Graduado en Ingeniería Informática

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros Informáticos

TRABAJO FIN DE GRADO

An Algorithm to Normalize GraphQL Queries

Autor: Juan Garrido Ruiz
Director: Manuel M. Nieto Rodríguez

MADRID, ENERO DE 2020
ÍNDICE

1. Introduction .................................................................................................................. 1
   1.1. Background ............................................................................................................. 1
   1.2. Motivation .............................................................................................................. 1
   1.3. Research questions ............................................................................................... 2
   1.4. Delimitations .......................................................................................................... 2

2. Theory ................................................................................................................................ 3
   2.1. GraphQL .................................................................................................................. 3
       2.1.1. GraphQL schema ............................................................................................. 4
       2.1.2. GraphQL queries .............................................................................................. 5
       2.1.3. Rewriting rules ................................................................................................. 6

3. Method ............................................................................................................................. 12

4. Algorithm ......................................................................................................................... 13
   4.1. function Main ........................................................................................................... 15
   4.2. Function modify_query ........................................................................................... 16
   4.3. Function Rule8 ......................................................................................................... 18
   4.4. Function Rule3 ......................................................................................................... 20
   4.5. Function Rule4 ......................................................................................................... 23
   4.6. Function Rule7 ......................................................................................................... 25
   4.7. Function Rule2 ......................................................................................................... 27
   4.8. Function Rule1 ......................................................................................................... 29
   4.9. Function treeFunction ............................................................................................. 30

5. Discussion ......................................................................................................................... 32
   5.1. Results ...................................................................................................................... 32
   5.2. Method ...................................................................................................................... 33
   5.3. Sources ...................................................................................................................... 34

6. Conclusion .......................................................................................................................... 35
   A. Tests ............................................................................................................................... 38
LISTINGS

1. Example of a schema ........................................ 3
2. Example of query ........................................... 3
3. Result from the query ...................................... 4
4. Example query ............................................ 4
5. Example schema file ...................................... 4
6. Example query ............................................ 5
7. Example query ............................................ 6
8. Example query ............................................ 8
9. Result query Listing 2.9 after apply rule4 ................. 8
10. Example query ........................................... 9
11. Result query Listing 2.11 after apply rule5 ............... 9
12. Example query ........................................... 9
13. Result query Listing 2.13 after apply rule7 ............... 9
14. Example query ........................................... 10
15. Result query Listing 2.15 after apply rule8 ............... 10
16. Example query ........................................... 10
17. Result query Listing 2.7 ................................ 11
18. Example query ........................................... 30
19. Example query ........................................... 31
20. Schema file ............................................. 38
21. Query1 ............................................. 38
22. Result query1 ........................................... 39
23. Query2 ............................................. 39
24. Result query2 ........................................... 39
25. Query3 ............................................. 40
26. Result query3 ........................................... 40
27. Query4 ............................................. 40
28. Result query4 ........................................... 41
29. Query5 ............................................. 41
30. Result query5 ........................................... 41
31. Query6 ............................................. 42
32. Result query6 ........................................... 42
33. Query7 ............................................. 43
34. Result query7 ........................................... 43
35. Query8 ............................................. 44
36. Result query8 ........................................... 44
37. Query9 ............................................. 44
38. Result query9 ........................................... 45
39. Query10 ............................................. 45
40. Result query10 .......................................... 45
41. Query11 ............................................. 46
42. Result query11 .......................................... 46
<table>
<thead>
<tr>
<th></th>
<th>Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>Query12</td>
<td>Result query12</td>
</tr>
<tr>
<td>44</td>
<td>Query13</td>
<td>Result query13</td>
</tr>
<tr>
<td>45</td>
<td>Query14</td>
<td>Result query14</td>
</tr>
<tr>
<td>46</td>
<td>Query15</td>
<td>Result query15</td>
</tr>
<tr>
<td>47</td>
<td>Query16</td>
<td>Result query16</td>
</tr>
<tr>
<td>48</td>
<td>Query17</td>
<td>Result query17</td>
</tr>
<tr>
<td>49</td>
<td>Query18</td>
<td>Result query18</td>
</tr>
<tr>
<td>50</td>
<td>Query19</td>
<td>Result query19</td>
</tr>
<tr>
<td>51</td>
<td>Query20</td>
<td>Result query20</td>
</tr>
<tr>
<td>52</td>
<td>Query21</td>
<td>Result query21</td>
</tr>
<tr>
<td>53</td>
<td>Query22</td>
<td>Result query22</td>
</tr>
<tr>
<td>54</td>
<td>Query23</td>
<td>Result query23</td>
</tr>
<tr>
<td>55</td>
<td>Query24</td>
<td>Result query24</td>
</tr>
<tr>
<td>56</td>
<td>Query25</td>
<td>Result query25</td>
</tr>
<tr>
<td>57</td>
<td>Query26</td>
<td>Result query26</td>
</tr>
<tr>
<td>58</td>
<td>Query27</td>
<td>Result query27</td>
</tr>
<tr>
<td>59</td>
<td>Query28</td>
<td>Result query28</td>
</tr>
<tr>
<td>60</td>
<td>Query29</td>
<td>Result query29</td>
</tr>
<tr>
<td>61</td>
<td>Query30</td>
<td>Result query30</td>
</tr>
<tr>
<td>62</td>
<td>Schema file</td>
<td></td>
</tr>
</tbody>
</table>

III
Resumen

En 2012, Facebook estaba desarrollando su aplicación para móviles. A medida que la aplicación iba adquiriendo un mayor grado de complejidad, esta empezaba cada vez más a sufrir errores y su comportamiento dejaba bastante que desear. Después de evaluar diferentes opciones para su API, decidieron crear lo que conocemos actualmente como GraphQL. Tras tres años de uso interno, Facebook, decidió publicar la especificación. A medida que fue pasando el tiempo, GraphQL se hizo cada vez más popular llegando hasta el punto de que grandes empresas como Netflix, Twitter, Airbnb, Amazon o Github la incluyeron rápidamente en sus proyectos. Debido al gran crecimiento de GraphQL, esta tecnología aún tiene algunos problemas que tienen que estudiarse para proceder a su posterior corrección. Uno de estos problemas es que no existe un estándar de creación de queries. Como una petición puede ser creada de muchas formas distintas esto conlleva a que los tiempos de ejecución sean siempre distintos y algunas queries contengan algunos campos repetidos o innecesarios. Este proyecto de fin de grado intenta solucionar este problema creando un algoritmo capaz de reescribir todas las queries a un estándar, produciendo la query más óptima que se pueda crear para esa petición. Estandarizando las queries solucionaremos este problema y aquello llamado "field collection" no será necesario de ejecutar.

Abstract

In 2012, Facebook was rebuilding their Facebook’s native mobile applications. As Facebook’s mobile apps became more complex, their mobile app suffered poor performance and it frequently crashed. After evaluating different options to implement an API data (including RESTful server resources) they decided to create what we know nowadays as GraphQL. After three years of internal usage, in 2016, Facebook released the specification of this framework. This framework has become popular to the point that big companies as Netflix, Twitter, Airbnb, Amazon or Github quickly included this framework in their projects. In spite of all the advances that this technology has brought, it has some problems derived from the lack of a formal form to create the queries. As one query can be written in so many different forms, responses times are affected producing different times even though the queries are asking for the same. This thesis attempts to solve this problem by creating an algorithm capable to rewire the queries in order to normalize them. Normalizing the queries we will solve this problem and what is called "field collection" will not be needed.
1. INTRODUCTION

1.1. Background

On 2016, Facebook released the specification and a reference implementation from the framework that they were using internally for three years. This framework was used in their mobile app due to all the problems that REST gave them. They discover that they could not implement REST in their app. It was slow on the network, because so many times they needed so many trips to the API for have the completely result, and fragile, the client/server relationship was fragile in terms of changes to the API needed to be carefully carried over the client code or the app could just crash. After decided that they were not going to implement REST in their app they started to think about another alternative, creating afterwards what we know as GraphQL. GraphQL consisted in a query language for APIs with a new type of Web-based data access interfaces that presents an alternative to the notion of REST-based interfaces. GraphQL needs one single call to retrieve exactly the data that you are asking for. This framework replace REST multiple requests. GraphQL also presents a clear description of the data in the API and gives the opportunity to ask for what you want and nothing else. Nowadays GraphQL has a big and active community and is available in dozen of different languages that are managed by different community members. Companies like Netflix, Twitter, Airbnb, Amazon or Github quickly included this framework within their projects and the community is also worldwide, having meetups on every continent and conferences around all the world.

1.2. Motivation

In the paper "Semantics and Complexity of GraphQL" [1], Olaf Hartig and Jorge Pérez shows that GraphQL has something called "field collection". This field collection merges all fields that are repeated inside a query result. That means that a field is not going to be twice in the same level of a query result.

In the paper [1], Olaf Hartig and Jorge Pérez propose seven rules with which you can rewrite every query in GraphQL such that for the resulting query there is no need to perform field collection. Both queries are semantically the same so they will produce the same result. Every GraphQL query can be rewritten into a new query that is not redundant and is in ground-typed normal form. The aim of this thesis project is to implement an algorithm to normalize GraphQL queries. As queries in GraphQL can be created in so many different ways and one query can be written in several forms this causes different responses times in queries even though the queries are asking for the same. Normalizing GraphQL queries will avoid the execution of field collection avoiding that extra time that this functionality takes.

After some months of work, one new rule was added to the project. Instead of use the last seven rules mentioned in the paper [1], a total of eight rules were used, seven rules plus the last added.
1.3. **Research questions**

1. How can a GraphQL query be normalized in practice? What algorithm can be used for this purpose?

2. What are the properties of this algorithm?

The goal of this thesis project is to optimize GraphQL queries by developing an algorithm that can rewrite a query to a new query in a normal form. This algorithm will be developed using eight rules that after a correct use of them will generate a new non-redundant query in ground-typed form.

1.4. **Delimitations**

As the project only focuses on rewriting queries, some delimitations have to be introduced:

1. As the aim of this project is to rewrite queries to the normal form, the implementation will assume that the schema file is correct and there are not errors in it.

2. As the tool has to rewrite queries, python will be the language that is going to be used in order to achieve the objective.

3. As the tool has to be tested and tests are needed, me, as the developer of the tool, will create all tests that I consider necessary to achieve the goal.
2. THEORY

In this section I will describe and give an overview about GraphQL.

2.1. GraphQL

GraphQL is a query language for your API. If you want to fetch data from for example LinkedIn API [2] you will ask to the API for the information that you want. Normally the way those typically works is that you have a specific endpoint or URL that you are asking and that endpoint determines what data comes back. When you use a REST API you fetch an URL and that URL is returning a JSON object full of data. GraphQL aims to improve upon those sort of ideas, instead of an API where you hit an URL and just accept whatever data that the URL returns back, GraphQL allows you to ask for a specific type of data. In the following example I explain how we ask for an specific kind of data:

Listing 1: Example of a schema

```typescript
type Project{
    name: String
    tagline: String
    contributors: [User]
}

type User{
    id: ID
    name: String
    age: Int
    gender: String
    email: String
    height: Float
    city: String
}

type Query {
    project(name: String): Project
    user(id: ID): User
}
```

Here we defined the way our data looks. I defined an object type project with a name that is a string, a tagline which is an string and a contributor that has to be of type User. Having the description of the data we can start asking for the content based on the schema:

Listing 2: Example of query

```typescript
project(name: "Thesis"){
    tagline
}
```

In GraphQL you can pick selective information just specifying it. For example in this GraphQL query we are asking for the tagline of a project with the name "Thesis". Then the result comes back with the exactly answer that you were asking for:
GraphQL returns an object with the project and a tagline of type string that says "An algorithm to Normalize GraphQL Queries".

GraphQL provides a complete and an understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time. This framework can retrieve exactly the data you need from the API with a single call replacing multiple REST requests.

For example going back to the example of the REST-based LinkedIn API, if you want to know all the information from a user, first you will receive a list with all the users id. The list contains the id from the user that we are asking for. Knowing this id we will go back to the API and ask for the user information using the id obtained in the previous request. With GraphQL we only need to ask one time to the API for the information that we want from the id that we specify.

2.1.1. GraphQL schema

The schema file determines which kind of operations and types our API will have. In this file we determine the different fields that our API will have and the relation that these fields have with other fields. In the following schema file we define three types, Airplane, Human and Alien. The interface Character is implemented by the types Alien and Human. Character defines four fields that are id, name, friends and age. These four fields are a property from the Character that are implemented as Alien and Human:

Listing 5: Example schema file

```graphql
interface Character {
    id: ID
    name: String
    friends: [Character]
    age: Int
}
```
type Alien implements Character {
  id: ID
  name: String
  friends: [Character]
  age: Int
}

type Human implements Character {
  id: ID
  name: String
  friends: [Character]
  age: Int
}

type AirPlane{
  id: ID
  name: String
  length: Float
}

type Query {
  getCharacter(id: ID): Character
  getAlien(id: ID): Character
  getHuman(id: ID): Character
  getAirplane(id: ID): AirPlane
}

schema {
  query: Query
}

Interface: Interfaces in GraphQL contains a list of named fields with their arguments. These interfaces can be then implemented by object types which will have all fields defined in the interface. Fields on an interface work the same as fields on an object type.

Query Type: The Query type is supposed to be the query root operation type of the schema. The fields on the query type shows which fields are at the top level on a GraphQL query.

2.1.2. GraphQL queries

GraphQL queries are strings that are represented in a Json like format. GraphQL queries are used to communicate with the API. In the query you specify the data that you want from the server and the server returns the answer to that request using a Json like format. A GraphQL query has the following aspect:

Listing 6: Example query

{
The last example shows how GraphQL queries look like and how they are structured in layers. The root object in the previous example is "getAlien(id: 45)" and inside it, we specify that we want the Alien's name and the age. We can ask for "getAlien(id: ID)" because the query type in the schema file is defined with this field and its argument which is Character. This part will be inside layer 1. We can see that we have a selection set called "friends" that ask for the name and the age from all the friends of Alien which id is 45. The fields inside the selection set will be in layer 2.

Listing 7: Example query

```graphql
getAlien(id: 45) {
  name
  age
  friends{
    name
    age
  }
}
```

This example shows a query which is asking for the friends of the Alien which id number is 45. This query also specifies that it must return the name and the age of all those friends that are of type Alien. We have to notice that we can specify the Character’s type using the expression on t{ ... }.

2.1.3. Rewriting rules

In the paper "Semantics and Complexity of GraphQL" [1], Hartig and Pérez defined what ground-typedness is. They defined that if a GraphQL query \( \varphi \) satisfies the following grammar this \( \varphi \) is in ground-typed normal form:

\[
\varphi ::= \psi \ | \ \chi \ | \ \varphi \ \psi \\
\psi ::= \text{on } t \ \{ \ \chi \ \}
\chi ::= f[\alpha] \ | \ \ell : f[\alpha] \ | \ f[\alpha] \ { \ \varphi \ } \ | \ \ell : f[\alpha] \ { \ \varphi \ }
\]

\( \ell : f[\alpha] \): This construction captures the notion of "field aliases". This aliases are used to rename the field names that appear in the query result.

\( f[\alpha] \): This construction is the most basic expression. The expression "f" belongs to fields and \( \alpha \) represents a partial mapping from Arguments to Vals. \( \alpha \) can be empty and for this case we will just write f.
The previous information comes from the paper "Semantics and Complexity of GraphQL" [1].

Query $\varphi$ is in ground-typed normal form only if the following conditions are satisfied:

1. $t \in O_t$ for every expression of the form on $t \{ ... \}$ in a query $\varphi$. Being $O_t$ those Object Types which are defined in the schema file using the keyword type.

2. The expressions with the form on $t \{ ... \}$ do not appear mixed with another selection set.

3. An expression of the form on $t \{ \}$ does not appear within another on $t \{ \}$ expression.

Also, they defined in [1] what a GraphQL query $\varphi$ non-redundant is. $\varphi$ is non-redundant if it satisfies that for every sub-expression in $\varphi$ of the form $\varphi_1 \ldots \varphi_k$ there are no indexes $i, j \in \{1, ..., k\}$ such that $i \neq j$ and

- $\varphi_i = \varphi_j = f[\alpha]$, or
- $\varphi_i = \varphi_j = \ell : f[\alpha]$, or
- $\varphi_i = f[\alpha]\{ \beta \}$ and $\varphi_j = f[\alpha]\{ \gamma \}$, or
- $\varphi_i = \ell : f[\alpha]\{ \beta \}$ and $\varphi_j = f[\alpha]\{ \gamma \}$, or
- $\varphi_i = \text{on } t \{ \beta \}$ and $\varphi_j = \text{on } t \{ \gamma \}$.

For every query $\varphi$ that adapts to a schema $S$ there exists a non-redundant query $\varphi'$ that is in ground-typed normal form such that $\varphi \equiv \varphi'$. This proof can be obtained applying the rules defined on their paper [1].

Hartig and Pérez shows that applying the following rules copied from their paper [1] it is possible to normalize a query:

1. if $\chi = f[\alpha]$ or $\chi = \ell : f[\alpha]$ then
   $\chi \varphi_1 \ldots \varphi_i \chi \varphi_{i+1} \ldots \varphi_k \equiv \chi \varphi_1 \ldots \varphi_i \varphi_{i+1} \ldots \varphi_k$

2. if $\chi = f[\alpha]$ or $\chi = \ell : f[\alpha]$ then
   $\chi \{ \beta \} \varphi_1 \ldots \varphi_i \chi \{ \gamma \} \varphi_{i+1} \ldots \varphi_k \equiv \chi \{ \beta \} \varphi_1 \ldots \varphi_i \varphi_{i+1} \ldots \varphi_k$

3. on $t \{ \varphi_i \ldots \varphi_k \} \equiv$ on $t \{ \varphi_1 \} \ldots$ on $t \{ \varphi_k \}$

4. on $t \{ \text{ on } t \{ \varphi \} \} \equiv$ on $t \{ \varphi \}$

5. if $\text{implements}_S(t) = [t_1 \ldots t_k]$ then
   on $t \{ \varphi \} \equiv$ on $t_1 \{ \varphi \} \ldots$ on $t_k \{ \varphi_1 \}$

6. if $\text{union}_S(t) = [t_1 \ldots t_k]$ then
   on $t \{ \varphi \} \equiv$ on $t_1 \{ \varphi \} \ldots$ on $t_k \{ \varphi_1 \}$
7. if \( t_1, t_2 \in O_t \) and \( t_1 \neq t_2 \) then
\[
\{ \text{on } t_1 \{ \text{on } t_2 \{ \varphi \} \} \} \equiv \varepsilon
\]

8. if \( \text{type}_S(f) = t \) or \( \text{type}_S(f) = L_t \) and \( \chi = f[\alpha] \) or \( \chi = l:f[\alpha] \) then
\[
\chi \{ \varphi \} \equiv \chi \{ \text{on } t \{ \varphi \} \}
\]

The aim of this project is to create a process that ensures the normalization of all the queries using the eight rules.

- Rule 1 removes those fields that are repeated in the same layer.
- Rule 2 merges the fields that are inside a selection set that is repeated in the same layer.
- Rule 3 takes each field that is within an "on t" type selection set and separates them to facilitate the application of rule 4 and rule 7 later.
- Rule 4 merges those fields that are within an "on t" type selection set that is within another "on t" type selection set sharing the same type "t". The following example shows how a query can be rewritten to ground-typed normal form just applying rule 4:

Listing 8: Example query

\[
\{ \\
on \text{Alien}\{ \\
on \text{Alien}\{ \text{name} \} \\
} \\
\}
\]

The query is asking for the name of the Alien. Here we can see that on Alien is repeated twice and is nested (on Alien\{ name \} is inside on Alien\{ \}). After the apply rule 4 we will have a query in ground-typed normal form without redundancies. As we see the following query is the result of the application of rule 4 and removed the nested selection set on Alien\{ \}.

Listing 9: Result query Listing 2.9 after apply rule 4

\[
\{ \\
on \text{Alien}\{ \text{name} \} \\
} \\
\]

- Rule 5 replace the interface type for all those types that implements an interface. The following example shows how a query can be rewritten to ground-typed normal form just applying rule 5:
Having a query with a field of the form on t and being t the interface type Character. Rule5 replace this type for all the types that implements Character. Here we are asking for the Character’s name and as the schema field defined different types of Character, after the application of rule5, this rule will replace the Character type for all those types of Characters having the following result:

Listing 11: Result query Listing 2.11 after apply rule5

```graphql
{ 
  on Human{
    name
  }
  on Alien{
    name
  }
}
```

- Rule7 removes all the selection set of type "on t", being the type of this selection set t1, that are inside another selection set of type "on t" but being the type of this selection set t2. T1 and t2 are different types.

Listing 12: Example query

```graphql
{ 
  on Alien{ 
    on Human{ name } 
  }
}
```

In this example the query asks for the Human’s name in the Alien Character. Rule7 returns an empty query due to there is a nest and the object types are different.

Listing 13: Result query Listing 2.13 after apply rule7

```graphql
{ 
}
```

- Rule8 takes the type that corresponds to the field f that has in the schema, in the previous schema file we have Character as type, and the rule creates a new query introducing the selection set "on t" (in this case t is Character) and all the fields that were inside the previous selection set.
Listing 14: Example query

```
{
  friends{
    on Alien{ name }
    id
  }
}
```

This query is asking for the friends' id and for the name of those friends that have type Alien. Rule8 introduces the interface type from the schema file as on Character and all the fields that were inside the previous selection set.

Listing 15: Result query Listing 2.15 after apply rule8

```
{
  friends{
    on Character{
      on Alien{ name }
      id
    }
  }
}
```

At the beginning they were a total of eight rule but due to the short time I could not implement rule number 6. For this reason I did not add this rule to this section.

The following example presents a query with some redundancies:

Listing 16: Example query

```
{
  getAlien(id: 45){
    friends{
      name
    }
    name
    friends{
      id
    }
  }
}
```

This query has a selection set called friends that is repeated twice and will ask twice for friends. This query is a valid query in GraphQL but it has redundancies. Applying the second rule, we can remove those redundancies in the selection set, having the following query as a result:
Listing 17: Result query Listing 2.7

```graphql
{
  getAlien(id: 45){
    friends{
      name
      id
    }
    name
  }
}
```

Finally after apply the rules we obtain a non-redundant query in ground-typed form.
3. METHOD

This section aims to describe the methods applied and the process followed to address the research questions.

The first step was the analysis process. The intention of this process was to define the requirements needed for the correct implementation of the thesis project. This analysis process was done previously the implementation. The process started by reading the GraphQL specification [3] and the paper "Semantics and complexity of GraphQL" [1]. GraphQL specification showed me how the structure of a GraphQL query was.

After understanding the main ideas of this thesis project the question of how implement the algorithm in order to normalize GraphQL queries appeared. To answer the first part of the first research question, "How can a GraphQL query be normalized in practice?", reading the article [1] we can see that thanks to a series of rules, in total eight, we can normalize all queries. By applying these rules using them with different orders, we can normalize the queries. In summary, these rules do not have an order; but there are some that must be applied before others, so by using them in the correct order it is possible to normalize a query.

At this point another challenge, how to develop the algorithm, was found. The first idea was the use of a parser. The option was easy because there were already implemented parsers so it was not neccessary to develop it and the parser would give the chance to analyse the data in a simple way. After trying out different parsers, the output of any of these was not the expected one. These outputs were convoluted and they were not going to help developing the algorithm. Because of that the next idea was to design a parser to restructure the query generating a new query without redundancies and in normal form; but on the other hand this would take a long time.

Once the option of use a parser was discarded, another challenge appeared, which was how to decide the order of how the rules were going to be applied, this question was part of the research questions that were proposed previously. To answer the second part from the first research question, "What algorithm can be used for this purpose?", some tests were created and it was analysed how the rules would be applied. After some tests with redundacies, that were not in a normal form and their respective solutions, were written, a clear idea of how to solve the problem related to the application of the rules mentioned in the paper "Semantics and complexity of GraphQL" [1] was obtained.
4. ALGORITHM

This section explains the algorithm and shows also some pseudo code in order to define the algorithm.

As I exposed previously, the algorithm consists in the implementation of the seven rules mentioned before in the section 2.1.3 which with the correct order they can rewrite a query into the normal form. After finding a possible way to apply the rules and rewrite a GraphQL query, I developed a serie of tests with their own solutions and I started to implement the algorithm. This implementation was done in Python following the rules of this programming language [4]. The algorithm is the consequence of different modules separated in order to make the code readable. The algorithm receives an input and generates an output. The input must be a query and the path to the schema file. The query must have the form: query = """ hero{ friends{ name } friends{ id } } """. The query can have multiple spaces, line breaks or tabs. It also can have the braces separted from the selection set like: hero { or friends {. The only rule that this form has is that the close braces must be separated from the field, they cannot be together, for example: id}. The output generated will be the query rewritten to the normal form.

As the main part of the algorithm is the implementation of the seven rules, I implemented them in separate functions/modules. Each module is the implementation of each rule. The following figure shows how the algorithm works. First, main function is called. This function calls module rule2 and after that calls module modify_query passing as input the output from rule2. Module modify_query calls within the function rule8 and then rule3. Module rule3 calls module rule4 and rule7 using the output from rule4 as input for rule7. When modify_query ends, main function calls rule1 using the output from modify_query as input for rule1. When rule1 ends its execution the algorithm finishes returning the final output obtained from rule1 as a string. The module names are just orientative and, for example, for rule5 there is not a specific module, this is because this rule is applied in the module rule8 with the rule8.
4.1. function Main

Given a GraphQL query q and a schema file, rewrites query q into a query q’ that will be a query in the normal form. This function calls the rest of functions, including those ones that implement the seven rules mentioned in section 2.1.3.

Algorithm 1 Main(queryList, schema)

1: queryList = convertToList(q)
2: construct_types(schema)
3: finalQuery = []
4: auxList = []
5: equal = False
6: while equal == False do //1
7:   finalQuery = rule2(finalQuery)
8:   equal = finalQuery == rule2(finalQuery) // if finalQuery and the output from rule2 is the same, equal will be True
9: end while
10: finalQuery = modify_query(finalQuery, schema)
11: finalQuery = rule1(finalQuery)
12: return " ".join(finalQuery) // 2

convertToList: Splits a string into a list where each word is a list item.

construct_types: Saves on types, which is a global variable of type list, the interface types and those types that implements them. This function receives the schema file as input. It searches for those interface types that appears in the schema file and saves them in types. It also searches for those types which implements an interface type and saves them within the variable. Types is a list of lists which contains on each list an interface type (in the first position of the list) and those types that implements it.

1: If after apply twice rule2 the two outputs are the same, that means that there are not more redundacies, but, if the outputs are different that means that there are redundancies and the query needs to call again the function rule2. This is done until all redundancies are removed from the query.

2: Joins all the elements from the finalQuery list having a string as output.
4.2. Function modify_query

This function calls function rule8 and function rule3. It reverses the list passed as argument (queryList) just to start analyzing those selection sets that are more deep in the query. The aim of this function is to apply rule8 in those selection sets where fields of type "on { ... }" are.

Algorithm 2 modify_query(queryList,schema)

```python
finalQuery = queryList
2: pos = []
3: auxList = []
4: secondAuxList = []
5: thirdAuxList = []
6: reversedList = queryList[::-1] //reverses the order from the list. From [1,2,3] to [3,2,1]
7: for element in reversedList do
8:   if element.last == "{" and element.next != "on" then //it is a selection set
9:     pos = beginningEnd(element.index, reversedList)
10:    auxList = reversedList[pos[0]:pos[1]+1]
11:    if element.index == reversedList.length - 1 then
12:       break
13:     else if "on" is in auxList and element.prev.last != "{" then
14:        secondAuxList = rule8(finalQuery, schema, (queryList.length -1 -pos[0]))
15:        thirdAuxList = rule3(secondAuxList[(queryList.length-1-pos[0]) : (queryList.length -1 -pos[1])], schema)
16:       finalQuery.append(secondAuxList[:pos[0]]) //appends the elements from the beginning until the index of pos[0]
17:       finalQuery.append(thirdAuxList)
18:    end if
19:  end if
20: end for
21: finalQuery = rule8(finalQuery, schema, 0)
22: finalQuery = rule3(finalQuery, schema)
23: return finalQuery
```

element.index: Returns the index of the element within the reversedList.

element.last: Element is a string from the string list. This operation will take the last letter from the string. For example if the string is "friends{", element.last will return "{".

element.next: Returns the element that is located in the index + 1. If the list is: [1,2,3,4]
and the element is 2, element.next returns 3.

beginningEnd: It is a simple function that iterates the list from the position indicated, normally is the position of a selection set, until the position where the closing brace that closes that selection set it is. Returns a list of two elements, the first one is the index of the selection set and the second one is the index of the closing brace.
4.3. Function Rule8

Given a query q that was converted to a list in the function main, being this one the initial query passed as input, without changes in order to expand it and prepare it for the application of the rest of rules. This function implements the functionality from rule number eighth and rule number five mentioned in section 2.1.3. Receives the query and the schema path as inputs. Rule8 uses a library called difflib, which returns the word that matches with the word passed as argument.

Algorithm 3 Rule8(queryList, schema, position)

```python
finalQuery = []
object_types = []
file = open(schema,'r')
data = file.readlines()  # returns all lines in the file, as a list where each line is an item in the list object.
file.close()
field = queryList[position][:-1] + ":"  # 1
save = difflib.get_close_matches(field,data)
for word in types  # 3
    if type == word[0] then
        objectTypes = word[1:]  # saves all the elements from the list except the first one which is the interface type.
    end if
end for
for i in objectTypes do
    finalQuery = finalQuery.append("on" + " " + i + "{")
    finalQuery = finalQuery.append(queryList[pos[0]:pos[1]] + "")
end for
finalQuery.append("" )  # closes the selection set
for i in queryList[pos[1]:] do
    finalQuery = finalQuery.append("on" + " " + i + "{")
    finalQuery = finalQuery.append(queryList[pos[0]:pos[1]] + "")
end for
finalQuery.append("" )  # closes the selection set
```

difflib.get_close_matches: Returns the element that matches or is close to match to the element passed as first argument. The second argument passed as input is the string to compare. This function is needed because sometimes exact matches do not work due to some fields can be replaced by other fields. For example, "hero(episode: EMPIRE)" does not match in the schema file. For this reason exact matches does not work.

1: queryList[position][:-1] returns the string without the last element. For example,
being "friends{" the string, it will return "friends". Then it adds ":" to the string.

2: Returns the next string from the position given. For example, having "hero(episode: Episode): Character" as string and being "hero(episode: Episode):" the position given, it will return "Character".

3: types is a global variable which contains a list of lists with the interface type and the types that implements it. It is initialized in function main calling a function called construct_types(). For example, the appearance of this variable can be [[Character, Droid, Human]], being the first element the interface type and the rest those types that implements it.

4: Appends the elements from the beginning of the list until the position indicated + 1. It adds 1 because it also wants to add the field that is in position.

beginningEnd: It is a simple function that iterates the list from the position indicated, normally is the position of a selection set, until the position where the closing brace that closes that selection set it is. Returns a list of two elements, the first one is the index of the selection set and the second one is the index of the closing brace.
4.4. Function Rule3

Given a query list \( q \), being the output from rule8, separates each field with the form \( t \{ \varphi_1 \ldots \varphi_k \} \) into the following expression \( \equiv \) \( t \{ \varphi_1 \} \ldots t \{ \varphi_k \} \). Calls rule4 and rule7 passing as argument each subexpression separated before.

**Algorithm 4 Rule3(queryList, schema)**

1. \( \text{finalQuery} = [\] \)
2. \( \text{auxQuery} = [\] \)
3. \( \text{auxList} = [\] \)
4. \( \text{pos} = [\] \)
5. \( \text{First} = \text{False} \)
6. \( \text{countBraces} = 0 \)
7. \( \text{element} = 1 \)
8. \( \text{finalQuery.append(queryList[0])} \)
9. \( \text{while element} \leq \text{queryList.length} - 1 \) do
10. \( \text{if queryList[element] == "}" then} \)
11. \( \quad \text{if countBraces} > 1 \) then
12. \( \quad \quad \text{countBraces} = \text{countBraces} - 1 \)
13. \( \quad \quad \text{if checkType(element,queryList) == True then} \)
14. \( \quad \quad \quad \text{auxQuery} = \text{auxQuery[-2]} //\text{Removes two last elements from auxQuery} \)
15. \( \quad \quad \text{else} \)
16. \( \quad \quad \quad \text{auxQuery} = \text{auxQuery[-1]} //\text{Removes last element from auxQuery} \)
17. \( \quad \text{end if} \)
18. \( \quad \text{element} = \text{element} + 1 \) //\text{Removes last element from auxQuery} \)
19. \( \text{end if} \)
20. \( \text{else} \)
21. \( \quad \text{countBraces} = \text{countBraces} - 1 \)
22. \( \quad \text{First} = \text{False} \)
23. \( \quad \text{auxQuery} = [\] \)
24. \( \text{finalQuery.append("")} \)
25. \( \quad \text{element} = \text{element} + 1 \)
26. \( \text{end if} \)
Algorithm 4 Rule3(queryList,schema) (continued)

```plaintext
28:   else if queryList[element] == "on" and First == False then
        First == True

30:       countBraces = countBraces + 1
            auxQuery.append(queryList[element])

32:       auxQuery.append(queryList[element+1])
        finalQuery.append(queryList[element])

34:       finalQuery.append(queryList[element+1])

40:       finalQuery = finalQuery + auxList[2:] //Appends the elements from the index 2 until the end
            if auxList == [] then
                if queryList[element].last == "{" then
                    pos = beginningEnd(queryList[element:])
                    element = element + pos + 1
                    continue

                end if

            end if

42:            element = element + 1

44:            continue

else

46:                if queryList[element].last == "{" then
                    pos = beginningEnd(queryList[element:])
                    finalQuery = finalQuery + queryList[element:element+pos+1] //Appends the elements from the list except the two first ones
                    element = element + pos + 1

52:                    continue

else

54:                        finalQuery.append(queryList[element])
                        element = element + 1

56:                        continue

end if
```

60:   end if
Algorithm 4 Rule3(queryList, schema) (continued)

else if queryList[element] == "on" and First == True then

62: countBraces = countBraces + 1
64: auxQuery.append(queryList[element])
66: auxQuery.append(queryList[element+1])
68: element = element + 2 //we add two because we had already added the next

68: end if

70: continue

72: finalQuery.append("}
74: element = 0
76: auxList = finalQuery
78: finalQuery = []
80: while element <= auxList.length - 1 do
82: if auxList[element] == "on" and auxList[element+1].last == "{" and aux-
86: List[element+2].last == "}" then
88: element = element + 2
90: element = element + 1
92: end if
94: finalQuery.append(auxList[element])
96: end while
98: return finalQuery

checkType: Returns True in case the element is a selection set of type "on t{ ... }" and
false if it is a selection set of type, for example, "friends{". This function calls TreeFunc-

element.last: Element is a string from the string list. This operation will take the last
letter from the string. For example if the string is "friends{", element.last will return "}".

beginningEnd: It is a simple function that iterates the list from the position indica-
ted, normally is the position of a selection set, until the position where the closing brace
closes that selection set it is. Returns a list of two elements, the first one is the index
of the selection set and the second one is the index of the closing brace.
4.5. Function Rule4

Given a subexpression s that has the form "on t{ φ }" removes those redundancies of the form " on t{ on t{ φ } }" being "t" the object type and being both "t" the same type. Produces an output of the following form " on t{ φ } ".

Algorithm 5 Rule4(queryList)

```
finalQuery = []
2: first = False // True if it is inside a selection set of type on t{ ... }
savetype = ""
4: element = 0
    while element <= queryList.length - 1 do
6:        if queryList[element] == "on" and first == False then
7:            first == True
8:        savetype = queryList[element+1] //saves the object type
9:        finalQuery.append(queryList[element])
10:       finalQuery.append(queryList[element+1])
11:       element = element + 2
12:       continue // continues with the next iteration of the loop
13:    else if queryList[element] == "on" and first == True and savetype == queryList[element+1] then
14:        element = element + 2
15:        continue
16:    else if queryList[element] == "on" and first == True and savetype != queryList[element+1] then
17:        finalQuery.append(queryList[element])
18:        element = element + 2
19:        continue
20:    else if queryList[element].last == "{" and queryList[element] != "on" then // if it is a selection set
21:        finalQuery = finalQuery + queryList[element:] //Appends the rest of elements in queryList from the chosen element until the end of the list
22:        return finalQuery
23:    else if queryList[element].last != "{" and queryList[element] != "on" and queryList[element] != "}" then// if it is a field and not a selection set
24:        finalQuery.append(queryList[element])
25:    else if queryList[element] == ")" then
26:        finalQuery.append(queryList[element])
27:    end if
28:    element = element + 1
30: end while
return finalQuery
```
letter from the string. For example if the string is "friends", element.last will return "".
4.6. Function Rule7

Given a subexpression s that has the form "on t{ φ }" removes those redundancies of the form " on t1{ on t2{ φ } }" having as output an empty expression. t1 and t2 have not got the same type.

Algorithm 6 Rule7(queryList,schema)

```
finalQuery = []
2: first = False // True if it is inside a selection set of type, for example "on t{ ... }", savetype = ""
4: element = 0
    object_types = []
6: file = open(schema,'r')
    data = file.readlines() //returns all lines in the file, as a list where each line is an item in the list object.
8: file.close()
for word in data do
10:   if word == "type" and word.next != "Query" then
        object_types.append(word.next)
12: end if
14: while element <= queryList.length - 1 do
16:   if querList[element] == "on" and first == False then
18:     if querList[element+1].last in object_types then
        first == True
20:     savetype = querList[element+1] //saves the object type
22:     finalQuery.append(querList[element])
24:     finalQuery.append(querList[element+1])
26:     element = element + 2
28: else
30:     continue // continues with the next iteration of the loop
32: end if
```
Algorithm 6 Rule7(queryList,schema) (continued)

   else if querList[element] == "on" and first == True and savetype != querList[element+1] then
      30:     if querList[element+1].last in object_types then
                return []
            32:        else
                  finalQuery.append(querList[element])
            34:             finalQuery.append(querList[element+1])
                  element = element + 2
            36:             continue
         38:     else if querList[element].last == "{" and querList[element] != "on" then// if it is
                  finalQuery = finalQuery + queryList[element:]) //Appends the rest of elements
                  in queryList from the chosen element until the end of the list
            40:       return finalQuery
            42:    else if querList[element].last != "(" and querList[element] != "on" and quer-
                  List[element] != ")" then
                  else if querList[element] == ")" then
            44:         finalQuery.append(querList[element])
         46:     element = element + 1
   end while
38:     return finalQuery

element.next: Returns the element that is in the position

element.last: Element is a string from the string list. This operation will take the last letter from the string. For example if the string is "friends", element.last will return "."
4.7. Function Rule2

Given a query list \( q \), merges those selection sets repeated in the same level merging also those fields that are within this selection sets. Receives the output from function rule3 as input.

**Algorithm 7 Rule2(queryList)**

```
finalQuery = []
auxQuery = []
element = 0
flag = False //Helps to know when a field it is inside a selection, for example "friends{" countBraces = 0
position = 0
while element <= queryList.length - 1 do
  if queryList[element] == "on" then
    finalQuery.append(queryList[element])//appends "on"
    finalQuery.append(queryList[element+1])//appends the object type
    element = element + 2
    continue // continues with the next iteration of the loop
  else if queryList[element] not in finalQuery and flag == False then
    finalQuery.append(queryList[element])
    element = element + 1
    continue
  else if queryList[element].last == "{" and queryList[element].length > 1 and flag == False then
    flag = True
    position = finalQuery.index(queryList[element])
  else if Flag == True then
    if queryList[element].last == "}" then
      if countBraces == 1 then
        flag = False
        position = position + 1
        finalQuery[position:position] = auxQuery[1:]//1
      auxQuery = []
      countBraces = 0
      position = 0
      else
        countBraces = countBraces - 1
        auxQuery.append(queryList[element])
    end if
```

27
**Algorithm 7** Rule2(queryList) (continued)

```python
else
34: auxQuery.append(queryList[element])
   if queryList[element].last == "{" then
36:     countBraces = countBraces + 1
   end if
38: end if
else
40: finalQuery.append(queryList[element])
end if
42: element = element + 1
end while
44: return finalQuery
```

element.last: Element is a string from the string list. This operation will take the last letter from the string. For example if the string is "friends", element.last will return "."

1: Appends the auxQuery elements, from the position 1 until the end of the list, in the position given by the variable position.
4.8. Function Rule1

Given a query list q, merges those fields repeated in the same level. Receives the output from function rule2 as input.

Algorithm 8 Rule1(queryList)

```
(finalQuery = []
2: auxQuery = treeFunction(queryList)
   for element in auxQuery do
4:     if element not in finalQuery or element == "\}" or element == "on" then //1
          finalQuery.append(element)
6:     end if
end for
8: return clear(finalQuery)
```

clear: As finalQuery is the consecution of elements appended from auxQuery. This function will converts the tuples into single elements, removing the level part from the tuple (field, level) having just a list of fields.

1: If the field with its level was not added before or the field is equal to "\}" or "on".
4.9. Function treeFunction

Given a query list q, transforms the query creating a list of tuples with the form: (field, level) being level, the name from the selection set where the field is. As there can be nested selection sets with the same name, the level argument will have the number from the level, being 0 the layer where all the fields inside the root are and adding one in case another selection set is found.

Algorithm 9 treeFunction(queryList)

```
finalQuery = []
2: auxList = []
nLayer = 0
4: for element in queryList do
   if element.last == "{" then // field.last is the last letter from the string, for example
      auxList.append(element+nLayer)
nLayer = nLayer + 1
8: else if element == "}" then
      auxList.remove(auxList.last)
   else
      nLayer = nLayer - 1
   end if
12: finalQuery.append((element,auxList.last))// appends the last element from the
   list, in that case it will be the layer where the field it is.
end for
14: return finalQuery
```

element.last: Element is a string from the string list. This operation will take the last letter from the string. For example if the string is "friends", element.last will return "".

Having the following query:

```
{ getAlien(id: 45) {
   friends{
      name
      age
   }
}
```

Listing 18: Example query

This function will produce the following output:
Listing 19: Example query

```graphql
[
  (getAlien(id: 45){ , []),
    (friends{ , getAlien(id: 45)0)
      (name , friends{1)
      (age , friends{1)
    () , friends{1)
  () , getAlien(id: 45)0)
]```

5. DISCUSSION

5.1. Results

The creation of different tests with different complexities shows how the algorithm works. Thanks to these tests it is possible know that the algorithm works like it has to work. A total of 81 tests were created using different schema files. Last section is a summary of all the tests that were created. I tried to cover all possibilities according to the specification of the algorithm. The tests, test different cases, for example:

- Queries with redundancies
- Queries without redundancies
- Queries that are not in ground-typed form
- Queries that are in ground-typed form
- Nested queries
- Queries that need one rule to be normalized
- Queries that need more than one rule to be normalized

When the testing process started, the algorithm had some problems with some tests that did not match with the expected result. Thanks to having prepared the tests before the start of the implementation the aim of the project was understood better and how it should works. This fact helped to follow the specification that was prepared some weeks before the application of the first tests. All the queries tested had a satisfactory result, which means that all the expected results coincided with the results provided from the algorithm.

The following table shows all the tests realised and for each test which rule had to be applied to normalize the query. As it can be can seen, rule5 and rule8 are applied every time except for the queries from query36 to query46 due to these queries were already in the normal form and none rule was needed. However, I cannot be absolutely sure that the algorithm works correctly for all possible cases.
5.2. Method

The first idea that I had was to apply the tests one by one. When I had more than 10 tests I thought about automatize the testing part implementing a program which takes all the queries, one by one, and should checks the output from the algorithm with the expected result. This implementation also takes the schema file that is specified.

I could not implement rule6 cited in the paper [1] due to the short time.

Probably, the main critique I can make about the project can be how I started to structure it. When I started with the thesis I did not know how to start with it and I spent too much time on something that later I did not need or I did not use. The same happened to me when I was documenting the project, I wrote some things that later I did not use.

How I structured the code also gave me some problems. After a few days of not working in the thesis due to I had some assignments to do for university, it took me a few days to understand what the intent of some parts of the code was. For example, after testing the algorithm and obtaining an incorrect output, sometimes the process to resolve the error was slow. The documentation of the algorithm was not the best, so sometimes the time to understand my code was greater than the time needed to fix the error.
5.3. Sources

The paper "Semantics and Complexity of GraphQL", [1], which was written by Olaf Hartig and Jorge Peréz, was the main source that I used to prepare my project. My project is based on their work. It was difficult to find information related to my project on Internet. The unique information that I found and I could use for the research, was that paper. Thanks to their work, I could use their rewritten rules in order to implement the algorithm and understand the aim of this project. The official documentation of GraphQL, [3], also helped me to understand GraphQL.

As the code it is written in Python, I had also to find some information about this programming language. When I was implementing the algorithm I had to read more about some characteristics of this language so I searched for information in their official documentation, [4].
6. CONCLUSION

GraphQL is a query language invented by Facebook originally in 2012 due to the problems they found when they were building the newsfeed, which was the first thing that we saw when we opened the Facebook App. The first implementation was slow on the network and fragile in terms of changes in the API. It was necessary to be careful because it was easy for the App to crash. With all this problems, they decided to create what we know nowadays as GraphQL. After three years of internal use, Facebook released in 2015 a specification and a reference implementation of its GraphQL framework. Nowadays GraphQL has a big and active community and is available in dozens of different languages that are managed by different community members. Companies like Netflix, Twitter, Airbnb, Amazon or Github quickly included this framework within their projects and the community is also worldwide, having meetups on every continent and conferences around all the world.

In spite of all the advances that this technology has brought, it has some problems derived from the lack of a formal form to create the queries. As the queries can be created in so many different ways and one query can be written in several forms this causes different responses times in queries even though the queries are asking for the same.

"Semantics and Complexity of GraphQL" [1] establishes seven rules with which you can rewrite every query to the normal form. This means that all the queries in normal form or those queries that were already rewritten to the normal form will need the same response time. During the course of this project one rule was added, being eight the total amount of rules instead of seven.

Creating an algorithm capable of rewrite any query into the normal form assures that field collection will not be needed and we will save time in the response time. All the responses will take the same time and the problem of multiple forms to write a query will be solved, which is the aim of this thesis project.
As the time to perform the thesis was not too long, rule number six was impossible to implement. As future work, the implementation of rule number six it is needed as well as the implementation of their tests and their respective solutions. This algorithm could be introduced in the query IDE of GraphQL solving with that the problems that this query language has.

Finally to finish, in relation to test the algorithm I created a serie of test with their own solution. This tests are applied automatically thanks to a program that I programmed in order to make the testing part easier. This queries can be used in the future as well as the testing program to test future projects and everyone can contribute writing more queries due to everything is in my gitlab: https://gitlab.ida.liu.se/juaga388/thesis.
REFERENCIAS


A. TESTS

This appendix presents a summary from all the tests that were used to test the algorithm and their results. The schema used to test the algorithm was copied from the paper "Semantics and Complexity of GraphQL" [1]:

Listing 20: Schema file

type Starship {
    id: ID
    name: String
    length: Float
}

interface Character {
    id: ID
    name: String
    friends: [Character]
    cousins: [Character]
    parents: [Character]
}

type Droid implements Character {
    id: ID
    name: String
    friends: [Character]
    primaryFunction: String
}

type Human implements Character {
    id: ID
    name: String
    friends: [Character]
    cousins: [Character]
    parents: [Character]
    starships: [Starship]
}

enum Episode { NEWHOPE EMPIRE JEDI }
union SearchResult = Human | Droid | Starship


type Query {
    hero(episode: Episode): Character
    search(text: String): [SearchResult]
}

schema {
    query: Query
}

Query1: It is only necessary to apply the rule1 and it asks twice for the name of the hero in the episode empire. It has a field redundancy problem with which rule number one it can be fixed.

Listing 21: Query1

hero[episode:EMPIRE]{
    name
Result after the application of rule1.

Listing 22: Result query1

```graphql
hero[episode:EMPIRE]{
  on Droid{
    name
  }
  on Human{
    name
  }
}
```

Query2: It is only necessary to apply the rule1 due to it has some fields redundancies because it ask four times for the name and three times for the id of the hero in the episode empire.

Listing 23: Query2

```graphql
hero[episode:EMPIRE]{
  name
  name
  name
  name
  id
  id
  id
}
```

Result after the application of rule1.

Listing 24: Result query2

```graphql
hero[episode:EMPIRE]{
  on Droid{
    name
    id
  }
  on Human{
    name
    id
  }
}
```
Query3: This query, tests if the algorithm is capable to remove those repeated fields that are mixed. The query asks four times for the name and three times for the id of the hero in the episode empire. This problem can be fixed applying rule1.

Listing 25: Query3

```
hero[episode: EMPIRE]{
    name
    id
    name
    id
    name
    id
}
```

Result after the application of rule1.

Listing 26: Result query3

```
hero[episode: EMPIRE]{
on Droid{
    name
    id
}
on Human{
    name
    id
}
}
```

Query4: This query asks for the friends name and the friends id of the hero in the episode empire. The selection field friends is repeated twice so here we have a redundancy problem that affects a selection field. Applying rule2 this problem can be fixed.

Listing 27: Query4

```
hero[episode: EMPIRE]{
    friends{
        name
    }
    friends{
        id
    }
}
```

Result after the application of rule2.
Query5: Here we have a nest within the selection field friends. This nesting has two fields called cousins that are repeated twice and field friends that are also repeated twice. This can be fixed after apply rule2.

Result after the application of rule2.

```
Listing 28: Result query4

hero[episode:EMPIRE]{
  on Droid{
    friends{
      id
      name
    }
  }
  on Human{
    friends{
      id
      name
    }
  }
}

Query5: Here we have a nest within the selection field friends. This nesting has two fields called cousins that are repeated twice and field friends that are also repeated twice. This can be fixed after apply rule2.

Listing 29: Query5

hero[episode:EMPIRE]{
  on Human{
    name
    friends{
      cousins{
        id
      }
      cousins{
        name
      }
    }
    friends{
      id
    }
    id
  }
}

Result after the application of rule2.

Listing 30: Result query5

hero[episode:EMPIRE]{
  on Human{
    name
    friends{
      id
      cousins{
        name
      }
    }
  }
}
```
Query6: Here we have a two nest within the selection field friends. This nesting has two fields called parents that are repeated twice, nested inside the selection field cousins that are also repeated twice and field friends that are repeated twice too. This can be fixed after apply rule2.

Listing 31: Query6

```graphql
hero[episode: EMPIRE]{
  on Human{
    name
    friends{
      cousins{
        id
        parents{
          name
        }
        parents{
          id
        }
      }
      cousins{
        name
      }
    }
    friends{
      id
    }
  }
}
```

Result after the application of rule2.

Listing 32: Result query6

```graphql
hero[episode: EMPIRE]{
  on Human{
    name
    friends{
      id
      cousins{
        name
        id
        parents{
```
Query7: This query has a problem related with the fields friends and cousins that are repeated twice where the field cousins is nested inside the field friends. Each field cousins is inside of each field friends. This can be fixed applying rule2.

Listing 33: Query7

```graphql
hero[episode:EMPIRE] {
  on Human {
    friends {
      cousins {
        id
      }
    }
    friends {
      cousins {
        name
      }
    }
  }
}
```

Result after the application of rule2.

Listing 34: Result query7

```graphql
hero[episode:EMPIRE] {
  on Human {
    friends {
      cousins {
        id
        name
      }
    }
  }
}
```

Query8: Here we have a redundancy where the fields friends, cousins and parents are repeated twice. The complexity to solve this query is bigger due to this fields are nested within each other. This can be fixed with rule2.
Listing 35: Query8

```graphql
hero[episode:EMPIRE]{
  on Human{
    friends{
      cousins{
        id
        parents{
          name
        }
      }
    } friends{
      cousins{
        name
        parents{
          name
        }
      }
    }
  }
}
```

Result after the application of rule2.

Listing 36: Result query8

```graphql
hero[episode:EMPIRE]{
  on Human{
    friends{
      cousins{
        id
        parents{
          id
          name
        }
      }
    }
  }
}
```

Query9: This query asks two times for the friends names of the hero in the episode empire. This can be fixed first applying rule 2 to merge the two fields friends and then applying rule1 to remove the redundancy in the field name.

Listing 37: Query9

```graphql
hero[episode:EMPIRE]{
  friends{
    name
  }
}
```
Result after the application of rule2 and rule1.

Listing 38: Result query9

```
hero[episode:EMPIRE]{
  on Droid{
    friends{
      name
    }
  }
  on Human{
    friends{
      name
    }
  }
}
```

Query10: This query asks twice for the name of the hero in the episode empire and the field friends is repeated twice asking for the friends name and the friends id of the hero in the episode empire. This can be fixed applying rule2 and rule 1.

Listing 39: Query10

```
hero[episode:EMPIRE]{
  name
  friends{
    name
  }
  friends{
    id
  }
  name
}
```

Result after the application of rule2 and rule1.

Listing 40: Result query10

```
hero[episode:EMPIRE]{
  on Droid{
    name
    friends{
      id
    }
    name
  }
}
```
Query11: This query asks twice for the name of the hero in the episode empire and the field friends is repeated twice asking for the friends name and the friends id of the hero in the episode empire. The difference with the query10 is that query11 also asks for the id of the hero in the episode empire in order to check that the algorithm takes all the elements as it has to do it. This can be fixed applying rule2 and rule 1.

Listing 41: Query11

```plaintext
hero[episode:EMPIRE]{
  name
  friends{
    name
  }
  friends{
    id
  }
  name
  id
}
```

Result after the application of rule2 and rule1.

Listing 42: Result query11

```plaintext
hero[episode:EMPIRE]{
  on Droid{
    name
    friends{
      id
      name
    }
    id
  }
  on Human{
    name
    friends{
      id
      name
    }
    id
  }
  name
  id
}
```
Query12: This query has the filed name repeated twice and the cousins id of the friends is also repeated twice. As well the field friends is repeated twice having another redundancy. This can be fixed applying first rule2 and then rule1.

Listing 43: Query12

```graphql
hero[episode:EMPIRE]{
  on Human{
    name
    friends{
      cousins{
        id
      }
      cousins{
        id
      }
    }
    friends{
      id
    }
    name
    id
  }
}
```

Result after the application of rule2 and rule1.

Listing 44: Result query12

```graphql
hero[episode:EMPIRE]{
  on Human{
    name
    friends{
      id
      cousins{
        id
      }
    }
    id
  }
}
```

Query13: This query asks twice for the name of the Human hero in the episode empire, twice for the cousins id and there are several selection fields repeated as parents, cousins and friends. Applying rules one and two these redundancies can be removed.
Listing 45: Query13

hero[episode:EMPIRE] {
    on Human {
        name
        friends {
            cousins {
                id
                parents {
                    name
                }
                parents {
                    id
                }
            }
            cousins {
                id
            }
        }
    }
    friends {
        id
        name
        id
    }
}

Result after the application of rule2 and rule1.

Listing 46: Result query13

hero[episode:EMPIRE] {
    on Human {
        name
        friends {
            id
            cousins {
                id
                parents {
                    id
                    name
                }
            }
        }
    }
    id
}

Query14: This query asks twice for the name of the Human hero in the episode empire, twice for the cousins id, twice for the parents name and there are several selection fields repeated as parents, cousins and friends. Applying rules one and two these redundancies can be removed
Listing 47: Query14

```javascript
hero[episode:EMPIRE] {
  on Human {
    name
    friends {
      cousins {
        id
        parents {
          name
        }
        parents {
          id
          name
        }
      }
      cousins {
        id
      }
    }
    friends {
      id
      name
      id
    }
  }
}
```

Result after the application of rule2 and rule1.

Listing 48: Result query14

```javascript
hero[episode:EMPIRE] {
  on Human {
    name
    friends {
      id
      cousins {
        id
        parents {
          id
          name
        }
      }
      id
    }
  }
}
```

Query15: Has two selection fields of type on Human that are nested and asks for the Human hero name in the episode empire. The problem can be solved applying rule3 and rule4.
Query 15: Has three nested selection fields of type on Human and asks for the Human hero name in the episode empire. The problem can be solved applying rule 3 and rule 4.

Query 16: Has five nested selection fields of type on Human and asks for the Human hero name in the episode empire. The problem can be solved applying rule 3 and rule 4.
Listing 53: Query17

```graphql
hero[episode:EMPIRE]{
  on Human{
    on Human{
      on Human{
        on Human{
          on Human{
            name
          }
        }
      }
    }
  }
}
```

Result after the application of rule3 and rule4.

Listing 54: Result query17

```graphql
hero[episode:EMPIRE]{
  on Human{
    name
  }
}
```

Query18: This query is asking for the name of a Human of type Droid which has a Human type. This syntax is correct for GraphQL but it does not make sense. With the application of rule3, rule4 and rule7 this problem can be solve.

Listing 55: Query18

```graphql
hero[episode:EMPIRE]{
  on Human{
    on Droid{
      on Human{
        name
      }
    }
    name
  }
}
```

Result after the application of rule3, rule4 and rule7.

Listing 56: Result query18

```graphql
hero[episode:EMPIRE]{
  on Human{
    name
  }
}
```
Query19: This query nests a Droid type within a selection field of type Human. This problem is solved applying rule3, rule4 and rule7.

Listing 57: Query19

```java
hero[episode:EMPIRE]{
  on Human{
    on Droid{
      on Human{
        on Human{
          on Human{
            on Human{
              name
            }
          }
        }
      }
    }
  }
}
```

Result after the application of rule3, rule4 and rule7.

Listing 58: Result query19

```java
hero[episode:EMPIRE]{
  on Human{
    name
  }
}
```

Query20: This query nests an on Droid type after four on Human nested selection fields. This problem is solved applying rule3, rule4 and rule7.

Listing 59: Query20

```java
hero[episode:EMPIRE]{
  on Human{
    on Human{
      on Human{
        on Human{
          on Droid{
            id
            on Droid{
              name
            }
          }
        }
      }
    }
  }
}
```
Result after the application of rule3, rule4 and rule7.

Listing 60: Result query20

```graphql
hero[episode:EMPIRE]{
  on Human{
    id
    name
  }
}
```

Query21: This query is asking for the Human’s id, the Droid’s name and the Human’s name. It has three Human type nested and in the third layer has another Human type nested and a Droid type. The algorithm will remove the redundancies related to the Human type and it will remove the Droid type.

Listing 61: Query21

```graphql
hero[episode:EMPIRE]{
  on Human{
    on Human{
      on Human{
        on Human{
          id
        }
        on Droid{
          name
        }
        name
      }
    }
  }
}
```

Result after the application of rule3, rule4 and rule7.

Listing 62: Result query21

```graphql
hero[episode:EMPIRE]{
  on Human{
    id
    name
  }
}
```

Query22: Asks for the id of the hero in the episode empire and the Human’s name in this episode. On Human is nested twice and the field name is repeated twice. The algorithm will remove this redundancies and will normalize the query.
Listing 63: Query22

```graphql
hero[episode: EMPIRE] {
  id
  on Human {
    name
    on Human {
      name
    }
  }
}
```

Result after the application of rule3, rule4 and rule1.

Listing 64: Result query22

```graphql
hero[episode: EMPIRE] {
  on Droid {
    id
  }
  on Human {
    id
    name
  }
}
```

Query23: Asks for the name of the Character Human in the episode empire twice and for the id. The field on Human is nested. The algorithm will solve this redundancies removing the nest and the redundancy in the field name.

Listing 65: Query23

```graphql
hero[episode: EMPIRE] {
  on Human {
    name
    on Human {
      id
      name
    }
  }
}
```

Result after the application of rule3, rule4 and rule1.

Listing 66: Result query23

```graphql
hero[episode: EMPIRE] {
  on Human {
    id
    name
  }
}
```
Query24: Asks for the name of the Human’s hero in the episode empire twice. This selection field on Human is nested six times. Those redundancies are not needed, after the application of the algorithm these redundancies will be removed.

Listing 67: Query24

```graphql
hero[episode:EMPIRE] {
  on Human {
    on Human {
      on Human {
        on Human {
          on Human {
            on Human {
              name
            }
          }
        }
      }
    }
  }
}
```

Result after the application of rule3, rule4 and rule1.

Listing 68: Result query24

```graphql
hero[episode:EMPIRE] {
  on Human {
    name
  }
}
```

Query25: Asks for the name of the Human’s hero in the episode empire four times in different parts of the query. The selection field on Human is nested six times. As this kind of repetition are not needed in GraphQL, the algorithm will remove them having as a result the "result query25".

Listing 69: Query25

```graphql
hero[episode:EMPIRE] {
  on Human {
    on Human {
      name
    }
    on Human {
      name
    }
    on Human {
      name
    }
    on Human {
      name
    }
    on Human {
      name
    }
  }
}
```
Result after the application of rule3, rule4 and rule1.

Listing 70: Result query25

hero[episode: EMPIRE] {
  on Human {
    name
  }
}

Query26: Asks for the name of the Human’s hero in the episode empire six times in different parts of the query. The selection field on Human is nested six times. As this kind of repetition are not needed in GraphQL, the algorithm will remove them having as a result the "result query26".

Listing 71: Query26

hero[episode: EMPIRE] {
  name
  on Human {
    on Human {
      name
    }
  }
}

Result after the application of rule3, rule4 and rule1.
Listing 72: Result query26

```javascript
hero[episode:EMPIRE]{
  on Droid{
    name
  }
  on Human{
    name
  }
}
```

Query27: This query has a selection field friends repeated twice inside another selection field on Human that is nested inside another selection field of the same type on Human. The algorithm will remove those redundancies having as result "result query27".

Listing 73: Query27

```javascript
hero[episode:EMPIRE]{
  id
  on Human{
    name
    on Human{
      friends{
        id
      }
      friends{
        name
      }
    }
  }
}
```

Result after the application of rule3, rule4 and rule2.

Listing 74: Result query27

```javascript
hero[episode:EMPIRE]{
  on Droid{
    id
  }
  on Human{
    id
    name
    friends{
      name
      id
    }
  }
}
```
Query 28: This query asks for the name of all the heroes in the episode empire, also it asks for Droid’s name and for the Human’s id. As we can see this query has a redundancy related to the field name.

Listing 75: Query 28

```graphql
hero[episode:EMPIRE]{
  on Droid{
    name
  }
  on Human{
    id
  }
  name
}
```

Result after the application of rule 3, rule 4, rule 7 and rule 1.

Listing 76: Result query 28

```graphql
hero[episode:EMPIRE]{
  on Droid{
    name
  }
  on Human{
    id
    name
  }
}
```

After check that all these tests generate the correct result, I changed the schema file adding a new type called animal that implements Character:

Listing 77: Schema file

```graphql
type Starship {
  id: ID
  name: String
  length: Float
}
interface Character {
  id: ID
  name: String
  friends: [Character]
  cousins: [Character]
  parents: [Character]
}
type Droid implements Character {
  id: ID
  name: String
  friends: [Character]
```
primaryFunction: String

type Human implements Character {
  id: ID
  name: String
  friends: [Character]
  cousins: [Character]
  parents: [Character]
  starships: [Starship]
}
type Animal implements Character {
  id: ID
  name: String
  friends: [Character]
  cousins: [Character]
  parents: [Character]
  starships: [Starship]
}

enum Episode { NEWHOPE EMPIRE JEDI }
union SearchResult = Human | Droid | Animal | Starship

type Query {
  hero(episode: Episode): Character
  search(text: String): [SearchResult]
}
schema {
  query: Query
}

Query29: Asks twice for the name of the hero in the episode empire. This redundancy can be solved applying the algorithm.

Listing 78: Query29

hero[episode:EMPIRE]{
  name
  name
}

Result after the application of rule1.

Listing 79: Result query29

hero[episode:EMPIRE]{
  on Droid{
    name
  }
  on Human{
    name
  }
  on Animal{
    name
  }
}
Query30: Asks twice for the Human’s name of the hero in the episode empire in different parts of the query. The selection field on Human is nested twice. These redundancies are not needed and they can be removed applying the algorithm.

Listing 80: Query30

```graphql
hero[episode:EMPIRE]{
  id
  on Human{
    name
    on Human{
      name
    }
  }
}
```

Result after the application of rule3, rule4 and rule1.

Listing 81: Result query30

```graphql
hero[episode:EMPIRE]{
  on Droid{
    id
  }
  on Human{
    id
    name
  }
  on Animal{
    id
  }
}
```
<table>
<thead>
<tr>
<th>Firmante</th>
<th>CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fecha/Hora</td>
<td>Tue Jan 14 19:09:18 CET 2020</td>
</tr>
<tr>
<td>Emisor del</td>
<td>EMAILADDRESS=<a href="mailto:camanager@fi.upm.es">camanager@fi.upm.es</a>, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES</td>
</tr>
<tr>
<td>Certificado</td>
<td></td>
</tr>
<tr>
<td>Numero de Serie</td>
<td>630</td>
</tr>
<tr>
<td>Metodo</td>
<td>urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.shal (Adobe Signature)</td>
</tr>
</tbody>
</table>