introduced in Java 8 – `CompletableFuture<T>` [27]. `CompletableFuture<T>` also implements the interface `CompletionStage` which is a stage of the asynchronous computation where computations can be made upon completion, or execution can be handed over another asynchronous process [27]. The class `graphql.execution.ExecutionStrategy` in the GraphQL Java library contains the abstract method `execute(ExecutionContext context, ExecutionParameters parameters)` which returns a `CompletableFuture<ExecutionResult>`.

AsyncExecutionStrategy ("Default")

As already mentioned, `AsyncExecutionStrategy` is by default the execution strategy chosen by GraphQL Java for query operations. `AyncExecutionStrategy` will dispatch each field as a `CompletableFuture<T>`[23], which entails that the executor

does not care which field completes first. According to the GraphQL specification, this is the best performing execution strategy [23].

AsyncSerialExecutionStrategy ("Serial")

As mentioned, `AsyncSerialExecutionStrategy` is the default execution strategy for mutation operations in GraphQL Java. It offers asynchronous execution where one field will be resolved at a time [28].

Batching with DataLoader

In a naive implementation, GraphQL resolvers are unaware of the global query and are focused on transporting the data of the field they are connected to only. The fact that GraphQL uses field specific instead of endpoint specific resolvers to fetch data from the server leads to a dreaded performance issue – the N+1 problem [6] – which refers to the gateway making additional and unnecessary trips to the data source to collect data for nested fields [6]. The N+1 problem is amplified in a GraphQL context compared to a RESTful API, since the cost of executing a query cannot be computed in advance [6], and the reason for this is the fact that GraphQL is declarative.

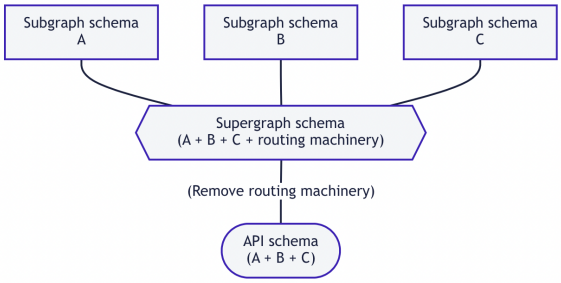
A solution to the N+1 issue, the DataLoader pattern, was originally proposed by Facebook, who released a JavaScript library called `DataLoader` [29]. `DataLoader` uses lazy data loading by letting data fetchers return promises instead of values. In its simplest version, a data fetcher collects the value from the data store directly when it is invoked. This can lead to, like mentioned above, unnecessary round trips and increased latency. When utilizing batching with data loaders, the logic related to accessing the data store is moved to a *batch function*. In each call to the data fetcher, the data loader is loaded with the id of the wanted object. When all relevant ids are loaded, the list or set of loaded ids are sent to to batch function which makes a batched call to the data store. In the process, many separate calls have been replaced by a single one [23].

2.2 Apollo Federation

A monolithic GraphQL server works against the scalability and flexibility required in modern applications [2]. It is a tough task to manage and maintain monolithic graph servers [30], as they easily start smelling of rigidity. According to Endris et al. [31], a federated query processing system "provides a unified access interface to a set of autonomous, distributed, and heterogeneous data sources." That description suits Apollo Federation well. Apollo Federation is an open architecture that allows connecting several different GraphQL APIs to a gateway server, and by doing to creating a unified supergraph API where all data defined in the subgraphs are available through one single port [2], like in **Figure 3.2**. In this way, clients can take advantage of GraphQL's declarative nature while backend developers only need to implement the subgraph related to the microservice they manage. This rather elegant separation of concern is accomplished by the introduction of syntactic directives on top of the standard GraphQL specification.

Central to the Apollo Federation architecture is the notion of type references and type extensions. The directive `@extends` makes it possible to add types that are defined in Subgraph A can be given additional fields in Subgraph B. For example, let us imagine a type `AType` that is defined in Subgraph A. This type has a field `relatedObjectsInSubgraphB`, which returns a list of `BTypes`, a type defined in Subgraph B. Instead of introducing a stronger coupling between the subgraphs by importing the whole BType definition into the AType graph, the AType type in Subgraph A can be extended in Subgraph B. If each microservice is in charge of the types connected to its domain and therefore defined in their GraphQL subgraph, a looser coupling between the microservices making up the application will be maintained. Subgraph A will be oblivious to the new data fields — Subgraph B and the Apollo gateway server are responsible for resolving this data [32].The gateway then uses the supergraph schema to resolve type references and extensions between subgraphs [2]. The supergraph schema is then composed from all the subgraph schemas in the federated architecture.

Figure 2.2: A federated supergraph [2]



At the time of writing, the operation type `Subscription` is not supported by Apollo Federation. However, Kim et al.[33] found that the Subscription type was only present in 20% of GraphQL APIs, with could arguably mitigate this lack of support.

2.2.1 Query Plans in Apollo Federation

In a federated query engine with distributed and heterogeneous data sources, a global strategy is needed to ensure efficient data resolving and to collect the result into a query answer [31]. This strategy is also called *query plan*. The goal of query planning is to translate the global query sent to the gateway server, into several queries to be sent to the individual subgraphs, expressed in non-Apollo Federation syntax [31, 34]. A query plan looks similar to a serialized JSON or GraphQL query and it has a root node of type `QueryPlan` [34].

```
QueryPlan {
    ...
}
```

Each node inside the query plan is either a `Fetch node`, a `Parallell node`, a `Sequence node`, or a `Flatten node`.

Fetch Node

This type appears at least once in every query plan. Operations that only fetches data from one single subgraph contain only one `Fetch node`, since no orchestration between subgraph is needeeds. Each `Fetch node` takes an argument denoting which service to fetch data from.

Listing 2.4: Example of a Fetch node

```
QueryPlan {
  Fetch(service: "ATypeService") {
    {
      fetchAllATypes {
        id
        appearsFirstIn
        isDefinedIn
      }
    }
  },
}
```

Parallel Node

A `Parallel` node contains two or more `Fetch` nodes. Since a `Fetch` node will only resolve data from a single subgraph, the gateway knows that all immediate children of a `Parallel` node can be fetched in parallel.

Sequence Node

As opposed to the `Parallel node`, a `Sequence node` must be fetched serially, since the result of one node depends on the response from the parent node. As an example, consider this query:

Listing 2.5: A query including two parallel selection sets and three separate schemas

```
query Query($lookupBTypeId: ID!) {
  lookupBType(id: $lookupBTypeId) {
    relatedObjectInSubgraphA {
      appearsFirstIn
    }
    relatedObjectsInSubgraphC {
      appearsFirstIn
    }
  }
}

{
  "lookupBTypeId": 1,
}
```

Both fields `relatedObjectInSubgraphA` and `relatedObjectsInSubgraphC` depend on what instance of BType that has been fetched. Therefore, the child of the `QueryPlan node` is a `Sequence node`. The query plan of this query is listed in **Listing 2.6**.

Flatten Node

The final type of node defined in Apollo Federation is the `Flatten node`. It is the child of a `Sequence node` and always contains a `Fetch node` [34]. The data returned from the `Fetch node` is supposed to be merged with that returned by the parent node. In order for the gateway to know where to append this data, the `Flatten node` takes an argument, `path`. An example of a query resulting in a query plan that includes all of the nodes introduced in this section is given in **Listing 2.6**. Note that all three services, `ATypeService`, `BTypeService`, and `CTypeService` are represented, too.

Listing 2.6: Query plan for the query in 2.6

```
QueryPlan {
  Sequence {
    Fetch(service: "BTypeService") {
      {
        lookupBType(id: $lookupBTypeId) {
          relatedObjectInSubgraphA {
            __typename
            id
          }
          __typename
          id
        }
      }
    },
    Parallel {
      Flatten(path: "lookupBType") {
        Fetch(service: "CTypeService") {
          {
            ... on BType {
              __typename
              id
            }
          } =>
          {
            ... on BType {
              relatedObjectsInSubgraphC {
                appearsFirstIn
              }
            }
          }
        },
      },
      Flatten(path: "lookupBType.relatedObjectInSubgraphA") {
        Fetch(service: "ATypeService") {
          {
            ... on AType {
              __typename
              id
            }
          } =>
          {
            ... on AType {
              appearsFirstIn
            }
          }
```

```
46          },
47        },
48      },
49    },
50  }
```

In the `Flatten node`, two GraphQL fragments are used, separated by `=>`. According to the Apollo Federation specification [34], the first fragment is a *representation* of `AType` and `BType` respectively, while the second fragment contains the fields that the data fetchers in `CTypeService` (the first `Flatten node`) and `ATypeService`(the second `Flatten node`) need to resolve.

2.3 Related Works

In this section, an overview of related works on GraphQL performance will be given.

The advantages that GraphQL offers in terms of declarative data fetching have spawned a discussion about whether GraphQL is "the new REST". However, works on migrating from RESTful services to GraphQL have presented drawbacks related to performance [35]. There is an intrinsic flaw connected to letting the client ask for the exactly the data they want: What if they write unreasonably expensive queries? In fact, it has been proved that even simple queries may return "prohibitively large" responses [36]. Cha et al. also points at this problem in their paper "A principled approach to GraphQL query cost analysis" [14]. The proposed solution is a GraphQL query analysis with the purpose of measuring the cost of a query without executing it [14].

GraphQL performance issues are explored from another angle in Roksela et al. [15]. In their paper "Evaluating Execution Strategies of GraphQL Queries" it is reported that the choice of execution strategy affects GraphQL performance[15]. Roksela et al. [15] evaluated GraphQL execution strategies in the most popular GraphQL implementation written in Python, `graphene`. The strategies evaluated were `serial synchronous` ("serial"), `standard asynchronous` ("async"), `batched asynchronous` ("batched"), and `cached asynchronous` ("cached"). These strategies were evaluated in the light of their resistance to two problems related to GraphQL — the N+1 problem and cyclic requests, where the same node is fetched more than once. The findings in [15] suggest that the positive effect of batching in query execution grew as the query depth increased. The authors also found that when used on cyclic queries, caching offered the largest improvement of any execution strategy, especially when used together with batching. However, caching affected performance in non-cyclic queries negatively [15].

When dealing with cyclic queries, the choice of execution strategy has a significant impact on latency [15]. Roksela et al. found that the response time for cyclic queries were seven times shorter when batching was used as opposed to a plain synchronous strategy [15]. Roksela et al. also point out the favorable traits of programming languages or libraries that offer support for asynchronous data fetching, and give as examples Python's `asynchio` library and the component `AsyncioExecutor` in `python-graphene` [15]. The authors conclude their paper with the statement that

it is important to study even more execution strategies as Apollo Federation is emerging, "where there is a need for global execution strategy" [15].

Cederlund [13] evaluated two GraphQL frameworks, Netflix's Falcor and Facebook's Relay + GraphQL, in terms of performance, response size, and number of service requests, using REST as the baseline. The study suggested that Falcor resulted in longer response times in most test cases [13]. In the case of Relay + GraphQL, however, Cederlund reported a decreased response time for parallel and sequential queries.

According to Endris et al. [31], a federated query engine is in charge of transforming a query directed to and validated against the global schema, into "an equivalent query expressed in the schema of the data sources, i.e., local query." [31]

Endris et al. [31], point out four conceptual difficulties in federated query processing: (i) data source description, (ii) query decomposition and source selection, (iii) query planning and optimization, and (iv) query execution. (i), (iii), and (iv) are also relevant for non-federated GraphQL services. However, as stated in Roksela et al., it is of importance to study how execution strategies affect performance in federated query engines like Apollo Federation[15].

At the time of writing, there are close to none academic texts published on Apollo Federation. A Java implementation of Apollo Federation in Spring Boot using third-party libraries GraphQL Java [21] and GraphQL Java Federation [37] was made in Karlsson [38]. Karlsson, in turn, built upon [22]. This thesis uses [38] as a starting point when developing the test environment. More about this in **Chapter 3** – Method.

3

Method

In this section, the methodology behind this study will be explained. First, a brief discussion on the phases of the project followed by an overview of the research methods and research approach used. Finally, a description of the developed application and test environment will be given.

3.1 Research Methodology

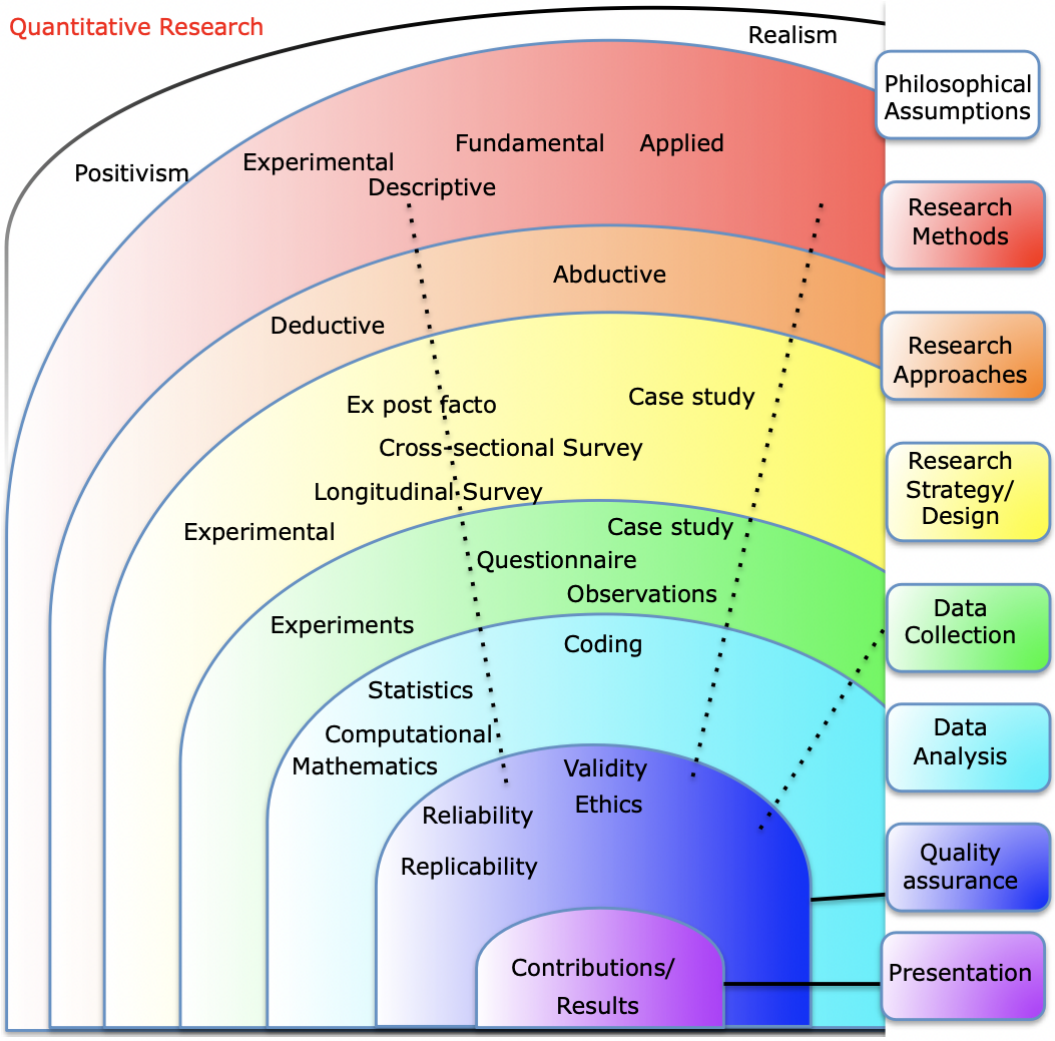
This study was divided into separate phases. Initially, a literary study was conducted to gather enough knowledge of the subject to enable identifying problems that could possibly be solved within the scope of this thesis. This phase led to the formation of the research questions:

1. How is the performance in Apollo Federation affected by the choice of execution strategy?
2. To what extent does the number of subgraphs affect performance in Apollo Federation?

According to [3], research methods "provide procedures for accomplishing research tasks". These questions are the research tasks that was solved by applying experimental research methods. These methods are often used when investigating performance [3]. The research approach used in this thesis was mainly deductive as the answer to these questions would be generalizations that rely on the collected data [3]. Furthermore, an experimental research strategy was used in this study. According to Håkansson, the goal of the research strategy "must be that the strategy leads to "correct, valid, and reliable results" [3]. By controlling the test environment and its variables by altering one variable at the time, the likelihood to achieve reliable results increases [3]. Moving on, the method by which the research data was collected in this thesis was through conducting experiments. These experiments were built by writing a test program in Java.

For this particular study, statistics will be used to analyse the collected data and to evaluate the results. Data analysis methods are used to draw conclusions from gathered data through "inspecting, cleaning, transforming, and modelling data" [3]. Håkansson also points outs that in the case of quantitative research, statistics and

Figure 3.1: The quantitative side of the methodology portal described by Anne Håkansson [3]



computational Mathematics are widely [3]. This connection is further illustrated in **Figure 3.1**.

In order to validate the results in this study, the experiment design must be designed in such a way that that it really measures the performance in a federated graph. Another aspect of quality assurance is *reliability*. In this study, reliability is connected to the consistency of the measurements produced by the test environment. Finally, all steps in conducting this study was included in the final report to ensure that the research can be replicated by others [3].

During the initial process of this thesis, a Gantt schedule was made to function as a guideline for the different phases. The actual work was roughly two weeks behind the Gantt schedule, leading to the final presentation being slightly delayed. In addition, the original Gantt schedule anticipated a shorter period of time with combined development and report writing. In fact, after the initial literary study, development and report writing was being conducted alternately until the final test phase.

3.2 Experiment Design

In order to measure the performance of execution strategies in GraphQL, three Spring Boot applications were developed using graphgl-java-kickstart [39] and graphql-java-federation [37].

The project in [38] was used as a starting point in the development phase. First, the JPA dependency was bypassed and the MySQL fields in the `application.properties` file were removed. The reason for removing the database was to conduct initial tests with static data only. Later during the project, a decision was made to keep the static data store, since round trips to a database server would add more variables to control in a test setting. Second, the services `moon_service` and `planet_service` in Karlsson's project were changed to `ATypeService` and `BTypeService`, according to the example in figure 3.2 [2]. The goal was to have a gateway exposing the subgraphs A + B + C + ... + G as a unified API. Furthermore, in order to output the query plans executed by the Apollo Gateway, the function `serializeQueryPlan` had to be imported from the `@apollo/query-planner` library.

3.2.1 Libraries and Frameworks Used

Here, an introduction to libraries and frameworks used in this study will be given. First, the Java Spring Boot framework will be discussed briefly, followed by graphql-java-kickstart/graphql-spring-boot. Finally, GraphQL Java Federation and @Apollo-server will be mentioned.

Java Spring Boot

The microservices were developed in Spring Boot, which is a Java framework for quickly getting applications up and running with the help of dependency injection – a designated embedded tomcat server, for example [40]. Dependency injection, also called Inversion of Control (IoC) makes it possible for application objects to state what other objects it need to work with, without actually instantiating the objects in the source code. The Spring IoC Container instead injects these dependencies as the object, or *the bean*, is created [41].

In Spring Boot, Spring annotations are used in order to find and pick up every component needed to configure the GraphQL server. Methods and constructors that needs to be found and wired into the framework are annotated with `@Bean`. According to Spring, "a bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container" [42]. IoC stands for Inversion of Control, which is the same thing as *dependency injection*To tell the framework that a class contains at least one bean definition, a class is annotated with `@Configuration`. Then, when the `JavaConfig` class sees a bean annotation, it executes the method and saves the response as a bean in its "bean factory". The annotation `@Autowired` is used on class properties to avoid setter methods [43]. This means that one does not have to use the `new` keyword to instantiate the given property, which is handy in complex applications. The `@Component` annotation appears on every class that is a bean. Say that a class has a property `ATypeService` and that this property in annotated with `Autowired`. The class `ATypeService` must then be preceded by a `@Component` annotation, otherwise the Spring framework will not know how to instantiate the

property `ATypeService` "just in time". Examples of this usage will be given in **Section 3.2.2**.

graphql-java-kickstart/graphql-spring-boot

GraphQL Spring Boot is a third part library built on GraphQL Java, the most used GraphQL implementation in Java. Being a part of Spring Boot, GraphQL Spring Boot utilizes Java beans to find all components in order to configure the server.

A central concept of a GraphQL Java server is that of the data fetcher [21]. Sometimes data fetchers are referred to as "resolvers" and vice versa but they are the same thing. In this thesis, the name *data fetcher* will be used consistently to denote the concept of a method that populates a GraphQL field with data, to avoid additional confusion.

Each field in the GraphQL schema is mapped to one `DataFetcher`. `DataFetcher` is a functional interface with only one method, which has the following signature:

```
T get(DataFetchingEnvironment environment)
```

This method is the key to changing execution strategies, and an explanation as to how will follow shortly. The return type `T` is the fetched value. The parameter, `DataFetchingEnvironment` is an object which contains all the information needed in order to perform a fetch operation. The method `getSource()`, for example, returns the parent node or the field that is about to be fetched.

The types in the static schema definition, like the one in **Listing 2.3**, and the data fetchers are wired together with the help of a class called `RuntimeWiring`. The RuntimeWiring follows the builder pattern and is needed to make the schema executable [44]. The RuntimeWirings for this project will be presented in the following sections.

When explaining data fetching in GraphQL, it is convenient to think about data fields, objects, and scalar types in the terms of recursive tree processes. First, an object type can be represented by a node. Then, each node has one or more outbound edges, which represent the data fields (comparable to *column names* in relational databases) of that object. Just like a data field can return an object type or a scalar type, the edges lead to either nodes or leafs.

```
1   query {
2       fetchAllATypes {
3           id
4           isDefinedIn
5           relatedObjectsInSubgraphB {
6               id
7               isDefinedIn
8           }
9       }
10  }
```

A DataFetcher call can be viewed as an operation which explores a single outbound edge. Each root node has its own DataFetcher. In this case it is called `FetchAllATypesDataFetcher` and returns a list of every `AType` in the data store.

The return type is not a leaf node, but a node with three outbound edges. Consequently, three new `DataFetchers` are fired. `DataFetcher1` follows the edge called "id", while `DataFetcher2` follows the edge called "isDefinedIn". Both these edges lead to leaf nodes and thus, the base case is reached and no more DataFetchers are fired. However, `DataFetcher3` explores the edge called "relatedObjectsInSubgraphB", which returns a node with two outbound edges, "id" and "isDefinedIn". Since both these edges lead to leaf nodes, the chain of DataFetcher calls ends.

Finally, the GraphQL Java documentation [21] stress the significance that the actual source of the data is not dictated by the GraphQL specification. For example, the data can be retrieved from a database or a REST service.

GraphQL Java Federation

The third party library GraphQL Java Federation makes it possible to connect multiple GraphQL subgraphs according to the Apollo Federation specification [37]. A `FederatedEntityResolver` is needed if a type defined in one subgraph is to be extended in another. It is also mandatory to use `RuntimeWiring` to connect the non-scalar fields of the schema to their designated DataFetchers. The different RuntimeWiring instances of this project can be found in **Section 3.2.2**. **Listing 3.1** shows Kudryashov's example code for defining a FederatedEntityResolver and creating a transformed `GraphQLSchema`:

Listing 3.1: Creating a transformed GraphQLSchema using the library GraphQL Java Federation [37].

```
List<FederatedEntityResolver<?, ?>> entityResolvers = List.of(
        new FederatedEntityResolver<Long, LongEntityDummy>("LongEntityDummy", id -> new
    LongEntityDummy(id, "qwerty")) {
        }
);

GraphQLSchema transformed = new FederatedSchemaBuilder()
        .schemaInputStream(getResourceAsStream("entity-schema.graphqls"))
        .runtimeWiring(RuntimeWiring.newRuntimeWiring().build())
        .federatedEntitiesResolvers(entityResolvers)
        .build();
```

ApolloServer and @Apollo/gateway

The `@Apollo/gateway` library is built on the functionality of `ApolloServer`, an open-source GraphQL server written in JavaScript. However, `@Apollo/gateway` adds functionality that allows the server to work as a gateway in Apollo Federation [38, 45]. The `ApolloServer` constructor takes the `ApolloGateway` instance as a parameter in its constructor [45].

To connect subgraphs to the gateway server, a `serviceList` is provided to the `ApolloGateway` constructor. The `serviceList` is an array of objects that specify the name and url of a subgraph in the federated graph, see **Listing 3.10** [45]. On startup, the federated schema is composed by the gateway sending introspection queries to the subgraphs in the service list. However, `serviceList` is now deprecated and replaced by `IntrospectAndCompose`, which works in a similar way [45]. `serviceList` was used in this project since, as mentioned in **Chapter 2**, it was used

in the server that was the starting point for this work [38]. Due to time constraints, it was not replaced by `IntrospectAndCompose`.

3.2.2 The Test Application

The development of the test application was a central part of this study. For this reason, it is also an essential part of the report. The extensive documentation is included in order to facilitate further development of the program.

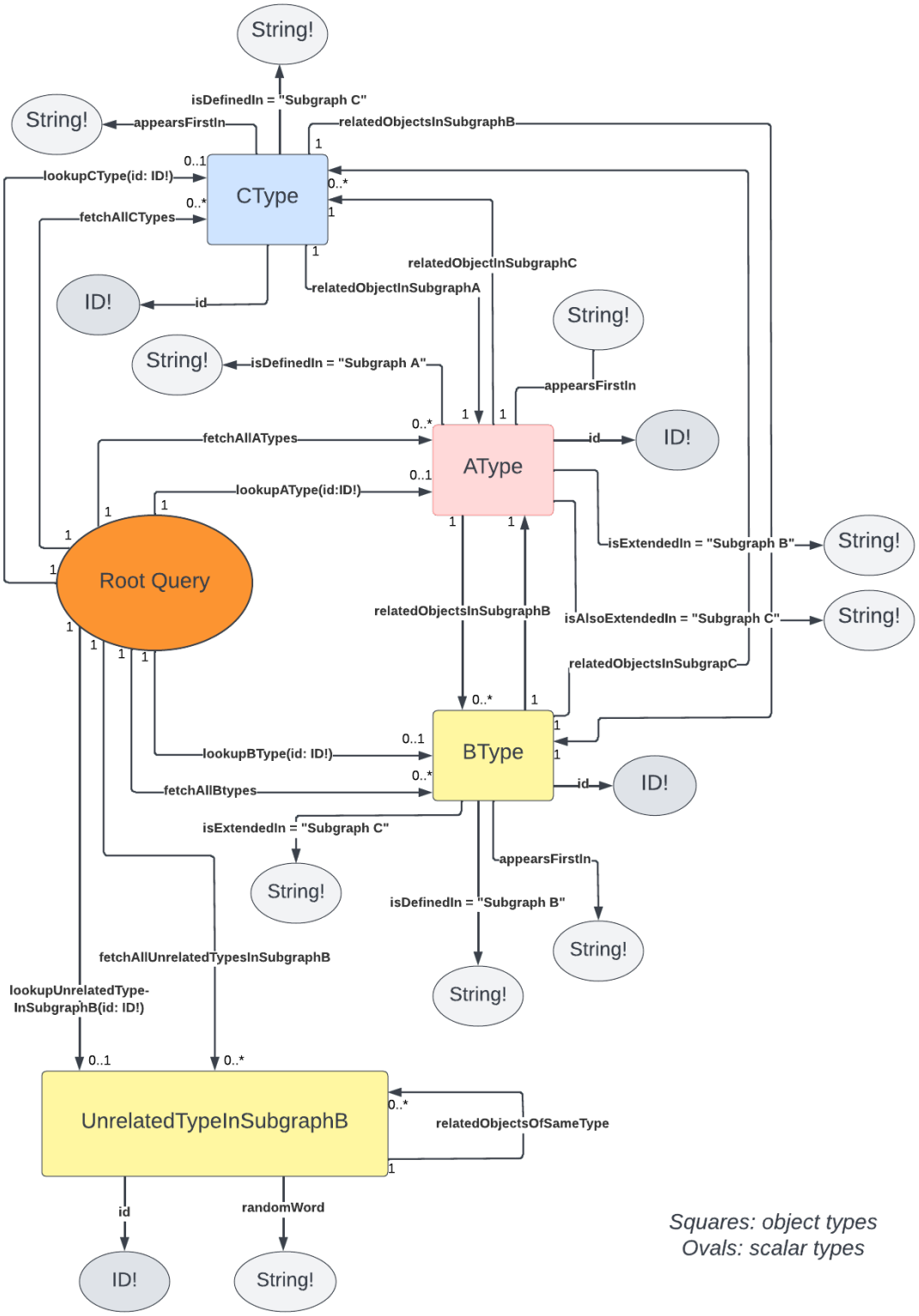
In order to collect data to analyze, three microservices were developed in Java, `ATypeService`, `BTypeService`, and `CTypeService`. These services were connected in an Apollo Server. Like mentioned earlier, the test applications build upon the work of Karlsson [38]. However, this work focuses on different execution strategies and their performance in a federated architecture with the goal of answering the research questions formulated in the introduction but repeated here for clarity:

- **RQ1:** How is the performance in Apollo Federation affected by the choice of execution strategy?
- **RQ2**: To what extent does the number of subgraphs affect performance in Apollo Federation?

To answer R1, it is crucial to be able to switch execution strategies, while data and services remain the same. To answer R2, three micro services were developed to enable running the same queries with either one, two, or three subgraphs connected to the gateway server in order to detect any relationship between latency and number of subgraphs.

This section will dive deeper into the implementation of the three microservices that when plugged into the Apollo gateway server, form a unified GraphQL interface. First, the schema of the subgraph will be introduced, including the types it defines and extends. Then, the data fetcher for each custom type will be presented. The final federated graph is presented in **Figure 3.2**, and may serve as a guide when reading this chapter.

Figure 3.2: The federated graph developed in this study



Subgraph A/ATypeService

The source code for the GraphQL schema in **Listing 2.3** object type is defined in Subgraph A, `AType`. [1] Two Query operations are defined, `lookupAType` and `fetchAllATypes`, which returns an AType and a list of ATypes respectively. A Mutation operation is also defined, which returns an AType.

Two data fetchers were created for Subgraph A, one for each query. The classes LookupATypesDataFetcher.java in **Listing 3.2** and FetchAllATypesDataFetcher.java in **Listing 3.3**both have the same two annotations. The first annotation, `@Component`, indicate that the class is a bean, and the second annotation, `@Slf4j`, stands for "Simple Logging Facade 4 Java" [46] and was used in the project to follow the flow of execution.

Some readers may react to the return types of the fetchers. When one would expect them to return ATypes they instead return *ATypeDTOs*. This deserves a brief explanation. A DTO (Data Transfer Object) is a pattern which can be used to "batch up what would be multiple remote calls into a single call" [47]. One call is cheaper than multiple. Related to the subject of DTOs is the property `ATypeConverter`. Its role is to convert plain `AType` objects to `ATypeDTO` objects. In the `get` method, the `ATypeService` is asked to return every `AType` that is present in the data store, which is in this project a static list in memory.

Listing 3.2: LookupATypeDataFetcher.java

```
@Slf4j
@Component
public class LookupATypeDataFetcher implements DataFetcher<ATypeDTO> {

    @Autowired
    private ATypeConverter aTypeConverter;

    @Autowired
    private ATypeService aTypeService;

    @Override
    public ATypeDTO get(DataFetchingEnvironment environment) throws Exception {
        log.info("Fetching AType data from Subgraph A");
        String id = environment
            .getArgument("id");
        AType aType = aTypeService
            .getATypeById(Integer.parseInt(id));
        return aTypeConverter.apply(aType);
    }
}
```

Listing 3.3: FetchAllATypesDataFetcher.java

```
@Slf4j
@Component
public class FetchAllATypesDataFetcher implements DataFetcher<List<ATypeDto>> {

```

[1] The idea was first to name the type "A", but that name affected the readability of the program negatively as the camel-case naming conventions usually followed in Java could not be followed. Another reason for settling on the name "AType" was to stress the fact that it denotes a *type*. The same reasoning goes for types defined in Subgraph B and Subgraph C.

```
    @Autowired
    private ATypeConverter aTypeConverter;

    @Autowired
    private ATypeService aTypeService;

    @Override
    public List<ATypeDto> get(DataFetchingEnvironment environment) throws Exception {
        log.info("Fetching all ATypes from Subgraph A");
        List<AType> allATypes = aTypeService.getAllATypes();
        return allATypes.stream()
            .map(u -> aTypeConverter.apply(u))
            .collect(Collectors.toList());
    }
}
```

The relevant code in the `@Configuration` class `GraphQLFactoryAsyncSerialExecutionStrategy` is shared in **Listing 3.4**. This code is similar for every subgraph – the main difference is the `RuntimeWiring`. The reason for this is that, as mentioned before, the `RuntimeWiring` turns the GraphQL schema file into an executable document by connecting the object types and scalars of a schema to their designated data fetchers.

Listing 3.4: GraphQLFactoryAsyncSerialExecutionStrategy.java

```
        @Value("classpath:/schemaSubgraphA.graphqls")
        private Resource resource;

        /* This bean is used to choose execution trategies. The three parameters
        represent execution strategies for Query Operation, Mutation Operation,
        and Subscription Operation. If null, default ExecutionStrategy
        will be used. */
        @Bean
        public ExecutionStrategyProvider executionStrategyProvider() {
            return new DefaultExecutionStrategyProvider(
                new AsyncSerialExecutionStrategy(),
                null,
                null
            );
        }

        @Bean
        public GraphQLSchema graphQL() throws IOException {
            InputStream inputStream = resource.getInputStream();

            log.info("Starting Subgraph A with execution strategy
                     AsyncSericalExecutionStrategy.");

            GraphQLSchema transformedGraphQLSchema = new FederatedSchemaBuilder()
                    .schemaInputStream(inputStream)
                    .runtimeWiring(createRuntimeWiring())
                    .excludeSubscriptionsFromApolloSdl(true)
                    .federatedEntitiesResolvers(federatedEntityResolverFactory.create())
                    .build();

            return GraphQLSchema.newSchema(transformedGraphQLSchema)
                    .build();
        }
```

```
34
35          /*
36          This method is needed to turn the GraphQL schema into an executable
37          document
38           */
39          private RuntimeWiring createRuntimeWiring() {
40              return RuntimeWiring.newRuntimeWiring()
41                      .type("Query", builder ->
42                              builder
43                                      .dataFetcher("lookupAType",
44                                          lookupATypeDataFetcher)
45                                      .dataFetcher("fetchAllATypes",
46                                          fetchAllATypesDataFetcher)
47                      )
48                      .type("Mutation", builder ->
49                              builder
50                                      .dataFetcher("createAType",
51                                          createATypeDataFetcher)
52                      )
53                      .build();
54          }
```

To enable the type `AType` to be extended in another subgraph, it is mandatory to provide a `FederatedEntityResolver`. Otherwise the fields defined in Subgraph A will not be accessible through the gateway server. To make the code in **Listing 3.4** slightly less unreadable, the registration of federated entity resolvers was moved to its own factory class, `com.example.demo.federation.FederatedEntityResolverFactory`.

Subgraph A was made available on port 8080 by adding `server.port:8080` to the `application.properties` file, available in **Appendix B**.

Subgraph B/BTypeService

The schema for Subgraph B served as an example when introducing GraphQL as a Schema Definition Language (**Listing 2.3**), but will be repeated here for clarity. However, one type definition that was not discussed earlier was that of `AType`. In Subgraph B, `AType` is extended and is given two additional data fields, with data residing in the `BTypeService`. The entity `AType` is unaware of its new fields, ”isExtendedIn” and ”relatedObjectsInSubgraphB”.

Listing 3.5: Schema definition for Subgraph B

```
type Query{
    lookupBType(id: ID!): BType
    fetchAllBTypes: [BType!]!
    lookupUnrelatedTypeInSubgraphB(id: ID!): UnrelatedTypeInSubgraphB
    fetchAllUnrelatedTypesInSubgraphB: [UnrelatedTypeInSubgraphB!]!
}
type Mutation{
    createBType(appearsFirstIn: String!, isDefinedIn: String!,
    relatedATypeId: Int!): BType!
}
type BType @key(fields: "id") {
    id: ID!
    appearsFirstIn: String!
    isDefinedIn: String!
```

```
    relatedObjectInSubgraphA: AType!
}
type UnrelatedTypeInSubgraphB @key(fields: "id") {
    id: ID!
    randomWord: String!
    relatedObjectsOfSameType: [UnrelatedTypeInSubgraphB!]!
}
type AType @key(fields: "id") @extends {
    id: ID! @external
    isExtendedIn: String!
    relatedObjectsInSubgraphB: [BType!]!
}
```

Note that the "id" in `AType` is followed by the annotation `@external` which is needed in order for the execution engine to know that this field is defined and lives in another service [38]. .

When compared to Subgraph A, a few more data fetchers are needed to populate the fields of this schema:

1. `BTypeDataFetcher`
2. `BTypesDataFetcher`
3. `UnrelatedTypeInSubgraphBDataFetcher`
4. `UnrelatedTypesInSubgraphBDataFetcher`
5. `RelatedObjectsOfSameTypeDataFetcher`
6. `CreateBTypeDataFetcher`

The second data fetcher in this list deserves a comment. The return type is a list of `BTypes`. However, if the parent node is the root query `fetchAllBTypes`, then every `BType` is returned. If, on the other hand, the parent node is an `ATypeDTO`, then the `BTypes` that are related to this `AType` are returned. The source code for `BTypesDataFetcher` is found in **Listing 3.6**. The other data fetchers are available in **Appendix B**.

Listing 3.6: BTypesDataFetcher.java

```
@Slf4j
@Component
public class BTypesDataFetcher implements DataFetcher<List<BTypeDto>> {

    @Autowired
    private BTypeConverter bTypeConverter;
    @Autowired
    private BTypeService bTypeService;

    @Override
    public List<BTypeDto> get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception
     {
        List<BType> bTypes;
        if (dataFetchingEnvironment.getSource() instanceof ATypeDto) {
            bTypes = bTypeService
                .getRelatedBTypesById(((ATypeDto) dataFetchingEnvironment
                    .getSource())
```

```
                    .getId());
        } else {
            log.info("Fetching BTypes from SubgraphB at {}",
                new Date(System.currentTimeMillis()));
            bTypes = bTypeService.getAllBTypes();
        }
        return bTypes == null ? new ArrayList<>() : bTypes.stream()
                .map(bType -> bTypeConverter.apply(bType))
                .collect(Collectors.toList());

    }
}
```

The reason for returning an empty list if there are no related `BTypes` is because neither the `AType` field "relatedObjectsInSubgraphB" or the root query accept null objects in return.

As to the GraphQL configuration class in Subgraph B, most elements are identical to the source code in **Listing 3.4**. However, the private method for supplying the GraphQL engine with a `RuntimeWiring` for Subgraph B contains can be found in **Listing 3.7**. All data fetchers are annotated `@Autowired` in the surrounding scope, and will therefore be instantiated automatically by the Spring Boot framework as singleton objects.

Listing 3.7: Method for creating the RuntimeWiring needed to make the schema executable

```
    private RuntimeWiring createRuntimeWiring(){
        return RuntimeWiring.newRuntimeWiring()
                .type("Query", builder ->
                        builder
                                .dataFetcher("lookupBType",
                                       bTypeDataFetcher)
                                .dataFetcher("fetchAllBTypes",
                                        bTypesDataFetcher)
                                .dataFetcher("lookupUnrelatedTypeInSubgraphB",
                                        unrelatedTypeInSubgraphBDataFetcher)
                                .dataFetcher("fetchAllUnrelatedTypesInSubgraphB",
                                        unrelatedTypesInSubgraphBDataFetcher)
                )
                .type("Mutation", builder ->
                        builder.dataFetcher("createBType", createBTypeDataFetcher)
                )
                /*
                These wirings are needed if there are fields in the types that are
                not scalars but object types.
                */
                .type("BType", builder ->
                        builder
                                .dataFetcher("relatedObjectInSubgraphA",
                                    aTypeDataFetcher)
                )
                .type("AType", builder ->
                        builder
                                .dataFetcher("relatedObjectsInSubgraphB",
                                    bTypesDataFetcher)
                )
                .type("UnrelatedTypeInSubgraphB", builder ->
                        builder
```

```
33                                  .dataFetcher("relatedObjectsOfSameType",
34                                      relatedObjectsOfSameTypeDataFetcher)
35                  )
36                  .build();
37      }
```

Like suggested by the comment starting on line 18, the lower wiring invocations are required when some queries return object types that contain at least one data field that return a non-scalar type. Recall that the execution stops after a scalar node is reach, as it corresponds to a leaf node in a graph abstraction.

Subgraph B was made available on port 8081, by setting the `port` variable in the `application.properties` file, which can be found in **Appendix B**.

Subgraph C/CTypeService

To be able to study to what extent the number of micro services affects performance in Apollo Federation, another micro service was built, `CTypeService`. Its is defined according to the following listing:

Listing 3.8: schemaSubgraphC.graphqls

```
1  type Query{
2      lookupCType(id: ID!): CType
3      fetchAllCTypes: [CType!]!
4  }
5
6  type Mutation{
7      createCType(appearsFirstIn: String!, isDefinedIn: String!, relatedATypeId: Int!): CType!
8  }
9
10 type CType @key(fields: "id") {
11     id: ID!
12     appearsFirstIn: String!
13     isDefinedIn: String!
14     relatedObjectInSubgraphB: BType
15     relatedObjectInSubgraphA: AType
16 }
17
18 type AType @key(fields: "id") @extends {
19     id: ID! @external
20     isExtendedAlsoIn: String!
21     relatedObjectInSubgraphC: CType
22 }
23
24 type BType @key(fields: "id") @extends {
25     id: ID! @external
26     isExtendedIn: String!
27     relatedObjectsInSubgraphC: [CType!]!
28 }
```

Both types from previous subgraphs are extended in this schema, `AType` on line 18 and `BType` on line 24. This means that `AType` is extended in two other schemas without that affecting the maintenance of that micro service.

The data fetchers built in this micro service are:

1. `ATypeDatafetcher`
2. `BTypeDatafetcher`
3. `CTypeDatafetcher`
4. `CTypesDatafetcher`
5. `CreateCTypeDatafetcher`

The source code for these data fetchers can be found in **Appendix B**. However, the method returning the `RuntimeWiring` for Subgraph C is given below:

Listing 3.9: Method returning the RuntimeWiring for Subgraph C

```
private RuntimeWiring createRuntimeWiring() {
        return RuntimeWiring.newRuntimeWiring()
                .type("Query", builder ->
                        builder
                                .dataFetcher("lookupCType",
                                        bTypeDataFetcher)
                                .dataFetcher("fetchAllCTypes",
                                        cTypesDataFetcher)
                )
                .type("Mutation", builder ->
                        builder.dataFetcher("createCType", createCTypeDataFetcher)
                )
               // Each CType has 0 or 1 related BType. The BType does not know this so
               // both directions of this relationship must be resolved in Subgraph C
                .type("CType", builder ->
                        builder
                                .dataFetcher("relatedObjectInSubgraphB",
                                        bTypeDataFetcher)
                )
                // Each CType is related to one AType
                .type("CType", builder ->
                        builder
                                .dataFetcher("relatedObjectInSubgraphA", aTypeDataFetcher)
                )
                // Each BType is related to a list of CTypes.
                .type("BType", builder ->
                        builder
                                .dataFetcher("relatedObjectsInSubgraphC",
                                        cTypesDataFetcher)
                )
                // Each AType is related to none or one CType
                .type("AType", builder ->
                        builder
                                .dataFetcher("relatedObjectInSubgraphC",
                                        cTypeDataFetcher)
                )
                .build();
    }
```

The Gateway Server

The code for the gateway server was written in JavaScript and is almost identical to the server in [38]. Some elements have been added, however. Open Telemetry

libraries have been added to provide tracing data for the gateway. Tracing was disabled during testing since it caused high latency.

The source code for the server is available in B. However, the lines where the three subgraphs A, B, and C are connected to the gateway is shown below:

Listing 3.10: Apollo gateway constructor

```
1 const gateway = new ApolloGateway({
2     serviceList: [
3         { name: "ATypeService", url: "http://localhost:8080/graphql" },
4         { name: "BTypeService", url: "http://localhost:8081/graphql" },
5         { name: "CTypeService", url: "http://localhost:8082/graphql" }
6     ],
7 });
```

To detach subgraphs when investigating R2, *To what extent does the number of subgraphs affect performance in Apollo Federation?*, line four and/or line five were commented out.

3.2.3 The Execution Strategies Evaluated in This Study

In this section, the execution strategies tested in this thesis will be explained in further detail. The initial idea was to replicate Roksela et al. [15] and see if the results for execution strategies in Java would be similar to those in Python. Unfortunately, due to a lack of time, it was not possible to implement batching an all the data fetchers. As a result, no comparisons can be made to the findings in Roksela et al. regarding batched or chached strategies. Moreover, it proved difficult to simulate synchronous execution as the `graphql.servlet.async-mode-enabled` flag in `application.properties` would not work. Therefore, the synchronous serial execution strategy in [15] could not be reproduced. The execution strategies that was evaluated in this study was instead `AsyncExecutionStrategy`, `AsyncSerialExecutionStrategy`, and a concurrent execution strategy, achieved by wrapping plain data fetchers in asynchronous data fetchers by calling the static method `AsyncDataFetcher.supplyAsync(wrapped dataFetcher)`.

One of the queries run in the tests will serve as an example when walking through the different ways data can be fetched by changing execution strategy: `B || B`. The name means that two types defined in Subgraph B are fetched in parallel. The reason that this query makes a particularly good example is that the root queries `fetchAllUnrelatedTypesInSubgraphB` and `fetchAllBTypes` are independent of each other and can be fetched in parallel.

Listing 3.11: Query "B ‖B"

```
1   query {
2     fetchAllUnrelatedTypesInSubgraphB {
3         id
4         randomWord
5         relatedObjectsOfSameType {
6             id
7             randomWord
8         }
9     }
10    fetchAllBTypes {
```

```
        id
        isDefinedIn
        appearsFirstIn
    }
}
```

AsyncExecutionStrategy ("default")

As mentioned earlier, `AsyncExecutionStrategy` is the default execution strategy for `Query` operations in the graphql-java library. To apply this strategy nothing had to be added to the GraphQL configuration file. However, to ensure the same conditions for each of the strategies, the Spring bean `ExecutionStrategyProvider` was added to the configuration file:

```
    @Bean
    public ExecutionStrategyProvider executionStrategyProvider() {
        return new DefaultExecutionStrategyProvider(
            new AsyncExecutionStrategy(),
            null,
            null
        );
    }
```

The `null` values mean that the default execution strategies for `Mutation` and `Subscription` operations are not altered.

AsyncSerialExecutionStrategy ("Serial")

To enable `AsyncSerialExecutionStrategy`, from now on referred to as *serial*, the Spring bean `ExecutionStrategyProvider` was added to the configuration file.

```
    @Bean
    public ExecutionStrategyProvider executionStrategyProvider() {
        return new DefaultExecutionStrategyProvider(
            new AsyncSerialExecutionStrategy(),
            null,
            null
        );
    }
```

Concurrent Execution Strategy ("Concurrent")

This execution strategy is a variation of `AsyncExecutionStrategy`. With this strategy, data fetchers are able to fetch data concurrently. Recall our example query `B || B`. When using this strategy, the `BTypesDataFetcher` and `UnrelatedTypesInSubgraphBDataFetcher` will be called concurrently. The GraphQL implementation in Java does not use concurrency by default [48]. In this project, a dedicated thread pool was used to enable concurrency, by adding the following code snippet to the GraphQL configuration file:

Listing 3.12: Dedicated thread pool used to achieve concurrent data fetching

```
    private ExecutorService threadPool = Executors.newFixedThreadPool(Runtime
            .getRuntime()
            .availableProcessors()
    );
```

As the tests are all conducted on a MacBook Pro from 2021 with an 32 GB Apple M1 Pro chip, ten threads will available to the thread pool. To use the dedicated thread pool in the data fetchers, they had to be wrapped in asynchronous data fetchers which return a `CompletableFuture<T>` instead of a `T`. This makes it possible for several data fetchers to be called in parallel, since they do not have to wait for the previous call to be completed. The `RuntimeWiring` for Subgraph B for this execution strategy is found in **Listing 3.13**.

Listing 3.13: RuntimeWiring for Parallel execution strategy

```
private RuntimeWiring createRuntimeWiring() {
    return RuntimeWiring.newRuntimeWiring()
            .type("Query", builder ->
                    builder
                            .dataFetcher("lookupBType",
                                AsyncDataFetcher.async(bTypeDataFetcher, threadPool))
                            .dataFetcher("fetchAllBTypes",
                                AsyncDataFetcher.async(bTypesDataFetcher, threadPool))
                            .dataFetcher("lookupUnrelatedTypeInSubgraphB",
                                AsyncDataFetcher.async(unrelatedTypeInSubgraphBDataFetcher,
                                    threadPool))
                            .dataFetcher("fetchAllUnrelatedTypesInSubgraphB",
                                AsyncDataFetcher.async(unrelatedTypesInSubgraphBDataFetcher,
                                    threadPool))
            )
            .type("Mutation", builder ->
                    builder
                        .dataFetcher("createBType", createBTypeDataFetcher)
            )
            /* this is needed if one of BType's fields ("relatedObjectInSubgraphA) is of type
            Entity AType */
            .type("BType", builder ->
                    builder
                        .dataFetcher("relatedObjectInSubgraphA",
                                AsyncDataFetcher.async(aTypeDataFetcher, threadPool))
            )
            .type("AType", builder ->
                    builder
                        .dataFetcher("relatedObjectsInSubgraphB",
                            AsyncDataFetcher.async(bTypesDataFetcher,threadPool))
            )
            .type("UnrelatedTypeInSubgraphB", builder ->
                    builder
                        .dataFetcher("relatedObjectsOfSameType",
                            AsyncDataFetcher.async(relatedObjectsOfSameTypeDataFetcher,
                                threadPool))
            )
            .build();
}
```

Some readers may notice that the data fetcher for the `Mutation` operation is not wrapped in an `AsyncDataFetcher`. The reason for this is, as mentioned in the theory section, that the `Mutation` operation requires the use of `AsyncSerialExecution-Strategy`.

3.2.4 Instrumentation

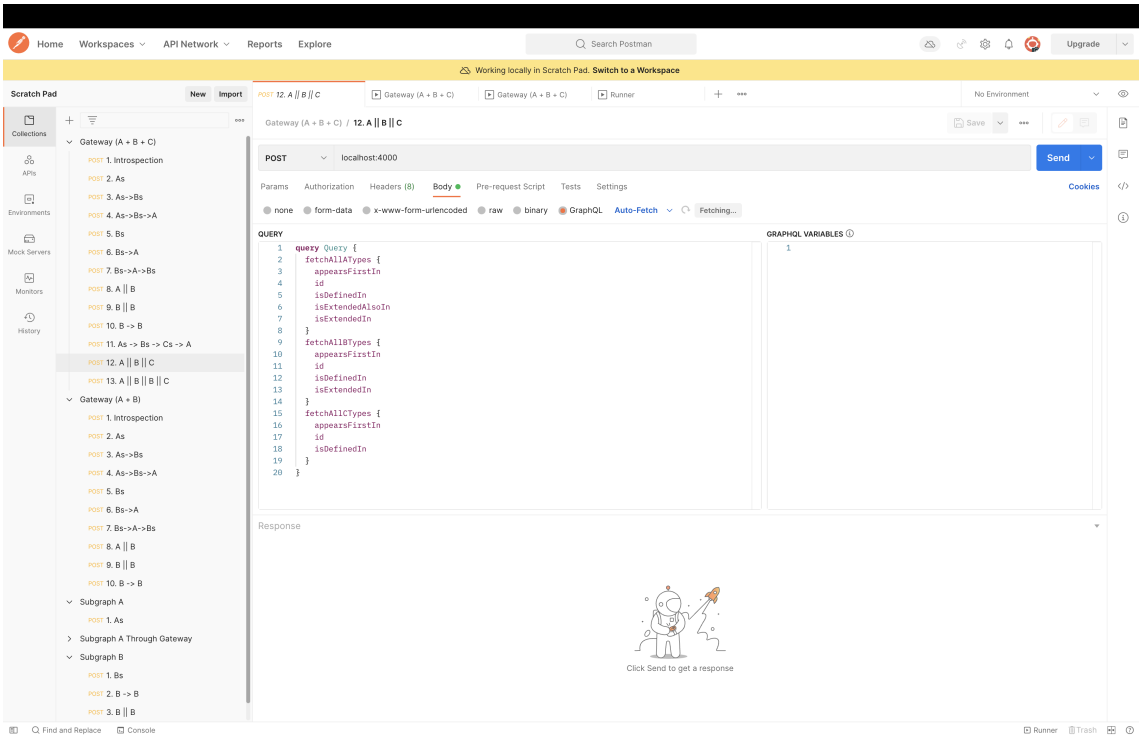
To learn more about how the evaluated execution strategies behaved in practice and to make sure that they did not behave unexpectedly, some instrumentation was used when developing the test application. Since `FederatedTracingInstrumentation` [37] only returned protobuf [49] messages intended for Apollo Studio [50], the decision was made to used the open source library Open Telemetry [51] to fetch gateway metrics instead.

To study how the execution strategies behaved in terms of threads and concurrency, a class `RequestLoggingInstrumentation` [52] is used.

3.2.5 Test Environment

The test environment was setup in Postman [53] where four query collections were defined: Subgraph A, Subgraph B, Gateway (A + B), and Gateway (A + B + C), as seen in **Figure 3.3**.

Figure 3.3: Test environment in Postman



The queries in each test collection were also grouped together by type:

1. Federated queries – queries which fetch data from more than one subgraph
2. Non-federated queries – queries which fetch data from one subgraph

The main goal of the collection Gateway (A + B) was to evaluate the overall performance of the different execution strategies, while Gateway (A + B + C), a super set of Gateway (A + B), was a means to explore if any further latency could be recorded after the introduction of another subgraph. The collections Subgraph A

and Subgraph B, both subsets of the gateway collections, had the purpose of investigating if the performance was better when querying the subgraph directly and not going through the gateway server.

Gateway (A + B)

Below are queries defined in the test collection Gateway (A + B). The query bodies are found in A.

1. `introspection`: used as a baseline for the other queries as it is not affected by the choice of execution strategy [15].
2. `As`: accessing all `A` instances with their attribute fields, `relatedObjectsInSubgraphB` excluded.
3. `As -> Bs`: Accessing all `A` instances with their attribute fields, including `relatedObjectsInSubgraphB`.
4. `As -> Bs -> A`: causing a many-to-one cycle.
5. `Bs`: accessing all `B` objects with their attribute fields, `relatedObjectInSubgraphA` excluded.
6. `Bs -> A`: accessing all `B` objects with attribute fields, including `relatedObjectInSubgraphA` with the fields defined in Subgraph A.
7. `Bs -> A -> Bs`: accessing all `Bs` with their respective `As` with all attributes, causing a many-to-many cycle, inspired by [15].
8. `B || B`: Parallel fetch of two types defined in Subgraph B (`B` and `UnrelatedTypeInSubgraphB`). The reason for including this query was to see if asynchronous data fetchers would show any latency reduction, since they can be executed in parallel.
9. `A || B`: This query is equivalent with query 2 and 5, fetched in parallel.
10. `B -> B`: Exploring the performance of a cyclic one-to-many relationship pointing towards objects of the same type.

Subgraph A

1. `As`: Same as in Gateway (A + B).

Subgraph B

1. `Bs`: Same as in Gateway (A + B).
2. `B -> B`: Same as in Gateway (A + B).
3. `B || B`: Same as in Gateway (A + B).

Gateway (A + B + C)

Apart from the queries defined in `Gateway (A + B)`, this collection had two additional queries:

1. `As -> Bs -> Cs -> A`: A many-to-one cyclic query involving fetching data from all three micro services.
2. `A || B || C` : Three parallel queries fetching data from Subgraph A, B, and C. The reason for including this query is to investigate if the performance is affected by the introduction of another micro service.
3. `A || B || B || C`: This query consists of four parallel queries. The query was included with the goal of investigating if the breadth of the query affect the execution strategies' performance.

3.3 Testing Methodology

The purpose of this study is to evaluate the performance of different execution strategies in GraphQL, and to investigate if Apollo Federation affects response time. However, it is important to control the test environment in order to give each strategy equal conditions. According to Blackburn et al. [54], "controlling for code warm-up is an important aspect of experimental design for high-performance run-times", which means that the cost of the JVM start-up must be taken into account when running the tests. The first iteration of an application is usually the one with the largest amount of dynamic compilation [54]. One could also argue that it would be reasonable to record performance after the JVM has warmed up and the application is in its most stable state, as this is the most common use-case [54].

The tests were conducted in Postman in offline-mode ("Scratch pad", with Wi-Fi turned off), as an attempt to control the test environment further. Each test suit involved one given collection of queries and one given execution strategy. To be able to record the average response times, a number of global variables were defined and JavaScript was used to update these variables during each test. Before each test run, the JVM was restarted and the GraphQL requests in the given collection were sent 100 times to get the application into a steady state. Then, this time measuring the response times, the queries were executed eleven times. When calculating the mean, the first request of each query was ignored since it was always clearly higher than the rest of the data points, which might indicate that some dynamic compilation was taking place during the first round. The JSON files generated by Postman during these tests are available in the GitHub repository for this thesis project. There, the interested reader can inspect the data.

When testing if there is a relationship between the number of subgraphs and performance in a federated architecture, the requests sent directly to the subgraph and not to the gateway served as a baseline result. All the measurements were conducted with the execution strategy `AsyncExecutionStrategy` since the most important thing here was to measure possible latency peaks due to an increasing number of subgraphs being connected to the gateway server.

3.3.1 Evaluation of Sources

In this thesis, the sources consist of academic papers, software documentation, and gray literature (blog posts and other non-academic material). There are two main reasons for using, and therefore trusting, these sources. The first reason is that

GraphQL is a relatively new technology. The number of academic papers published on the subject is therefore somewhat limited. The second reason is the practical nature of this project. A substantial part of conducting this study was building the test environment, and therefore, there was a need for descriptions of implementation details. Moreover, the gray literature used is written by authors who are active in the GraphQL community and help developing the technology, e.g., Lee Byron at Facebook, Andreas Marek and Brad Baker who has a central role in developing GraphQL Java, and Roman Kudryashov who wrote the library used in this thesis to add Apollo Federation support to GraphQL Java.

4

Results

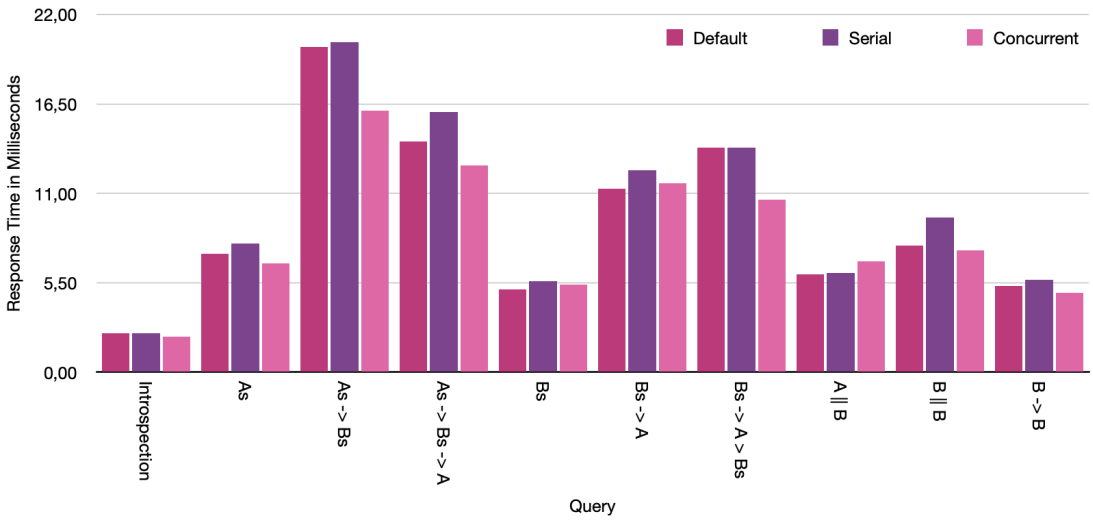
In this section, the results from the test will be presented. First, the results related to the evaluation of execution strategies are demonstrated. After that, test results will be presented related to a possible relationship between performance in Apollo Federation and the number of subgraphs making up the composed supergraph.

4.1 Evaluation of Execution Strategies

4.1.1 Query Collection Gateway (A + B)

In this section, the results for the queries to Gateway (A + B) are given. In **Figure 4.1**, the response time for each query is presented for both default (`AsyncExecutionStrategy`), serial (`AsyncSerialExecutionStrategy`), and concurrent execution strategy. The same results are demonstrated in **Figure 4.1** as well.

Figure 4.1: Response times for all queries in collection Gateway (A + B) as a function of the choice of execution strategy.



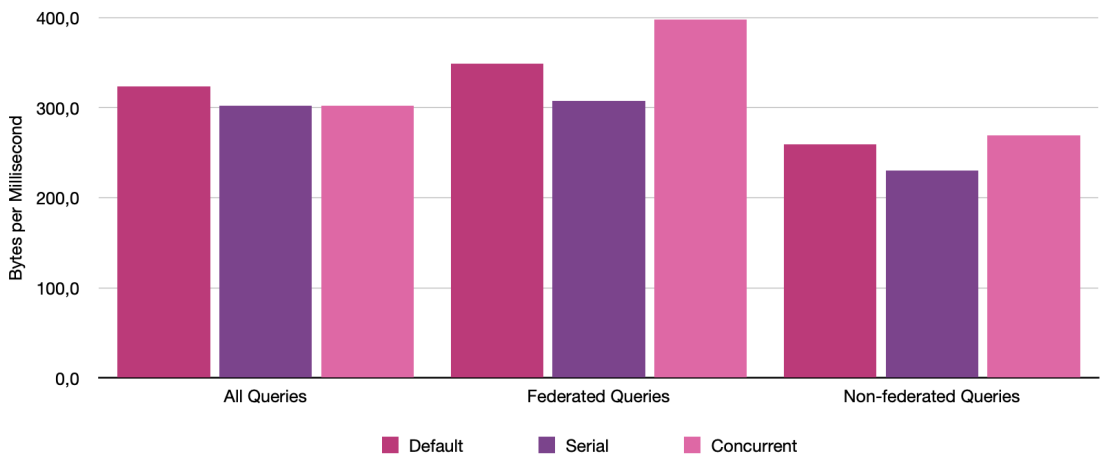
average response time for each query is presented. In the response size, the header is included.

The introspection query (baseline) is not effected by the choice of execution strategy to the same extent as the rest of the queries. To obtain an average measure of the strategies' performance across the board the bit rate, which is the number of bits processed per second, was calculated. However, to avoid large numbers, the units bytes per ms were used instead. The following equation was used:

$$Performance_{avg}(bytes/ms) = \frac{\sum_{i=1}^{n} R_i^{size}}{\sum_{i=1}^{n} R_i^{latency}} \tag{4.1}$$

where R_i^{size} is the size of a given query response and $R_i^{latency}$ is the response time for that response and execution strategy. Moreover, n is the number of queries excluding the baseline. The results are presented in **Figure 4.2**.

Figure 4.2: Average speed of data fetching in Gateway (A + B)



These results are unaware of cyclic dependencies, depth, and breadth of the queries, but provide an overall measure of speed. Nevertheless, the most conspicuous result is that of the concurrent execution strategy used on federated queries, that is, queries involving more than one subgraph. In this condition, concurrent execution strategy performed better than the default execution strategy. For all queries, on the other hand, the concurrent strategy was no better than the serial, and the default strategy showed the best performance.

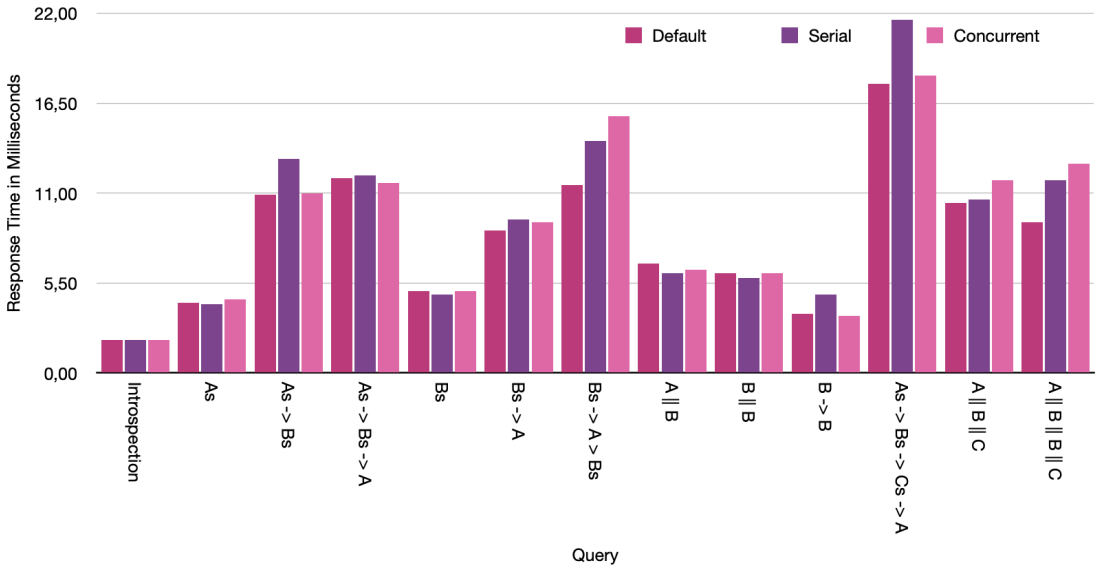
4.1.2 Query Collection Gateway (A + B + C)

The results for the Postman tests for collection Gateway (A + B + C) are presented in **Figure 4.3**. The `Introspection` query is used as baseline. As expected, latency was not affected by the choice of execution strategies.

Table 4.1: Results for queries in collection Gateway (A + B) for default, serial, and concurrent execution strategies

Query	Federated Query	Response Size (Bytes)	Execution Strategy	Response Time (ms)
Introspection	No	679	Default Serial Concurrent	2.40 2.40 2.20
As	No	916	Default Serial Concurrent	7.30 7.90 6.70
As → Bs	Yes	5050	Default Serial Concurrent	20.0 20.3 16.1
As → Bs → A	Yes	5050	Default Serial Concurrent	14.2 16.0 12.7
Bs	No	1540	Default Serial Concurrent	5.10 5.60 5.40
Bs → A	Yes	3310	Default Serial Concurrent	11.3 12.4 11.6
Bs → A → Bs	Yes	7220	Default Serial Concurrent	13.8 13.8 10.6
A ‖ B	Yes	2180	Default Serial Concurrent	6.00 6.10 6.80
B ‖ B	No	2710	Default Serial Concurrent	7.80 9.50 7.50
B → B	No	1440	Default Serial Concurrent	5.30 5.70 4.90

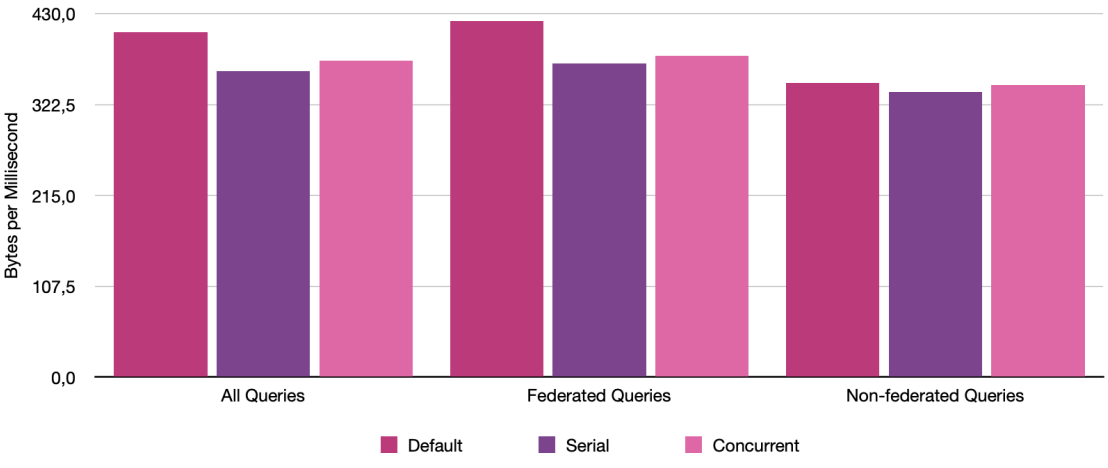
Figure 4.3: Response times for every query in collection Gateway (A + B + C) for each of the evaluated execution strategies



In **Table 4.2**, the average response time for each query and execution strategy is laid out.

The byte rates for query collection Gateway (A + B + C) are presented in **Figure 4.4**.

Figure 4.4: Average speed of data fetching in Gateway (A + B + C)



When calculating the byte rate, the results from all the queries in test collection Gateway (A + B + C) show that the speed of default execution was on average 408.5 Bytes / ms, 362.3 Bytes / ms for serial execution, and 374.2 Bytes / ms for concurrent execution. Note that the results for from Gateway (A + B) was not

Table 4.2: Results for queries in collection Gateway (A + B + C) for execution strategies default, Serial, and Concurrent

Query	Federated Query	Response Size (Bytes)	Execution Strategy	Response Time (ms)
Introspection	No	679	default	2.00
			Serial	2.00
			Concurrent	2.00
As	No	916	default	4.30
			Serial	4.20
			Concurrent	4.50
As → Bs	Yes	5050	Default	10.9
			Serial	13.1
			Concurrent	11.0
As → Bs → A	Yes	5050	Default	11.9
			Serial	12.1
			Concurrent	11.6
Bs	No	1540	Default	5.00
			Serial	4.80
			Concurrent	5.00
Bs → A	Yes	3310	Default	8.70
			Serial	9.40
			Concurrent	9.20
Bs → A → Bs	Yes	7220	Default	11.5
			Serial	14.2
			Concurrent	15.7
A ‖ B	Yes	2180	Default	6.70
			Serial	6.10
			Concurrent	6.30
B ‖ B	No	2710	Default	6.10
			Serial	5.80
			Concurrent	6.10
B → B	No	1440	Default	3.60
			Serial	4.80
			Concurrent	3.50
As → Bs → Cs → A	Yes	4770	Default	17.7
			Serial	21.6
			Concurrent	18.2
A ‖ B ‖ C	Yes	3970	Default	10.4
			Serial	10.6
			Concurrent	11.8
A ‖ B ‖ B ‖ C	Yes	5140	Default	9.20
			Serial	11.8
			Concurrent	12.8

replicated for this query collection, as the default execution strategy outperforms both the serial and concurrent strategies.

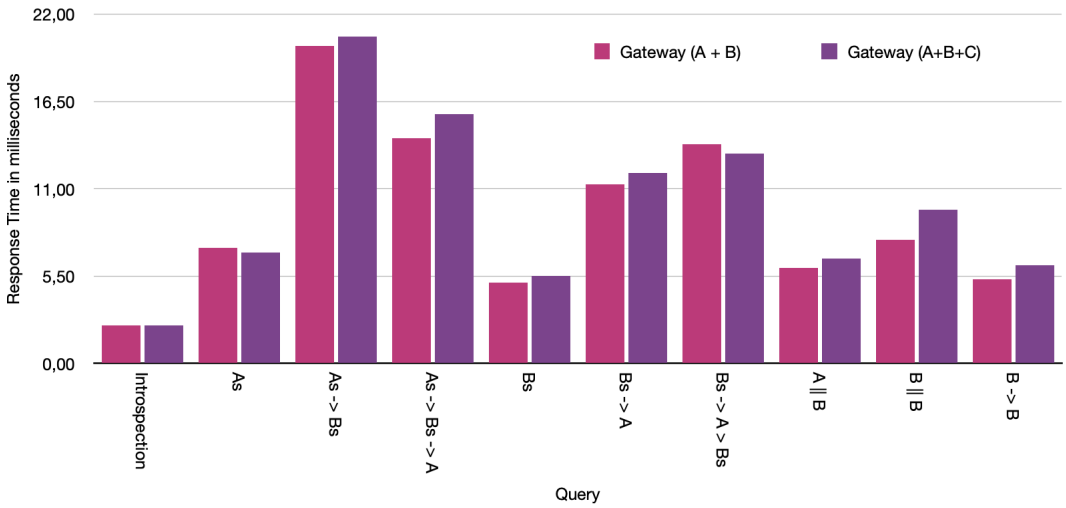
4.2 The Effect of Number of Subgraphs on Response Times

In this section, results on how the number of subgraphs in a federated architecture affect latency are presented.

4.2.1 Query Collection Gateway (A + B)

In this section, the queries are taken from query collection Gateway (A + B) which consists of ten queries, including the baseline `Introspection`. To try to answer research question 2, *To what extent does the number of subgraphs affect performance in Apollo Federation?*, the test queries remained constant while the number of subgraphs registered in the Apollo gateway server varied. For each query, the lighter staple represents a federated architecture with two subgraphs, while the darker staple represents Gateway (A + B + C). These results are also presented in **Table 4.3**.

Figure 4.5: Response times for AsyncExecutionStrategy ("default") as a function of number of subgraphs



4.2.2 Subgraph A

In **Table 4.4**, the latency for query `As` is given for four different conditions. The first condition is when the subgraph is queried directly, that is, the request is sent to `localhost:8080/graphql`. That result is used as a baseline for the other conditions.

The latency for sending the same query to the Apollo gateway server on `localhost:4000`, with only Subgraph A connected to it, is 83.9 % higher. However, the same query sent to a unified supergraph consisting of two and three subgraphs are 47.7 % respectively 46.2 % higher.

The execution strategy used on these tests was `AsyncExecutionStrategy`.

Table 4.3: Response times for queries sent to the gateway server, with either two or three subgraphs registered

Query	Gateway (A + B)	Gateway (A + B + C)
Introspection (Baseline)	3.76	3.33
As	14.03	13.47
As → Bs	29.86	34.70
As → Bs → A	32.43	30.03
Bs	14.20	16.03
Bs → A	25.87	26.27
Bs → A → Bs	28.00	33.53
A ‖ B	18.74	18.90
B ‖ B	20.13	18.40
B → B	10.37	12.60

Table 4.4: Response time (mean) for the sole query in test collection Subgraph A

Query	API	Number of Subgraphs	Response Time (ms)
As	localhost:8080/graphql	1	6.50
As	localhost:4000	1	11.89
As	localhost:4000	2	9.60
As	localhost:4000	3	9.50

4.2.3 Subgraph B

In **Table 4.5**, the latency for test collection Subgraph B is given for four different conditions. The first condition is when the subgraph is queried directly, that is, the request is sent to `localhost:8081/graphql`. That result is used as a baseline for the other conditions.

Unlike Subgraph A, Subgraph B cannot be the sole subgraph connected to the Apollo gateway server, since `AType` is extended in Subgraph B, and changing the schema was not desirable. Therefore there is no result for that condition.

The latency for query `Bs` was 28.6 % higher than the baseline when querying the Apollo server with two connected subgraphs. On the other hand, the latency for querying a supergraph consisting of three subgraphs decreased with 2.4 % compared to the baseline. The execution strategy used during these tests was `AsyncExecutionStrategy`.

Table 4.5: Response time (mean) for the queries in test collection Subgraph B

Query	API	Number of Subgraphs	Response Time (ms)
Bs	localhost:8081/graphql	1	8.40
	localhost:4000	2	10.80
	localhost:4000	3	8.20
B $\rightarrow$ B	localhost:8081/graphql	1	7.90
	localhost:4000	2	10.50
	localhost:4000	3	8.90
B || B	localhost:8081/graphql	1	10.0
	localhost:4000	2	10.50
	localhost:4000	3	10.30

5

Discussion

This section contains a discussion on the results. However, a caveat is required since there are variables affecting the performance measurements that cannot be completely controlled, for example the JVM and the operative system that the JVM runs on. There is also a possibility that there are unknown variables affecting the result. It is important to bear this in mind when reading the next section.

5.1 Evaluation of Execution Strategies

In this section, the results related to **RQ1** – *How is the performance in Apollo Federation affected by the choice of execution strategy?* – are discussed. First, the results from query collection Gateway (A + B) will be analyzed, followed by the results from the gateway with three services, Gateway (A + B + C).

5.1.1 Gateway (A + B)

While there default execution strategy, `AsyncExecutionStrategy`, showed the best overall performance in this query collection, some results suggest that there is in fact a gain from using concurrent data fetching, e.g., for queries `A - > Bs` and `Bs -> A -> Bs`. Furthermore, the results suggest that the choice of execution strategy has a greater impact on nested queries, that is, any query with a name containing an arrow, for example "`Bs -> A`". For queries translated to a single fetch node — like "`As`" and "`Bs`" – serial execution strategy performs slightly better than standard and concurrent strategies. A possible explanation for this is that only one thread is active in the data fetcher fetching the data for "`Bs`" since parallelism is not needed since only one data fetcher is invoked, see **Figure 5.1**. For concurrent execution, however, the thread pool is still used, which comes at a cost, while the advantages of multi-threaded execution is not harvested. The logging for "`Bs`" with concurrent execution strategy can be seen in **Figure 5.2**, and it is clear that only one thread is used. In **Figure 5.3**, on the other hand, the parallel nature of query `B || B` becomes apparent as almost all threads in the thread pool are active at the same time.

Figure 5.1: Request logging for query "Bs" with execution strategy AsyncSericalExecutionStrategy

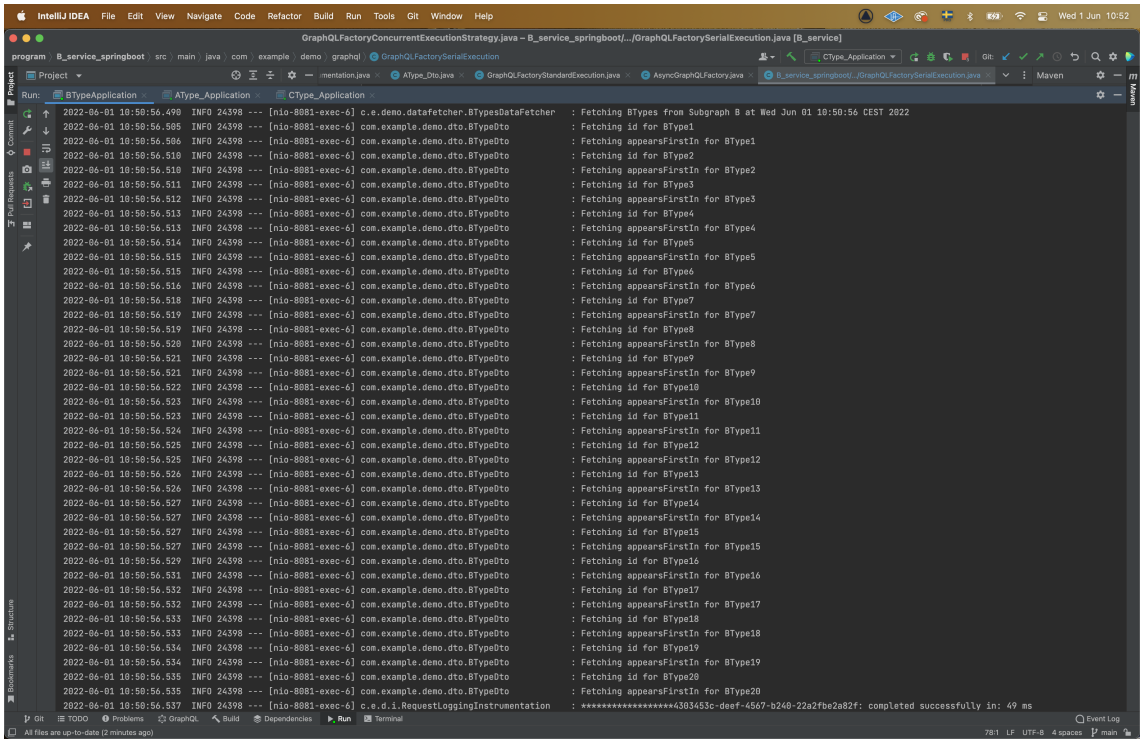


Figure 5.2: Request logging for query "Bs" with concurrent execution strategy

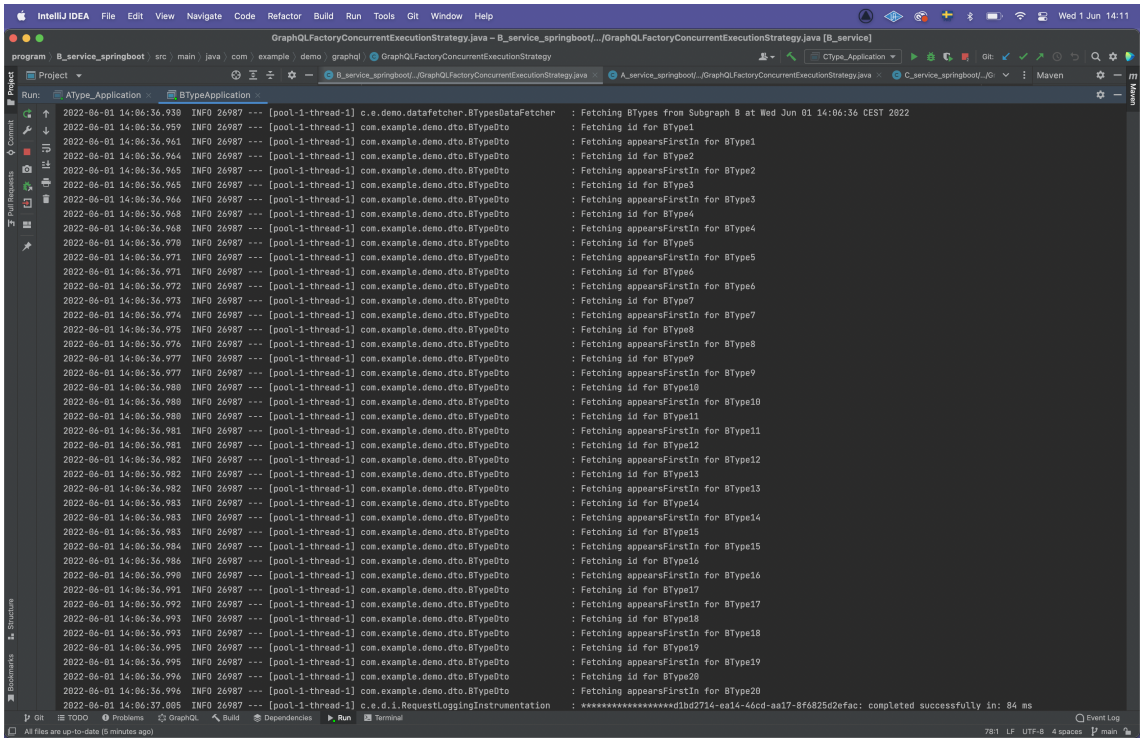
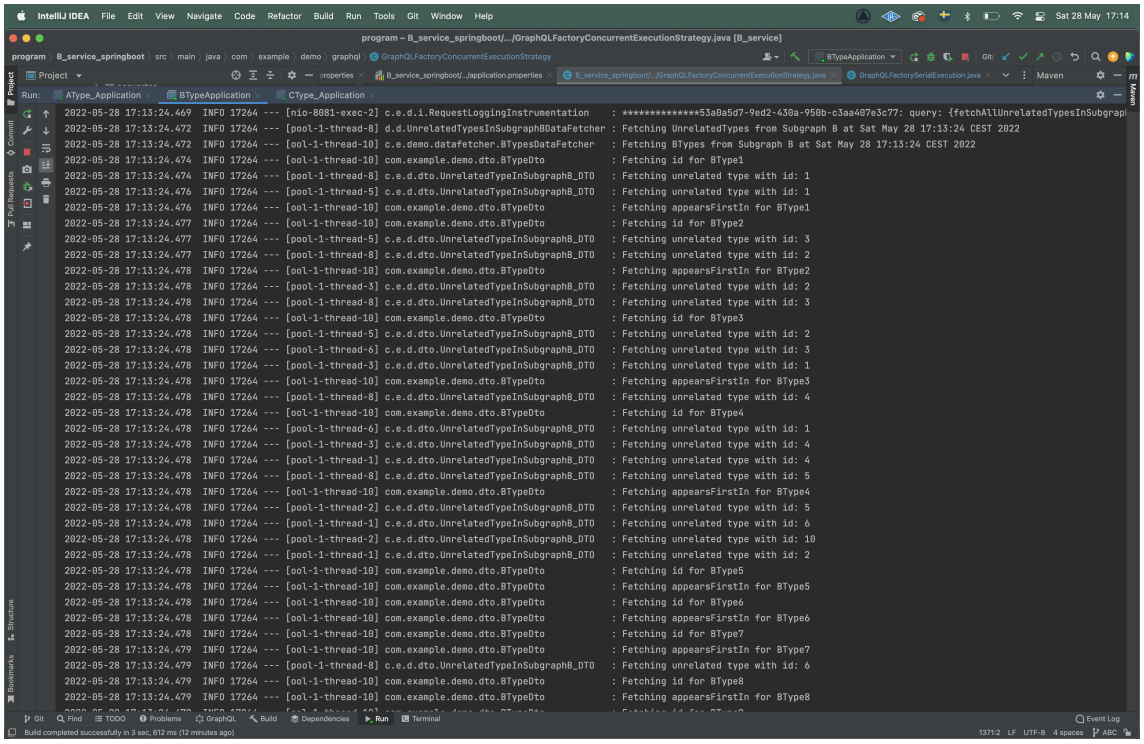


Figure 5.3: Request logging for query "B ||B" with concurrent execution strategy



5.1.2 Gateway (A + B + C)

These results suggest that `AsyncExecutionStrategy` is in fact the best performing execution strategy across the board. For a small project with only a few subgraphs, `AsyncExecutionStrategy` performs best in terms of latency. However, the results for Gateway (A + B + C) are not in line with the results for Gateway (A + B) regarding a possible performance gain by using a concurrent strategy. Instead, the concurrent execution strategy performs on par with or worse than the default strategy. For some queries, the concurrent strategy even performed worse than the serial. Therefore, no generalization can be made from this set of data regarding the concurrent strategy.

The slowest response time of all test items in this study was for the query "`As -> Bs -> Cs -> A`" using a serial execution strategy. For this query, the average latency for serial strategy was 22.0 % higher than the default execution strategy. A possible explanation for this is that the penalty of using a serial execution strategy grows as query depth increases.

An interesting observation is that the federated queries seem to perform better than the non-federated ones, just looking at how many bytes per milliseconds that are fetched. A possible reason for this is that the query planning in the gateway server contribute to better performance. However, in order to draw any conclusions, non-federated queries of the same depth would have to be tested too. There is only one nested non-federated query in the study – "`B -> B`" – which in **Figure 4.3** display considerably shorter response times than, e.g., "`Bs -> A`", which has the same depth.

Finally, batching, the strategy not implemented in this project, will receive some attention. Batching is necessary for any production ready GraphQL service. If batching had been implemented and tested, it is possible that any gain in performance would not have been as obvious as in [15] since there is no database to make remote calls to in this project.

5.2 Relationship Between Performance and Number of Subgraphs in Apollo Federation

In this section, a discussion will follow regarding the results acquired to answer **RQ2**, *To what extent does the number of subgraphs affect performance in Apollo Federation?*

5.2.1 Query Collection Subgraph A

The results in **Figure 4.4** suggest that performance is affected when querying a GraphQL schema through an Apollo gateway server as opposed to querying the subgraph directly. When replicating the test for the one-subgraph gateway server, the result showed that the latency increased with 58.3 % compared to the baseline, as opposed to 83.9 % in the original test. These numbers are too diverse to form a basis upon which to draw any conclusion regarding a "gateway effect". Nevertheless, the new result is more in line with the latency observed for the two- and three subgraph gateways (47.7 % respectively 46.2 % higher). These number do suggest that there is a trade off when querying the GraphQL schema through an Apollo gateway server. A possible explanation for a higher latency for a federated graph is the overhead related to the gateway's query planning.

This query collection contained only one query, which means that no discussion can be made regarding any other query type than one containing a single `Fetch` node, that is, breadth and depth of the query are both one.

5.2.2 Query Collection Subgraph B

As mentioned in **Chapter 4**, it was not possible to test the "one-subgraph-gateway" condition from the previous section without changing the schema in Subgraph B. Therefore, there are no results verifying the peak in latency for this condition in query collection Subgraph A. On the contrary, with decreasing latency for query "`Bs`" compared to the baseline, the results suggest no relationship between performance and the number of subgraphs in Apollo Federation per se. Moreover, the tests were conducted in a small environment and it is possible that a larger graph would have revealed other tendencies.

The results for query "`B -> B`" followed a similar pattern with a larger latency for two subgraphs, but a similar latency as the baseline for three subgraphs. The results for the parallel `B || B` showed only a 3 – 5 % increase in latency for two and three subgraphs. Based one these values it is not possible to draw any conclusion as to Apollo Federation having an effect on data fetching latency in a GraphQL server, neither in a positive nor negative way.

5.2.3 Query Collection Gateway (A + B)

In **Figure 4.5**, the response times for all the queries in **Section 3.2.5** are given, both when sent to Gateway (A + B) and Gateway (A + B + C). For six queries out of nine, the latency Gateway (A + B) was lower. According to these results, the latency for Gateway (A + B + C) was on average 5.87 % higher than for the federated graph consisting of two subgraphs, Gateway (A + B). However, this number is too low to suggest a relationship between number of subgraphs and performance in a federated supergraph.

6

Conclusion

6.1 Research Aims

The research aims with this study was to answer two questions:

- **RQ1**: How is the performance in Apollo Federation affected by the choice of execution strategy?
- **RQ2**: To what extent does the number of subgraphs affect performance in Apollo Federation?

To answer the first question, the results of this study suggest that the choice of execution strategy matters more with an increased query depth. The results go in line with the GraphQL specification which state that `AsyncExecutionStrategy` ("Default") offers best overall performance. Furthermore, the results point in the direction that concurrent data fetchers are not useful unless the queries are rather complex, involving several data fetchers, since starting separate threads comes at a cost. Serial execution strategy appears to work well in shallow and narrow queries, but will not offer the best performance for deeper queries, based on the these results. Therefore, it would be sensible to keep using the default execution strategy in a small project like the one in this study. However, no conclusions can be made regarding choosing execution strategies wisely in a larger federated architecture. Moreover, in a larger project, batching would be necessary in order to avoid the N + 1 problem. This study claims to be nothing but an evaluation of the performance with regard to latency for three execution strategies in a limited controlled space.

Regarding the second research question, it is not possible, on the basis of this study, to draw any conclusions about a relationship between the number of subgraphs in a federated architecture and performance with respect to latency. The test results show no clear pattern, and even if one had appeared, the data set would be too sparse to be able to deem it anything but circumstantial.

6.2 Research Objectives

The intention with conducting this research was to gain knowledge about GraphQL and Apollo Federation, while contributing with new insights regarding performance

of federated query processing in a test application written in Java.

6.3 Practical Implications

The study will hopefully bring to the table further knowledge about possible ways to use Apollo Federation in a Java context.

6.4 Future Research

For future research, it would be welcome to implement batching in this project and run the same tests to see if there is indeed a performance gain. Another possible path from here would be to investigate the performance in the new Apollo Federation II, which came out earlier this year, to see how it compares to this version.

6.5 Chapter Summary

In this chapter, the research questions were answered and a conclusion was made, followed by the research objectives and practical implications of this study. Finally, some suggestions for future research was made.

Bibliography

[1] V. Ravishankar, "Apollo sandbox: an open graphql ide for local development," 2021. Accessed 2022/05/23.

[2] Meteor, "Apollo Federation Documentation." Web page, 2022. Accessed 2022/03/03.

[3] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects," in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering FECS'13*, 2013.

[4] C. Richardson, "What are microservices?." Web page, 2021. Accessed 2022/03/03.

[5] R. Fielding and R. Taylor, "Principled design of the modern web architecture," in *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pp. 407–416, 2000.

[6] L. Shapton, "Solving the N+1 Problem for GraphQL through Batching." Web page, 2018. Accessed 2022/04/19.

[7] D. A. Hartina, A. Lawi, and B. L. E. Panggabean, "Performance analysis of graphql and restful in sim lp2m of the hasanuddin university," in *2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT)*, pp. 237–240, 2018.

[8] L. Byron, "GraphQL: A Data Query Language." Web page, 2022. Accessed 2022/03/04.

[9] B. Kane, "Coursera's Journey to GraphQL." Web page, 2020. Accessed 2022/03/03.

[10] G. MacWilliam, "To Federate or Stitch a GraphQL gateway, revisited." Web page, 2020. Accessed 2022/03/31.

[11] Meteor, "The what, when, why, and how of federated GraphQL." Web page, 2020. Accessed 2022/03/31.

[12] Capgemini, "About Us." Web page, 2020. Accessed 2022/05/25.

[13] M. Cederlund, "Performance of frameworks for declarative data fetching : An evaluation of falcor and relay+graphql," 2016.

[14] A. Cha, E. Wittern, G. Baudart, J. C. Davis, L. Mandel, and J. A. Laredo, *A*

Principled Approach to GraphQL Query Cost Analysis, p. 257–268. New York, NY, USA: Association for Computing Machinery, 2020.

[15] P. Roksela, M. Konieczny, and S. Zielinski, "Evaluating execution strategies of graphql queries," in *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, pp. 640–644, 2020.

[16] E. Lee, K. Kwon, and J. Yun, "Performance measurement of graphql api in home ess data server," in *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1929–1931, 2020.

[17] T. Shikhare, "How Netflix Scales its API with GraphQL Federation (Part 1)," 2020. Accessed 2022/05/12.

[18] GitHub, "GraphQL API," 2022. Accessed 2022/05/24.

[19] Meteor, "Apollo Federation Quickstart." Web page, 2022. Accessed 2022/02/21.

[20] CloudZero, "Netflix Architecture: How Much Does Netflix's AWS Cost?," 2021. Accessed 2022/03/05.

[21] A. Marek and B. Baker, "GraphQL Java Documentation." Web page, 2022. Accessed 2022/03/04.

[22] R. Kudryashov, "graphql-java-federation." Web page, 2020. Accessed 2022/05/25.

[23] Facebook, "GraphQL Specification." Web page, 2022. Accessed 2022/03/04.

[24] E. Porcello and A. Banks, *Learning GraphQL - Declarative Data Fetching For Modern Web Apps*. Sebastopol, CA: O'Reilly Media, Inc, USA, 2018.

[25] A. G. Inc., "GraphQL file types in Apollo Kotlin." Web page, 2022. Accessed 2022/03/29.

[26] baeldung, "Guide to java.util.concurrent.Future." Web page, 2021. Accessed 2022/05/25.

[27] baeldung, "Guide to java.util.concurrent.CompletableFuture." Web page, 2021. Accessed 2022/05/25.

[28] "Javadoc – Class AsyncSerialExcecutionStrategy." Web page, 2022. Accessed 2022/05/25.

[29] M.-A. Giroux, "The GraphQL Dataloader Pattern: Visualized." Web page, 2019. Accessed 2022/05/02.

[30] A. Nnakwue, "Entities in Apollo Federation - Reference and extend types across subgraphs." Web page, 2020. Accessed 2022/04/13.

[31] K. M. Endris, M.-E. Vidal, and D. Graux, *Chapter 5 Federated Query Processing*, pp. 73–86. Cham: Springer International Publishing, 2020.

[32] M. Wise, "Supercharge Your Data Graph with Apollo Federation," 2021. Accessed 2022/03/23.

[33] Y. W. Kim, M. P. Consens, and O. Hartig, "An empirical analysis of graphql api schemas in open code repositories and package registries," in *AMW*, 2019.

[34] Meteor, "Apollo Federation Documentation – Query plans." Web page, 2022. Accessed 2022/04/07.

[35] M. Vogel, S. Weber, and C. Zirpins, "Experiences on migrating restful web services to graphql," in *ICSOC Workshops*, 2017.

[36] O. Hartig and J. Pérez, "Semantics and complexity of graphql," in *Proceedings of the 2018 World Wide Web Conference*, pp. 1155–1164, 2018.

[37] R. Kudryashov, "How to GraphQL in Kotlin and Micronaut and create a single endpoint for access to microservices' APIs," 2020. Accessed 2022/05/12.

[38] A. Karlsson, "Automatic exposure of data using graphql and apollo federation," 2021.

[39] Spring, "GraphQL and GraphiQL Spring Framework Boot Starters," 2020. Accessed 2022/05/12.

[40] Spring, "Spring Boot," 2020. Accessed 2022/05/25.

[41] Spring, "The IoC Container." Web page, 2022. Accessed 2022/06/06.

[42] Spring, "Spring Documentation," 2022. Accessed 2022/05/25.

[43] TutorialsPoint, "Spring Autowired Annotation." Web page, 2022. Accessed 2022/05/25.

[44] A. Marek and B. Baker, "GraphQL Java Documentation/Schema." Web page, 2022. Accessed 2022/03/04.

[45] Apollo, "Implementing the Gateway." Web page, 2022. Accessed 2022/05/25.

[46] slf4j.org, "Simple Logging Facade for Java." Web page, 2021. Accessed 2022/05/25.

[47] M. Fowler, "Data Transfer Object." Web page, 2022. Accessed 2022/05/25.

[48] T. Nurkiewicz, "GraphQL server in Java: Part III: Improving concurrency," 2020. Accessed 2022/05/12.

[49] Google, "Protocol Buffers." Web page, 2022. Accessed 2022/05/25.

[50] Apollo, "Introduction to Apollo Studio." Web page, 2022. Accessed 2022/05/25.

[51] Apollo, "OpenTelemetry in Apollo Federation." Web page, 2019. Accessed 2022/05/17.

[52] P. Starritt, "Learn Spring Boot GraphQL." Web page, 2020. Accessed 2022/05/25.

[53] Postman, "Postman," 2022.

[54] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "Wake up and smell the

coffee: Evaluation methodology for the 21st century," *Commun. ACM*, vol. 51, p. 83–89, aug 2008.

Appendix A

Test Queries

Listing A.1: Introspection

```
query {
    __schema {
        types {
            name
        }
    }
}
```

Listing A.2: As

```
query {
    fetchAllATypes {
        id
        appearsFirstIn
        isDefinedIn
    }
}
```

Listing A.3: As → Bs

```
query {
    fetchAllATypes {
        id
        isDefinedIn
        isExtendedIn
        appearsFirstIn
        relatedObjectsInSubgraphB {
            id
            isDefinedIn
            appearsFirstIn
            relatedObjectInSubgraphA {
                id
                isDefinedIn
                isExtendedIn
                appearsFirstIn
            }
        }
    }
}
```

Listing A.4: $As \rightarrow Bs \rightarrow A$

```
query {
   fetchAllATypes {
       id
       appearsFirstIn
       isDefinedIn
       isExtendedIn
       relatedObjectsInSubgraphB {
           id
           appearsFirstIn
           isDefinedIn
           relatedObjectInSubgraphA {
               id
               appearsFirstIn
               isDefinedIn
               isExtendedIn
           }
       }
   }
}
```

Listing A.5: Bs

```
query {
    fetchAllBTypes {
        id
        appearsFirstIn
        isDefinedIn
    }
}
```

Listing A.6: $Bs \rightarrow A$

```
query{
    fetchAllBTypes {
        id
        isDefinedIn
        appearsFirstIn
        relatedObjectInSubgraphA {
            id
            appearsFirstIn
            isDefinedIn
        }
    }
}
```

Listing A.7: $Bs \rightarrow A \rightarrow Bs$

```
query{
    fetchAllBTypes {
        id
        isDefinedIn
        appearsFirstIn
        relatedObjectInSubgraphA {
            id
            appearsFirstIn
            isDefinedIn
            isExtendedIn
            relatedObjectsInSubgraphB {
```

```
                id
                appearsFirstIn
                isDefinedIn
            }
        }
    }
}
```

Listing A.8: $A \parallel B$

```
query {
    fetchAllBTypes {
        id
        appearsFirstIn
        isDefinedIn
    }
    fetchAllATypes {
        id
        appearsFirstIn
        isDefinedIn
    }
}
```

Listing A.9: $B \parallel B$

```
query {
    fetchAllUnrelatedTypesInSubgraphB {
        id
        randomWord
        relatedObjectsOfSameType {
            id
            randomWord
        }
    }
    fetchAllBTypes {
        id
        isDefinedIn
        appearsFirstIn
    }
}
```

Listing A.10: $B \rightarrow B$

```
query {
    fetchAllUnrelatedTypesInSubgraphB {
        id
        randomWord
        relatedObjectsOfSameType {
            id
            randomWord
        }
    }
}
```

Listing A.11: $As \rightarrow Bs \rightarrow Cs \rightarrow A$

```
query {
  fetchAllATypes {
    appearsFirstIn
```

```
    id
    isDefinedIn
    isExtendedIn
    relatedObjectsInSubgraphB {
      appearsFirstIn
      id
      isDefinedIn
      isExtendedIn
      relatedObjectsInSubgraphC {
        appearsFirstIn
        id
        isDefinedIn
        relatedObjectInSubgraphA {
          appearsFirstIn
          id
          isDefinedIn
          isExtendedIn
        }
      }
    }
  }
}
```

Listing A.12: A $\|B\|C$

```
query {
  fetchAllATypes {
    appearsFirstIn
    id
    isDefinedIn
    isExtendedAlsoIn
    isExtendedIn
  }
  fetchAllBTypes {
    appearsFirstIn
    id
    isDefinedIn
    isExtendedIn
  }
  fetchAllCTypes {
    appearsFirstIn
    id
    isDefinedIn
  }
}
```

Listing A.13: A $\|B\|B\|C$

```
query {
  fetchAllATypes {
    appearsFirstIn
    id
    isDefinedIn
    isExtendedAlsoIn
    isExtendedIn
  }
  fetchAllBTypes {
    appearsFirstIn
    id
```

```
    isDefinedIn
    isExtendedIn
  }
  fetchAllUnrelatedTypesInSubgraphB {
      id
      randomWord
      relatedObjectsOfSameType {
          id
          randomWord
      }
  }
  fetchAllCTypes {
    appearsFirstIn
    id
    isDefinedIn
  }
}
```

Appendix B

Selected Source Code

B.1 The Gateway Server

Listing B.1: gateway.js

```
const { Resource } = require('@opentelemetry/resources');
const { SimpleSpanProcessor, ConsoleSpanExporter } = require ("@opentelemetry/sdk-trace-base");
const { NodeTracerProvider } = require("@opentelemetry/sdk-trace-node");
const { registerInstrumentations } = require('@opentelemetry/instrumentation');
const { HttpInstrumentation } = require ('@opentelemetry/instrumentation-http');
const { ExpressInstrumentation } = require ('@opentelemetry/instrumentation-express');
const { ZipkinExporter } = require("@opentelemetry/exporter-zipkin");
const { ApolloServer } = require("apollo-server");
const { ApolloGateway } = require("@apollo/gateway");
const {serializeQueryPlan} = require('@apollo/query-planner');
const {ApolloServerPluginCacheControl} = require('apollo-server-core');
const {ApolloServerPluginInlineTrace} = require('apollo-server-core');
/*
// Register server-related instrumentation
registerInstrumentations({
  instrumentations: [
    new HttpInstrumentation(),
    new ExpressInstrumentation(),
    // **DELETE IF SETTING UP A GATEWAY, UNCOMMENT OTHERWISE**
    //new GraphQLInstrumentation()
  ]
});

// Initialize provider and identify this particular service
// (in this case, we're implementing a federated gateway)
const provider = new NodeTracerProvider({
  resource: Resource.default().merge(new Resource({
    // Replace with any string to identify this service in your system
    "service.name": "gateway",
  })),
});

// Configure a test exporter to print all traces to the console
const consoleExporter = new ConsoleSpanExporter();
provider.addSpanProcessor(
  new SimpleSpanProcessor(consoleExporter)
);
```

```
38
39 // Register the provider to begin tracing
40 provider.register();*/
41
42 const gateway = new ApolloGateway({
43     serviceList: [
44         // This entire `serviceList` is optional when running in managed federation
45         // mode, using Apollo Graph Manager as the source of truth.  In production,
46         // using a single source of truth to compose a schema is recommended and
47         // prevents composition failures at runtime using schema validation using
48         // real usage-based metrics.
49         { name: "A_service", url: "http://localhost:8080/graphql" },
50         { name: "B_service", url: "http://localhost:8081/graphql" },
51         { name: "C_service", url: "http://localhost:8082/graphql" }
52
53     ],
54 //     I'm commenting this so that it's not taking CPU from the tests
55
56
57     /*
58     // Experimental: Enabling this enables the query plan view.
59     __exposeQueryPlanExperimental: true,
60       experimental_didResolveQueryPlan: function(options) {
61         if (options.requestContext.operationName !== 'IntrospectionQuery') {
62           console.log(serializeQueryPlan(options.queryPlan));
63         }
64       }
65     */
66 });
67
68 (async () => {
69     const server = new ApolloServer({
70         gateway,
71         //tracing: true,
72         tracing: false,
73
74         // Apollo Graph Manager (previously known as Apollo Engine)
75         // When enabled and an `ENGINE_API_KEY` is set in the environment,
76         // provides metrics, schema management and trace reporting.
77         engine: false,
78
79         // Subscriptions are unsupported but planned for a future Gateway version.
80         subscriptions: false,
81         //cacheControl: true
82         //plugins: [ApolloServerPluginInlineTrace()]
83     });
84
85     server.listen().then(({ url }) => {
86         console.log(`Server ready at ${url}`);
87     });
88 })();
```

B.2 Subgraph A/ATypeService

Listing B.2: Application configuration in application.properties

```
spring.application.name:ATypeService
spring.cache.type:NONE
server.port:8080

graphql:
    servlet:
    //async-mode-enabled:false
    tracing-enabled:true
```

Listing B.3: LookupATypeDataFetcher.java

```
@Slf4j
@Component
public class LookupATypeDataFetcher implements DataFetcher<AType_Dto> {

    @Autowired
    private ATypeConverter aTypeConverter;
    @Autowired
    private ATypeService aTypeService;

    @Override
    public AType_Dto get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        log.info("Fetching AType data from Subgraph A");
        String id = dataFetchingEnvironment.getArgument("id");
        AType aType = aTypeService.getATypeById(Integer.parseInt(id));
        return aTypeConverter.apply(aType);
    }
}
```

Listing B.4: FetchAllATypesDataFetcher.java

```
@Slf4j
@Component
public class LookupATypeDataFetcher implements DataFetcher<ATypeDTO> {

    @Autowired
    private ATypeConverter aTypeConverter;
    @Autowired
    private ATypeService aTypeService;

    @Override
    public ATypeDTO get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        log.info("Fetching AType data from Subgraph A");
        String id = dataFetchingEnvironment.getArgument("id");
        AType aType = aTypeService.getATypeById(Integer.parseInt(id));
        return aTypeConverter.apply(aType);
    }
}
```

Listing B.5: CreateATypeDataFetcher.java

```java
@Component
public class CreateATypeDataFetcher implements DataFetcher<ATypeDTO> {

    @Autowired
    ATypeService aType_service;
    @Autowired
    ATypeConverter aType_Converter;


    @Override
    public ATypeDTO get(DataFetchingEnvironment environment) throws Exception {
        String appearsFirstIn = environment.getArgument("appearsFirstIn");
        AType aType = aType_service.create(appearsFirstIn);
        return aType_Converter.apply(aType);
    }
}
```

```java
/**
 * Provides a list of FederatedEntityResolvers that can be called in a RuntimeWiring to
 * turn a GraphQL service into part of a supergraph in Apollo Federation.
 */
@Component
public class FederatedEntityResolverFactory {
    @Autowired
    ATypeService aTypeService;
    @Autowired
    ATypeConverter aTypeConverter;

    public List<FederatedEntityResolver<?,?>> create() {
        //This is only needed if another service wants to implement AType in its schema
        List<FederatedEntityResolver<?, ?>> entityResolvers = List.of(
                new FederatedEntityResolver<Integer, ATypeDTO>("AType", id -> {
                    AType aType = aTypeService.getATypeById(id);
                    ATypeDTO aTypeDto = aTypeConverter.apply(aType);
                    return aTypeDto;
                }) {
                }
        );
        return entityResolvers;
    }
}
```

B.3 Subgraph B/BTypeService

Listing B.6: application.properties

```
spring.application.name:BTypeService
spring.cache.type:NONE
server.port:8081

graphql:
    servlet:
        //async-mode-enabled:false
        tracing-enabled:true
```

Listing B.7: ATypesDataFetcher.java

```
@Slf4j
@Component
/**
 * Fetches an ATypeDTO.
 */
public class ATypeDataFetcher implements DataFetcher<ATypeDto> {

    @Autowired
    private BTypeService bTypeService;


    @Override
    public ATypeDto get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        log.info("Fetching ATypeDto at {}", new Date(System.currentTimeMillis()));
        BTypeDto bTypeDto = dataFetchingEnvironment.getSource();
        return bTypeService.getRelatedATypeByID(bTypeDto);
    }
}
```

Listing B.8: BTypeDataFetcher.java

```
@Slf4j
@Component
/**
 * Resolves the root query "lookupBType(id: ID!)".
 */
public class BTypeDataFetcher implements DataFetcher<BTypeDto> {

    @Autowired
    private BTypeConverter bTypeConverter;
    @Autowired
    private BTypeService B_service;

    @Override
    public BTypeDto get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        log.info("Fetching data from BType in SubgraphB at {}", new Date(System.
    currentTimeMillis()));
        String id = dataFetchingEnvironment.getArgument("id");
        BType bType = B_service.getBTypeById(Integer.parseInt(id));
        return bTypeConverter.apply(bType);
    }
}
```

Listing B.9: BTypesDataFetcher.java

```java
@Slf4j
@Component
/**
 * Resolves the root query "fetchAllBTypes" or the AType data field "relatedObjectsInSubgraphB".
 */
public class BTypesDataFetcher implements DataFetcher<List<BTypeDto>> {

    @Autowired
    private BTypeConverter bTypeConverter;
    @Autowired
    private BTypeService bTypeService;

    @Override
    public List<BTypeDto> get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        List<BType> bTypes;
        if (dataFetchingEnvironment.getSource() instanceof ATypeDto) {
            bTypes = bTypeService
                    .getRelatedBTypesById(((ATypeDto) dataFetchingEnvironment
                            .getSource())
                            .getId());
        } else {
            log.info("Fetching BTypes from Subgraph B at {}", new Date(System.currentTimeMillis()));
            bTypes = bTypeService.getAllBTypes();
        }
        return bTypes == null ? new ArrayList<>() : bTypes.stream()
                .map(bType -> bTypeConverter.apply(bType))
                .collect(Collectors.toList());
    }
}
```

```java
@Component
/**
 * DataFetcher for the Mutation Operation in Subgraph B.
 */
public class CreateBTypeDataFetcher implements DataFetcher<BTypeDto> {

    @Autowired
    private BTypeService B_service;
    @Autowired
    private BTypeConverter bTypeConverter;

    @Override
    public BTypeDto get(DataFetchingEnvironment environment) throws Exception {
        String name = environment.getArgument("appearsFirstIn");
        boolean isVowel = environment.getArgument("isVowel");
        int relatedObjectInSubgraphA = environment.getArgument("relatedObjectInSubgraphA");

        var newBType = B_service.create(name, isVowel, relatedObjectInSubgraphA);

        return bTypeConverter.apply(newBType);
    }
}
```

Listing B.10: RelatedObjectsOfSameTypeDataFetcher.java

```java
@Component
/**
 * DataFetcher that resolves the field "relatedObjectOfSameType" in a UnrelatedObjectInSubgraphB.
 */
public class RelatedObjectsOfSameTypeDataFetcher implements DataFetcher<List<UnrelatedTypeInSubgraphB_DTO>> {

    @Autowired
    private BTypeService bTypeService;

    @Override
    public List<UnrelatedTypeInSubgraphB_DTO> get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        UnrelatedTypeInSubgraphB_DTO unrelatedTypeInSubgraphB_dto = dataFetchingEnvironment
                .getSource();
        List<UnrelatedTypeInSubgraphB_DTO> relatedObjects = bTypeService
                .getRelatedUnrelatedTypes(unrelatedTypeInSubgraphB_dto.getId());
        return relatedObjects;
    }
}
```

Listing B.11: UnrelatedTypeInSubgraphBDataFetcher.java

```java
@Slf4j
@Component
/**
 * Resolves the root query "lookupUnrelatedTypeInSubgraphB(id: ID!)".
 */
public class UnrelatedTypeInSubgraphBDataFetcher implements DataFetcher<UnrelatedTypeInSubgraphB_DTO>{
    @Autowired
    BTypeService bTypeService;

    private final ExecutorService executorService = Executors.newFixedThreadPool(
            Runtime.getRuntime().availableProcessors()
    );

    public UnrelatedTypeInSubgraphB_DTO get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        String id = dataFetchingEnvironment.getArgument("id");
        UnrelatedTypeInSubgraphB_DTO unrelated = bTypeService.getUnrelatedType(Integer.parseInt(id));
        log.info("Fetching UnrelatedTypeInSubgraphB: ", unrelated.getId());
        return unrelated;
    }


}
```

Listing B.12: UnrelatedTypesInSubgraphBDataFetcher.java

```java
@Slf4j
@Component
/**
 * Resolves the root query "fetchAllUnrelatedObjectsInSubgraphB".
 */
public class UnrelatedTypesInSubgraphBDataFetcher implements DataFetcher<List<
    UnrelatedTypeInSubgraphB_DTO>> {

    @Autowired
    private BTypeService bTypeService;

    @Override
    public List<UnrelatedTypeInSubgraphB_DTO> get(DataFetchingEnvironment
    dataFetchingEnvironment) throws Exception {
        if(dataFetchingEnvironment.getSource() == null) throw new Exception("Source cannot be
    null.");
        log.info("Fetching UnrelatedTypes from Subgraph B at {}", new Date(System.
    currentTimeMillis()));
        return bTypeService.getAllUnrelatedTypes();
    }
}
```

Listing B.13: FederatedEntityResolverFactory.java

```java
@Component
public class FederatedEntityResolverFactory {

    @Autowired
    BTypeService bTypeService;
    @Autowired
    BTypeConverter bTypeConverter;

    public List<FederatedEntityResolver<?, ?>> create() {
        //This is only needed if this service wants to extend AType in its schema
        List<FederatedEntityResolver<?, ?>> entityResolvers = List.of(
                new FederatedEntityResolver<Integer, ATypeDto>("AType", id -> {
                    List<BType> relatedBTypes = bTypeService.getRelatedBTypesById(id);
                    if (relatedBTypes == null) {
                        return new ATypeDto(id);
                    }
                    List<BTypeDto> bTypeDtos = relatedBTypes.stream()
                            .map(u -> bTypeConverter.apply(u)).collect(Collectors.toList());
                    return new ATypeDto(id, bTypeDtos);
                }) {
                },
                //This code snipped is the only code added to Subgraph B in order for Subgraph
    C to extend B
                new FederatedEntityResolver<Integer, BTypeDto>("BType", id -> {
                    BType bType = bTypeService.getBTypeById(id);
                    return bType == null ? null : bTypeConverter.apply(bType);
                }) {
                }
        );
        return entityResolvers;
    }
}
```

B.4 Subgraph C/CTypeService

Listing B.14: application.properties

```
spring.application.name:C_service
spring.cache.type:NONE
server.port:8082


graphql:
    servlet:
        //async-mode-enabled:false
        tracing-enabled:true
```

Listing B.15: ATypeDataFetcher.java

```
@Component
@Slf4j
public class ATypeDataFetcher implements DataFetcher<ATypeDto> {

    @Autowired
    private CTypeService cTypeService;


    @Override
    public ATypeDto get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        log.info("Fetching ATypeDTO at {}", new Date(System.currentTimeMillis()));
        CTypeDto cTypeDto = dataFetchingEnvironment.getSource();
        return new ATypeDto(cTypeService.getRelatedATypeByID(cTypeDto.getId()), cTypeDto);
    }
}
```

Listing B.16: BTypeDataFetcher.java

```
@Component
@Slf4j
public class BTypeDataFetcher implements DataFetcher<BTypeDto> {
    @Autowired
    private CTypeService cTypeService;
    @Autowired
    private CTypeConverter cTypeConverter;


    @Override
    public BTypeDto get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        log.info("Fetching BTypeDto at {}", new Date(System.currentTimeMillis()));
        CTypeDto cTypeDto = dataFetchingEnvironment.getSource();
        int relatedBType = cTypeService.getRelatedBTypeByID(cTypeDto.getId());
        List<CType> cTypes = cTypeService
                .getRelatedCTypesById(relatedBType);
        if (cTypes != null) {
            return new BTypeDto(relatedBType, cTypes.stream()
                    .map(c -> cTypeConverter.apply(c))
                    .collect(Collectors.toList()));
        }

        return new BTypeDto(relatedBType, new ArrayList<>());
    }
}
```

```java
@Component
public class CreateCTypeDataFetcher implements DataFetcher<CTypeDto> {

    @Autowired
    private CTypeService cTypeService;
    @Autowired
    private CTypeConverter cTypeConverter;

    @Override
    public CTypeDto get(DataFetchingEnvironment environment) throws Exception {
            String name = environment.getArgument("appearsFirstIn");
            String isDefinedIn = environment.getArgument("isDefinedIn");
            int relatedObjectInSubgraphA = environment.getArgument("relatedObjectInSubgraphA");
            int relatedObjectInSubgraphB = environment.getArgument("relatedObjectInSubgraphB");
            CType newCType = cTypeService.create(name, isDefinedIn, relatedObjectInSubgraphA,
    relatedObjectInSubgraphB);
            return cTypeConverter.apply(newCType);

    }
}
```

Listing B.17: CTypeDataFetcher.java

```java
@Component
@Slf4j
public class CTypeDataFetcher implements DataFetcher<CTypeDto> {

    @Autowired
    private CTypeConverter cTypeConverter;
    @Autowired
    private CTypeService cTypeService;

    @Override
    public CTypeDto get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {
        if (dataFetchingEnvironment.getSource() instanceof ATypeDto) {
            CType relatedCType = cTypeService
                    .getRelatedCTypeById(((ATypeDto) dataFetchingEnvironment.getSource()).getId
    ());
            return relatedCType == null ? null : cTypeConverter.apply(relatedCType);
        }
        log.info("Fetching data from CType in SubgraphB at {}", new Date(System.
    currentTimeMillis()));
        String id = dataFetchingEnvironment.getArgument("id");
        CType cType = cTypeService.getCTypeById(Integer.parseInt(id));
        return cTypeConverter.apply(cType);
    }
}
```

Listing B.18: CTypesDataFetcher.java

```java
@Slf4j
@Component
public class CTypesDataFetcher implements DataFetcher<List<CTypeDto>> {

    @Autowired
    private CTypeService cTypeService;
    @Autowired
    private CTypeConverter cTypeConverter;
    @Override
```

```
    public List<CTypeDto> get(DataFetchingEnvironment environment) throws Exception {
        log.info("Fetching List<CTypeDTO> from CTypesDataFetcher at {}", new Date(System.currentTimeMillis()));
        if (environment.getSource() instanceof BTypeDto) {
            List<CType> result = cTypeService.getRelatedCTypesById(((BTypeDto) environment.getSource()).getId());
            return result == null ? new ArrayList<>() : result.stream()
                    .map(c -> cTypeConverter.apply(c))
                    .collect(Collectors.toList());
        }
        List<CTypeDto> cTypes = cTypeService.getAllCTypes().stream()
                .map(c -> cTypeConverter.apply(c))
                .collect(Collectors.toList());
        return cTypes == null ? new ArrayList<>() : cTypes;
    }
}
```

Listing B.19: FederatedEntityResolverFactory.java

```
@Component
public class FederatedEntityResolverFactory {

    @Autowired
    CTypeService cTypeService;
    @Autowired
    CTypeConverter cTypeConverter;

    public List<FederatedEntityResolver<?,?>> create() {
        //This is only needed if this service wants to extend BType in its schema
        List<FederatedEntityResolver<?, ?>> entityResolvers = List.of(
                new FederatedEntityResolver<Integer, ATypeDto>("AType", id -> {
                    CType relatedCType= cTypeService.getRelatedCTypeById(id);
                    if (relatedCType == null) {
                        return new ATypeDto(id);
                    }
                    return new ATypeDto(id, cTypeConverter.apply(relatedCType));
                }) {
                },
                new FederatedEntityResolver<Integer, BTypeDto>("BType", id ->{
                    List<CType> relatedCTypes = cTypeService.getRelatedCTypesById(id);
                    if (relatedCTypes == null) {
                        return new BTypeDto(id);
                    }
                    List<CTypeDto> cTypeDtos = relatedCTypes.stream()
                            .map(u -> cTypeConverter.apply(u))
                            .collect(Collectors.toList());
                    return new BTypeDto(id, cTypeDtos);
                }) {}
        );
        return entityResolvers;
    }
}
```