Query Decomposer and Optimizer for Querying Scientific Datasets

ZHIYAN LU

Matriculation number: 2779846

March 2019

A thesis submitted in partial fulfillment for the degree of Master of Science

Institute of Computer Science

Supervisors:
Dr. Ioanna Lytra, Dr. Damien Graux

Examiners:
Prof. Dr. Jens Lehman

INSTITUTE FÜR INFORMATIK III
RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

Declaration of Authorship

I, ZHIYAN LU, declare that this thesis titled, 'Query Decomposer and Optimizer for Querying Scientific Datasets' and the work in this thesis presented here are my own. I confirm that:

- The work in this thesis was done wholly or mainly while in candidature for a research degree at the University of Bonn.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at the University of Bonn or any other institution, this has been clearly stated.
- Where I have consulted the published work of others in the thesis, it is always clearly attributed.
- Where I have quoted from the work of others, the sources are always given. I have already acknowledged all the main sources of help in the thesis. With the exception of these quotations, the thesis is my own work entirely.
- I have made clear exactly what was done by others and what I have contributed myself where the thesis is based on work done by myself jointly with others.

Signed: _____

Date: _____

RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

Abstract

Institute of Computer Science

Master of Science

by ZHIYAN LU

In recent years, the amount of scientific datasets has increased dramatically. There are more and more scientific datasets which are publicly available on the Web and can be studied and researched by users. However, being able to query or analyze scientific data without knowing about the underlying datasets is not possible at the moment, which should be not very convenient for the users. The goal of the thesis is to create a query engine that can be able to query scientific datasets transparently, without being aware of the available datasets. In our thesis, first, we build a query engine which is based on the decomposer of ANAPSID query engine, then we add the metadata file into it. In the experiment, we found some defects in the query engine of dealing with the complex queries, so we redesigned a new query engine using Graphql as the query language. As a result, we incorporated two new query engines so that users can use query data without knowing about the underlying datasets.

Acknowledgements

I would first have to thank my thesis advisors. Thank you so much for providing such an interesting topic for me. I could never start this thesis without them.

I would be grateful to my supervisors, Dr. Ioanna Lytra and Dr. Damien Graux. The door was always open whenever I meet trouble or had a question about my thesis. They allowed this paper to be my own work but steered me in the right direction whenever they thought I needed it consistently.

Finally, I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of my thesis. This accomplishment would not have been possible without them. Thank you all.

Contents

Chapter 1

Introduction

1.1 Motivation and Problem Statement

With the rapid development of information technology and the rapid expansion of network information technology, the network is a□ecting and changing our daily lives increasingly. In the current situation, the network is playing an increasingly important role. How to get the most quantity the most valuable information on the Internet at the fastest speed has gradually become the most concerned issue for all users. The huge amount of data on the Internet leads users to feel the great shock and surprise brought by the information explosion. Meanwhile, it is extremely di□cult for the users to find the data resources which are needed in the vast amount of network data in a very short time.

At the same time, the amount of scientific datasets also has increased dramatically in recent years. As a result, there are more and more scientific datasets which are publicly available on the Web and can be studied and researched by all users. Meanwhile, there are several data formats of the scientific datasets, and NetCDF[1] is one of the most popular and most widely used data formats of scientific datasets. For example, Copernicus data repository - http://www.copernicus.eu/[2] is a prominent collection of datasets related to climate, atmosphere, agriculture, and marine domains. The data format of the scientific datasets in Copernicus data repository is NetCDF. NetCDF is short for the Network Common Data Format. NetCDF is a set of software libraries and self-describing, machine-independent data formats, which is used to support the creation, accessing and sharing of array-oriented scientific data. The NetCDF file is created to store data in meteorological science, it is the format for generating data for various data

[1] https://www.unidata.ucar.edu/software/netcdf/docs/
[2] http://www.copernicus.eu/

acquisition software now. So we choose NetCDF as the data format of scientific data for researching in our thesis.

For the users, they always need to query or analyze the scientific data frequently. At this moment, However, it is not possible for users to query or analyze scientific data without knowing about the underlying datasets. Users should know the specific position of the data or the whole structure of the datasets before they are querying them, then users can visit the dataset to get the information about data which they want. Meanwhile, the dataset is always very huge and heterogeneous, which means it always consists of various parts or types, that must be confused for the users. We need to find a solution to help them.

So, we need the data query engine for those users to query the NetCDF datasets. Simply put, the data query engine is one of the core components of the big data processing architecture [1], usually there is a direct interface to the data application layer. In general, a typical data query engine [2] [3], will have the following three main characteristics:

1. Query engine provides a way which is to be controlled by the user's input query that is to be a relational database, thus enabling read and write data access.
2. Query engine provides a query interface to identify and interpret the input query language.
3. Query engine gets returned structured query results and implements an integrity strategy including the relational data models. It also provides various other resources and data which are required by the database management system.

As we know, there are a lot of query languages in the world, SQL and SPARQL are two of them which are popular now. SQL[3] is short for Structured Query Language and SPARQL[4] is short for SPARQL There will be a more detailed introduction about SQL and SPARQL in chapter 2.

The main problem is, being able to query or analyze scientific data without knowing about the underlying datasets is not possible at the moment, which should be not very convenient for the users. Let's take an easy but common example. There are two datasets which are supported by the Copernicus data repository. Both of the datasets have some variables about concrete time. The temperature of one concrete time is in the dataset "$MO\_201708\_PR\_PF\_6903279$", and the pressure of one concrete time is in another dataset "$MO\_201708\_PR\_PF\_6903280$". If a user wants to query the

[3] https://www.w3schools.com/sql/
[4] https://www.w3.org/TR/rdf-sparql-query/

temperature and the pressure of the time "24685.217951388888", he must have to know where the two variables, the $temperature$ and the $pressure$, are stored in which dataset first. That should be huge trouble for the users. The goal of my thesis is trying to solve this kind of problem. We create new query engines which will be able to query scientific datasets transparently, without being aware of the available datasets, and finally solve the problems and get the results for the users.

There already exist numerous kinds of query engines right now. Among them, we find a query engine called ANAPSID query engine[5] to query those scientific datasets in NetCDF data format. The most part which interests me is the decomposer part of the ANAPSID query engine, which can be improved to solve the existing problems. We need to understand how the decomposer part of the ANAPSID query engine work, to build a query engine based on the decomposer part of APAPSID query engine named as MGMDQE query engine, to improve the performance to solve the main problem.

During testing the MGMDQE query engine in the experiment, we find that there exist some defects when dealing with some complex queries. When the variables in the complex query have some inner-relationships, the MGMDQE query engine will get an error. Because of the defects in the MGMDQE query engine, we try to build another query engine named as GraphQLQE query engine which contains new solutions for query decomposers and new schemas of creating the query plan. It uses GraphQL as the input query language because of its flexibility and scalability, and we can write different schemas following some rules by ourselves to meet most kinds of the query types including the complex query which the MGMDQE query engine can't deal with.

1.2 My Contributions

In this thesis, the contributions of my work can be divided into two main parts.

The first part of my contributions focuses on the decomposer part of ANAPSID query engine. We build a new query engine called MGMDQE query engine based on the decomposer part of ANAPSID query engine. We try to improve performance by updating its structure and to add the metadata into it. After we update its structure, MGMDQE query engine can be divided into three main steps. The first step is called the Make plan step where the engine uses a tool called PLY to parse the input query. The second step is called the Create plan step where the engine groups the input query as a data structure and build a tree structure of the input query according to this data structure. The third step is called the Interact Proxy step where the query engine sends several

[5] https://github.com/anapsid/anapsid

requests by sockets to MongoDB which stores several NetCDF datasets and listens to the response back. Afterwards, we try to add the metadata into the MGMDQE query engine. The metadata is a data type which describes the datasets and it is the key to solve the main problem. The query engine can analyze the components in the metadata to get the information about the variables. By adding the metadata here successfully, the query engine can be able to query scientific datasets transparently without being aware of the available datasets.

The second part of my contributions tries to build another new query engine called GraphQLQE query engine which contains new solutions for query decomposer and new schemas of creating the query plan. It can be used to meet most kinds of the query types including those query types which MGMDQE query engine can't deal with. The input query language of the new query engine is the GraphQL. GraphQL is a new query language, one of its features is that it is not linked to any specific database or storage engine and is instead backed by the existing code and data, so the flexibility and the scalability become the biggest advantage of it and it is the main reason why the new query engine uses it. The main part of the GraphQLQE query engine is a lot of schemas written by myself as the query planner for input GraphQL. We make several plans for different queries to query the data in MongoDB. The grammars and the structures in the schemas of creating the query planning part are the cores of the GraphQLQE query engine. And the schemas can be expanded in the future.

1.3 How the Thesis is Organized

This thesis has seven chapters and three appendixes. The rest of the thesis is organized as follows:

- Chapter 2 Presents some literature and knowledge on the background information covering the topics of this thesis. Such as the cognition of scientific data, the basic structure of the NetCDF file, the concept of RDF and SPARQL the PLY(Python Lex-Yacc) technology for parsing the query, and the basic knowledge of GraphQL.
- Chapter 3 Introduces some recent related works about the query engine such as the database comparison for NetCDF Data storage, the Big Data Ocean Harmonisation, the Federated query engines and the basic idea of the ANAPSID Query Engine.
- Chapter 4 Introduces the basic architecture of MGMDQE query engine based on the decomposer part of ANAPSID query engine. Introduces how the MGMDQE

query engine works step by step, and add metadata into MGMDQE query engine to solve the main problem.

- Chapter 5 Introduces how the GraphQLQE query engine works for solving the problems which MGMDQE query engine meet. Introduces how are the new schemas and the structures of GraphQLQE query engine.
- Chapter 6 Provides a series of experiments to evaluate the quality and validation of the MGMDQE query engine and the GraphQLQE query engine by executing the query accurate and the spend time.
- Chapter 7 Concludes the whole thesis and discusses the works which need to be extended in the future.

Chapter 2

Background

In this chapter, I'll present some relative literature and basic knowledge on the background information covering the topics of this thesis. Such as the basic understanding of the scientific data, the structure of the NetCDF data format, the basic definition and some extensions of RDF and SPARQL, the tool of PLY(Python Lex-Yacc) technology for parsing the query, and the basic knowledge of GraphQL that will be used later.

2.1 Scientific Data

Scientific data is defined as the information collected which is using specific methods. These methods are used for studying or analyzing a specific purpose. In recent years, scientific data management is one of the interesting topics of scientific data [4].

The scientific data has two main features:

1. The scientific data is called the hypothesis-driven science, it means, the scientific data is collected that can support some scientific hypothesis which is proposed. In this process of support, it is frequently reported as one or more experiments.
2. The scientific data is also called the data-driven science, it means, where the scientific data is collected and it is analyzed to show not only the patterns within the dataset but also the time when it is combined with other data.

The scientific data can be created from two main sources:

1. Observation and Measurement. In a lot of scientific domains, the observation is still the most important method to create scientific data. The measurements are

carried out by scientists or researchers and recorded in notebooks or other other forms of note. Meanwhile, the measurement of scientific data is recorded by sensors and instruments increasingly.

2. Calculation. In other scientific domains, the scientific data can also be created by direct calculation of observables in quantity. Computer programs are often capable of showing the results with experiment and they are much cheaper and easier to simulate some unobservable situations, that may be very important and valuable for some scientific data.

   Scientific data is the main work object of my master thesis research, and we choose NetCDF as the data format of scientific data because it is the most popular and widely used data format of scientific data. There will be a more detailed introduction to the NetCDF data format later.

2.2 Basic structure of NetCDF

We will choose NetCDF as the format of scientific datasets for our research object because NetCDF is one of the most popular scientific database formats[5]. The NetCDF (Network Common Data Sheet) network common data format was developed by Unidata project scientists at the University of the Atmospheric Research Corporation (UCAR)[1] to study the characteristics of scientific data. It is array-oriented and suitable for network sharing. Data description and coding standards. Users can e□ectively manage and operate NetCDF datasets in a variety of ways. With NetCDF, we can e□ciently store, manage, acquire and distribute grid data. Because of its flexibility, it can transmit a large number of array-oriented data and is commonly used in many fields such as atmospheric science, hydrology, oceanography, environmental simulation, geophysics. Actually, we can find that NetCDF is an interface for scientific data access[6].

Mathematically, the data stored in NetCDF are a single-valued function of multiple independent variables. In terms of a formula, $f(x, y, z, ...) =$ value, the argument x, y, z, etc. of the function are called dimensions or axes in NetCDF, and the function value is called variable in NetCDF. A NetCDF dataset contains three description types: dimensions, variables, and attributes. Each type is assigned a name and an ID. These types together describe a data set. The NetCDF library can be accessed simultaneously. Multiple data set, using IDs to identify di□erent data sets. The variable stores the actual data, the dimension gives the variable dimension information, the attribute gives the auxiliary information attribute of the variable or the data set itself, and can be divided

[1] https://www.ucar.edu/

into the global attribute applicable to the entire file and the local attribute applicable to the specific variable, the global attribute. It describes the basic properties of the dataset and the source of the dataset.

The structure of a NetCDF file includes the following objects:

```
NetCDF name{
Dimensions:... //Define the dimension
Variables:... //Define variables
Attributes:... //properties
Data: ...//data
}
```

The excerpt of NetCDF file is shown in Appendix A.

The NetCDF software implementation is a free NetCDF package that contains a utility program that accesses NetCDF data and a library of interface functions in multiple languages. In our thesis, we have several example NetCDF files which are downloaded from the Copernicus data repository, and we will use these files to test our query engine.

2.3 RDF and SPARQL

RDF[2] (short for The Resource Description Framework), first published by W3C in 1999, is a semantic web standard vocabulary to describe resources regarding the relations to other resources and can represent the most structured or semi-structured data formats. By the explanation in W3C, we can find that, RDF is a standard model for data interchange on the Web. It has several essential features, one of the features is the facilitate data merging. Which means, even if the underlying schemas of are di□erent, it still support the evolution of di□erent schemas over time without requiring all the data specifically. In a sense, RDF extends the linking structure of the Web, it uses URIs which are used to name the relationship between two ends of the link [7].

Compared to traditional relational database management, RDF-based systems provide a great infrastructure to globally address row identifiers and property names by using existing web technologies for things, having the advantage that enables collaborative systems to integrate and share data across di□erent applications and organizations.

SPARQL[3] (short for SPARQL Protocol and RDF Query Language) is an RDF query language that is a semantic query language for databases [8]. It is used to retrieve and

[2] https://www.w3.org/RDF/
[3] https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

manipulate data stored in RDF format. SPARQL was made a standard by the RDF Data Access Working Group (DAWG)[4]. It is recognized as one of the key technologies of the semantic web and it is faced with large RDF datasets containing millions of triples. SPARQL allows users to write queries and do a full set of analytic query operations to exact useful information from the datasets, such as UNION, JOIN, FILTER and AGGREGATE. However, because of the large size of RDF datasets, supporting e□cient query language is a big challenge, while for the commercial and scientific requirements, services need to answer the query in short waiting time, even with a lot of complex operations and large basic triple patterns. This situation gives us the motivation to develop a high-performance query engine [9] [10].

SPARQL allows users to write some queries against "key-value" data which follows the RDF specification of the W3C. Meanwhile, it is a set of "subject-predicate-object" triples. Thus, we can know that SPARQL provides a full set of analytic query operations, the schemas of these operations are a part of the data which rather than requiring a separate schema definition.

2.4 PLY(Python Lex-Yacc)

PLY[5] is short for Python Lex-Yacc, which is an implementation of lex.py file and yacc.py file parsing tools for Python to parse the input query [11]. PLY consists of two files: lex.py file and yacc.py file. The lex.py file is used to deal with the lexical analysis part and the yacc.py file is used for creating a parser. In a nutshell, PLY is just a straight forward lex and yacc implementation [12]. Here is a list of its essential features which will help to understand what is PLY:

1. PLY is implemented entirely in Python and it is very closely modelled after traditional lex and yacc.

2. Parsing is based on a parsing method called LR-parsing[6]. LR-parsing is faster and has much memory e□cient so that it is better suited to large grammar. PLY has a lot of nice properties when dealing with parsing problems such as syntax errors. Currently, PLY builds its parsing tables using the LALR algorithm used in yacc [13].

3. PLY uses Python introspection features to build the lexers part and the parsers part. In this way, PLY reduces the number of files and also reduces the need for files to

[4] https://www.w3.org/2001/sw/DataAccess/homepage-20080115
[5] https://github.com/dabeaz/ply
[6] https://en.wikipedia.org/wiki/LR_parser

implement a separate lex and yacc tool before running the program. It simplifies the tasks and the costs of parser construction.

4. PLY can be used to build query parsers for some "real" programming languages like C and C++, which means, PLY can be used to parse those grammars which are consisted of hundreds of rules, that is the most feature of PLY tool.

As we mentioned before, PLY can be divided into two different sub-tools, Lex tool and Yacc tool. Lex is short for Lexical Analyzer Generator [14], which helps write several programs whose control flows are directed by instances of regular expressions in the input stream. Lex is also very suited for some transformations such as the editor-script type transformation, and it also can be used to get prepared for the segmenting input during parsing.

Yacc is short for Yet Another Compiler [15], which provides a tool for describing the input to a computer program, specifies the structures of the input and distinguish the code which will be invoked as each recognized structure. In a word, Yacc is a tool that turns the specification into a subroutine and handles the input process at the same time.

In the first step of the MGMDQE query engine, it uses PLY tool to parse the input query. After lex and yacc parsing, there comes a tree in SQL, which are common expression calculations. For example, it selects $a + sum(b + c)$ from T group by a, this statement is that lex and yacc parsing $a + Sum(b + c)$ will be parsed into a tree. The root is $+$, the left child is a, the right child is a sum, the child of the sum is the second $+$. The second child of the left is b, and the second child of the right is c. This is a binary tree structure with leaf nodes (such as table attributes), one-node nodes (such as sum), and binary nodes (such as $+$).

2.5 GraphQL

GraphQL[7] is an open source data query and manipulation language. The runtime of GraphQL is used for fulfilling the input queries with some existing data. GraphQL was developed internally by Facebook[8] in 2012 and was released in 2015 publicly. The advantages of GraphQL is that It provides a more efficient, powerful and flexible alternative to some web service architectures. Which means, GraphQL allows the clients to define the structure of the data required all by themselves, and the structure is just like the same structure of the data which is returned from the server exactly. Therefore, it can

[7] https://graphql.org/learn/
[8] https://github.com/facebook/graphql

prevent the excessively large amounts of data from being returned. For the operations in GraphQL, there are reading, writing and subscribing to changes to data (real-time updates).

GraphQL is a query language for the API [16], and it is a server-side runtime for executing queries by using a type system which is defined for the data. GraphQL is not linked to any specific database or storage engine, however, it is linked to the existing code and data instead of the database or engine. This provides the most advantage of GraphQL, the flexibility, which is also the main reason why I choose it in my second query engine.

A GraphQL service is created by defining different types and fields. It is used to provide different functions for each field and each type. A GraphQL operation can be a query (read operation), a modification (write operation), and a subscription (continuous read operation). Each of these operations is just a string of strings constructed according to the GraphQL standard. Once a GraphQL operation reaches the backend application from the front-end application, it first interprets the entire GraphQL schema on the back end and then parses the relevant data for the front end. GraphQL does not require network layer selection (usually HTTP) and does not require a data format (usually JSON). There is no requirement for an application architecture (usually a front-end separation architecture). As we can see, the details of how the schema stitching works are in the figure 2.1.

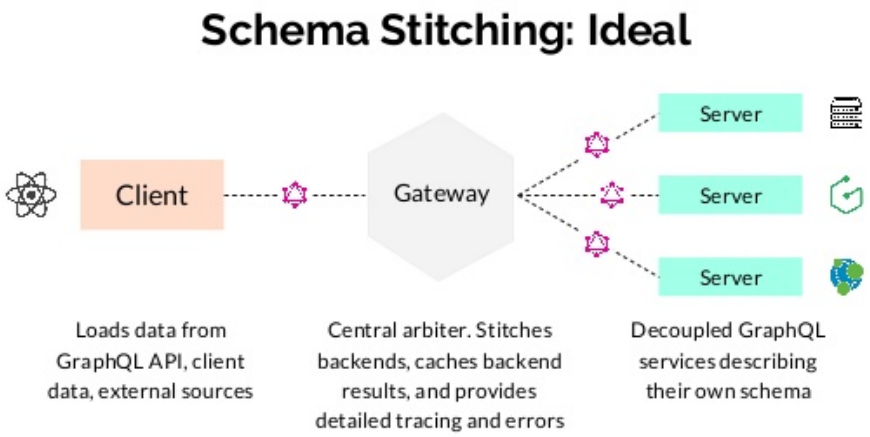

Figure 2.1: Graphql concept

I use GraphQL as the query language for the second query engine because of its flexibility. GraphQLQE query engine contains new solutions for query decomposers and new

schemas of creating the query plan. It uses GraphQL as the input query language, parses the query, and we can write different schemas following some rules by myself which are used to meet most of the query types. We can find more details about the GraphQLQE query engine in Chapter 5.

Chapter 3

Related Work

So far, we have given some brief introductions and background about the problems we meet and the motivation of why we are doing this work in the last two chapters. However, getting a good understanding of how NetCDF files are stored or how the ANAPSID query engine works can help us quickly to know the problems of the query engine. In this chapter, we will start by looking at the related work about how the NetCDF files are stored in the database, get known for Big Data Ocean Harmonisation, then coming to introduce the basic architecture of the ANAPSID query engine.

3.1 Database Comparison for NetCDF Data storage

Firstly, let's have a discussion about the database comparison of the NetCDF Data storage. So far, we have several different kinds of NoSQL database. We need to get familiar with them and to compare them so that we can find the best way for the NetCDF files storage in order to query the data in the database more efficiently.

3.1.1 NoSQL Database

A NoSQL database provides a mechanism for storage and retrieval of data that is modelled [17]. So that in recent years, NoSQL databases become more and more popular and widely used in big data and some other real-time web applications [18]. There are several data concepts which are used by the NoSQL databases, such as the key-value, the column, the graph, and the document. They are different from those which are used by default in relational databases. The advantage is making some operations in NoSQL become faster than other databases [19].

A scientific database is a computerized and organized collection of related data, which means, it can be accessed by users for some scientific studies such as the scientific inquiry and the long-term stewardship. We have found the category of several scientific datasets provided by Wikipedia [20]. According to these datasets, we can find some information on these scientific datasets and find a number of essential requirements that a database must have, which must be important and useful for the users.

1. The database must be guaranteed up-to-date and complete all the time. which means, the database must be very easy to update at any time, it needs that the data providers should be given some suitable incentives to update the information in the database.
2. The database must be presented in a very attractive format to the user, so that the user may get interested in it.
3. The technologies in the database must be compatible with the users. So that the users can use the database without any troubles.
4. The mechanism of the dialog in the database should be able to deal with user inquiries and to meet user needs. This must be the most important requirement for a database.

Here are several different kinds of NoSQL databases which can be the scientific database that we are familiar with: HBase, Redis, MongoDB and so on. According to the above explanations and conditions of scientific datasets, we need to compare them so that we can find the best solution for the NetCDF files storage. Let's compare them now.

3.1.2 HBase

3.1.2.1 Introduction

HBase[1] is a subproject in Apache Hadoop[2]. The Apache Hadoop is an open source version of BigTable that implements the Java language, so it dependents on the Java SDK. HBase relies on Hadoop's HDFS (Hadoop's Distributed File System) as the most basic storage infrastructure. HBase implements the compression algorithms, memory manipulation, and Bloom filters which are mentioned in the BigTable paper. The tables in HBase can be used as the inputs and the outputs for MapReduce tasks. These tables can be accessed through not only the Java API, but also REST APIs, Avro APIs, and Thrift's APIs [21].

[1] https://hbase.apache.org/
[2] https://en.wikipedia.org/wiki/Apache_HBase

3.1.2.2 Advantages and Disadvantages

Advantages:

1. HBase database has a table of large storage capacity, which can be used to accommodate millions of rows and millions of columns;
2. HBase database can be retrieved through the version, which means can search the required historical version of the data;
3. When HBase database with a high load, the horizontal slicing can be expanded by simply adding machines, and integration with Hadoop which ensures its data reliability (HDFS)[3] and its high-performance data analysis (MapReduce)[4];
4. On the basis of 3, it can effectively prevent the occurrence of a single point of failure.

Disadvantages:

1. HBase database is based on the Java language implementation and Hadoop architecture, which mean its API is more suitable for Java projects;
2. HBase database takes up a lot of memory so that gives poor read performance based on HDFS optimized for batch analysis;
3. The API of HBase database is relatively limited compared to some other NoSQL APIs.

3.1.3 Redis

3.1.3.1 Introduction

Redis[5] is an open source, written in ANSI C language that supports networking, memory-based, persistence-based, log-based, Key-Value databases and multilingual APIs. It is currently hosted by VMware development [22].

[3] https://hadoop.apache.org/docs/current1/hdfs_design.html
[4] https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
[5] https://en.wikipedia.org/wiki/Redis

3.1.3.2 Advantages and Disadvantages

Advantages:

1. Redis database has very rich data structure;
2. Redis database provides the function of the transaction, to ensure the atomicity of a bunch of commands, and make sure that the middle will not be interrupted by any operation;
3. Data of the Redis database is stored in memory, which means that read and write are at very high speed.

Disadvantages:

1. The o□cial cluster program came out after the Redis3.0, but there are still some architectural issues;
2. Persistence features of Redis database have a poor experience. It achieved through the snapshot method and it needs to write the entire database data to disk from time to time, so that the cost is very high;
3. Due to a memory database, the amount of stored data is related to the single machine's own memory size. Although Redis has a key expiration strategy, it still needs to predict in advance and sometimes save the memory. So that, if the memory of Redis grows too fast, it needs to delete the data regularly.

3.1.4 MongoDB

3.1.4.1 Introduction

MongoDB[6] is a high-performance, open-source, schema-free, document-type database with a C ++ development language that stores data in JSON-like documents with flexible schemas. It can be used to replace traditional relational databases or key/value storage in many scenarios. MongoDB was developed by the 10gen team [23].

3.1.4.2 Advantages and Disadvantages

Advantages:

[6] https://en.wikipedia.org/wiki/MongoDB

1. MongoDB has powerful automation shading function;
2. MongoDB has full index support, so that the query of MongoDB is very e□cient;
3. MongoDB uses Document-oriented (BSON)[7] storage, the data model is simple and powerful;
4. MongoDB supports the dynamic queries, query instructions and also the JSON[8] form of the mark, which means it can easily query the document embedded objects and arrays;
5. MongoDB Supports JavaScript expression query, execute arbitrary JavaScript function on the server side.

Disadvantages:

1. The size of a single document in MongoDB limit is 16M and 32-bit systems, and it does not support more than 2.5G of data;
2. The relatively large memory requirements for MongoDB, and it at least to ensure that the data (indexes, data and other system overhead) can be loaded into memory;
3. MongoDB has non-transactional mechanisms so that it can not guarantee the atomicity of events.

3.1.5 Conclusion

After a series of simple comparisons and experiments by my selves, I decide to choose MongoDB as the scientific NetCDF data storage because of its advantages and performance [24] [25].

3.2 Big Data Ocean Harmonisation

The BigDataOcean project is committed to using modern technological innovations and making rational use of them, which will revolutionize the way that the oceans and other related industries are used to operate. This innovation will create a whole new value chain that will have a huge impact in the future.

[7] http://bsonspec.org/
[8] https://www.json.org/
[8] http://www.bigdataocean.eu/site/

GeoDCAT[9] extends the properties of the DCAT[10] vocabulary, which is one of the most popular vocabularies to facilitate interoperability between the data catalogs on the Web, and it in order to describe geospatial data in the context of Big Data Ocean. The uses of GeoDCAT will enable the development of services that will harmonise and integrate seamlessly the underlying datasets [26].

The goal of the GeoDCAT is to allow seamless collection, harmonisation, and processing of cross-sectorial heterogeneous data, in order to be easily consumable by various types of stakeholders and Big Data Ocean services.

In Table 3.1, there is some General information about the elements of the metadata in GeoDCAT. And there are additional metadata elements, such as the vertical Coverage, the temporal Resolution and the temporal Coverage.

GeoDCAT Property	Description
dct:title	Dataset title
dct:description	Abstract describing the dataset
foaf:homepage	Online resource
dct:language	Dataset language
dct:spatial	Geographic location covered by the dataset
dct:publisher	Responsible party for the dataset
dct:accessRights	Limitations on public acce
...	...

Table 3.1: Metadata Elements in GeoDCAT

We will use the idea of the metadata here as an important reference for the metadata in our work which is introduced in chapter 4.

3.3 ANAPSID Query Engine

ANAPSID[11] is an adaptive query engine which is used to query SPARQL endpoints. In general, ANAPSID query engine adapts both the query execution schedulers to data availability and the run-time conditions. Meanwhile, ANAPSID query engine also provides some physical SPARQL operators, so that can help detect when a source becomes blocked or data tra□c is busty. Those physical SPARQL operators also produce the results as quickly as data arrives from the other sources.

The figure 3.1 shows the basic architecture of ANAPSID query engine. We can see that, the ANAPSID Query Engine is located between the user interface and the database

[9] https://joinup.ec.europa.eu/release/geodcat-ap-v10
[10] https://www.dcat.org/
[11] https://github.com/anapsid/anapsid

server. We can see that the ANAPSID Query Engine is located between the user interface and the database server. It has several di□erent components like query decomposer, query optimizer and adaptive query engine. The query decomposer and query optimizer are used to make a query planner when it receives the input query. The adaptive query engine is based on the architecture of wrappers and mediators. The work of the adaptive query engine is to query federations of SPARQL endpoints or the datasets in the database. In a word, the ANAPSID query engine accepts users' input queries and generates several SQL statements which will be sent to the database servers [27].

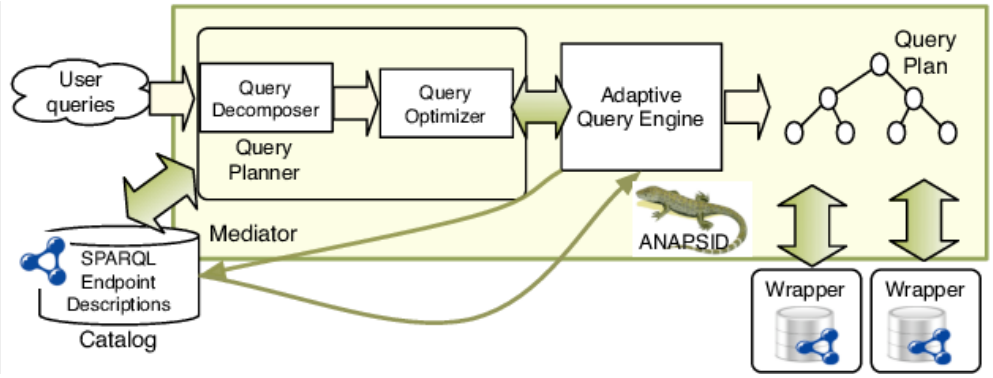

Figure 3.1: The ANAPSID Architecture

Here are the simple explanation and example of how the ANAPSID query engine works:

> *"The ANAPSID query engine provides a set of operators able to gather data from di erent endpoints. Opportunistically, these operators produce results by joining tuples previously received even when endpoints become blocked. Each join operator maintains a data structure called Resource Join Tuple (RJT), that records for each instantiation of the join variable(s), the tuples that have already matched. Suppose that for the instantiation of the variable $?x$ with the resource r, the tuples {T_1, ..., T_n} have been matched, then the RJT will be the pair (r, {T_1, ..., T_n}), where the first argument, head of the RJT, corresponds to the resource and the second, tail of the RJT, is the list of t" [27]*

After knowing the basic idea in ANAPSID query engine, we will focus on the decomposer part of the ANAPSID query engine and use it as the basic idea of my MGMDQE query engine which will be introduced in the chapter 4 in more detail.

Chapter 4

MGMDQE Query Engine

After introducing some related work in the last chapter 3, in this chapter, I'll focus on the MGMDQE query engine which is based on the decomposer part of the ANAPSID query engine. I introduce how the new query engine works step by step, and add metadata to improve the performance of the MGMDQE query engine. So that the MGMDQE query engine is able to meet the need which can be able to query scientific datasets transparently, without being aware of the available datasets.

4.1 NetCDF stored in MongoDB

After the different database comparison for NetCDF data storage in chapter 3, we decided to choose the MongoDB as the database for stored the NetCDF files. We will first use MongoDB to build the RDF database to store the data in NetCDF files.

4.1.1 Concepts Comparison of MongoDB and NetCDF

Because we choose the NetCDF as the data format of scientific datasets as our object, and we have already know the basic concepts in both MongoDB and NetCDF file so that we need to make clear that how to store the data of NetCDF into MongoDB. First of all, let's make a comparison of these concepts between MongoDB and NetCDF file, which must be useful to find the best way of storing the data in NetCDF file. In order to further understand better, we will compare them with the corresponding concepts in SQL.

SQL	MongoDB	NetCDF	Explanation
database	database	a file	the database
table	collection	variable	one table for one variable
row	document	dimension(1)	dimension 1 for the variable
column	field	dimension(2)	dimension 2 for the variable
index	index	attributes	explanation for variable and dimensions
value	value	data	data of the variable
primary key	primary key		MongoDB sets _id field as the primary key

Table 4.1: Concepts Comparison between SQL, MongoDB and NetCDF

In table 4.1, it shows some relationships between those relative concepts of SQL, concepts of MongoDB and concepts of NetCDF. Let's discuss more details about some important concepts in the table.

- Database In MongoDB, we can create multiple different and independent databases. For one MongoDB, it can hold multiple independent databases. In these databases, each of them has its own collections and permissions, and stores different data in different files. So that we can use only one MongoDB with several databases to store the data in several NetCDF files.
- collection(variable) A collection exists in the database and it is similar to the table in SQL. Collections in MongoDB have no fixed structure, which means that we can insert different data from different formats and types in one collection. Usually, the data which we insert into one collection must have some relationship, just like whose variables which correspond to real physical data will be in the same collection. Here is an easy example which will be used later in our thesis, the temperature and the pressure.
- document(dimension) A document of MongoDB is a set of key-value pairs. The document does not need to set the same fields nor the same data type, which is very different from the relational databases, it shows a very prominent feature of MongoDB here. A dimension of NetCDF file corresponds to an independent variable in a function, or an axis in a function image, which is a component of an N-dimensional vector in linear algebra (this is also the origin of the name of the dimension). In the NetCDF file, a dimension has a name and a range (or length, which is a mathematically defined domain, which can be a discrete set of points or a continuous interval).
- index(attributes) An attribute of NetCDF file annotates or interprets the value of a variable and the specific physical meaning of the dimension. Because variables and dimensions are just dimensionless numbers in NetCDF, to make people understand the specific meaning of these numbers, we have to rely on the attribute

object. In NetCDF, an attribute consists of an attribute name and an attribute value (usually a string).

4.1.2 Store NetCDF file in MongoDB

In the last several sections, we have already known the basic structure of the NetCDF file, the advantages of using MongoDB as the database for storing the data in NetCDF file, and the relationship between the concepts of MongoDB and NetCDF file, let's show the details about how to store the data in NetCDF file into MongoDB.

Because there exist some relations between MongoDB and NetCDF, we need to parse the Dimensions, the Variables the Attributes and the Data in the NetCDF file, so that we can store the data in NetCDF file into MongoDB with the structure.

Here we use PyMongo[1] and netCDF4[2]. PyMongo is a Python distribution containing tools for working with MongoDB, and is the recommended way to work with MongoDB from Python. Netcdf4-python is a Python interface to the NetCDF C library. We need to match the concepts in NetCDF file which the concepts in MongoDB as the table 4.1 shows. The Variable in NetCDF file matches the Collection in MongoDB, the Dimension in NetCDF file matches the Document and the Filed in MongoDB, and the Attributes in NetCDF file matches the Index in MongoDB, and the Data in NetCDF file matches the Values in MongoDB.

```
print(dimensions)
for item in variables_name:
    print()
    variable = dataset1.variables[item]
    print()
    collection = db[item]
    print()
    data = {
        'values': variable[:].tolist(),
        'dimensions':list,
        'name': variable.name
    }
    for attr in variable.ncattrs():
        if attr not in data.keys():
            if
            else:
    post_id = collection.insert_one(data)
```

[1] https://api.mongodb.com/python/current/
[2] http://unidata.github.io/netcdf4-python/

4.1.3 Examples of NetCDF in MongoDB

Let's use several example datasets of NetCDF files downloaded from Copernicus data repository.

Dataset "MO_201708_TS_MO_61277"

Dataset "MO_201708_TS_MO_68422"

Dataset "MO_201708_PR_PF_6903279"

Dataset "MO_201708_PR_PF_6903280".

The table for all databases in the MongoDB are shown in the table 4.2, and the table for all collections in the example dataset "MO_201708_PR_PF_6903279" is shown in the table 4.3.

Database	Storage Size	Collections	Indexes
MO_201708_TS_MO_61277	1.1MB	65	65
MO_201708_TS_MO_68422	1.1MB	65	65
MO_201708_PR_PF_6903280	232.0KB	13	13
MO_201708_PR_PF_6903279	228.0KB	13	13
...	...	...	...

Table 4.2: All datasets in MongoDB

Collection	Document	AvgSize	TotalSize	Indexes	TotalIndexes
DIRECTION	1	258.0B	258.0B	1	16.0KB
LATITUDE	1	462.0B	462.0B	1	16.0KB
LONGITUDE	1	465.0B	465.0B	1	16.0KB
PRES	1	20.1KB	20.1KB	1	16.0KB
PSAL	1	20.1KB	20.1KB	1	16.0KB
TEMP	1	20.1KB	20.1KB	1	16.0KB
TIME	1	352.0B	352.0B	1	16.0KB
...	1	...	...	1	16.0KB

Table 4.3: All collections in dataset "MO_201708_PR_PF_6903279"

In each Collection of one dataset, there is always an array (in some of the collections, there are several sub-arrays in the main array, such as the "PRES" collection) to store the values of this collection. let's take an easy example to have a look at the value and the indexes of one collection.

- 6 values of The Collection "TIME" in Dataset "MO_201708_PR_PF_6903279":

  0: 24685.217951388888

  1: 24690.221296296295

2: 24695.230208333334

3: 24700.23792824074

4: 24705.21747685185

5: 24710.216701388887

- 6 indexes of The Collection "TIME" in Dataset "MO_201708_PR_PF_6903279":

  name: "TIME"

  long_name: "Time"

  standard_name: "time"

  units: "days since 1950-01-01T00:00:00Z"

  valid_min: 0

  valid_max: 90000

4.2 Structure of MGMDQE Query Engine

The part that most interests to me is the query decomposer of ANAPSID query engine. As we have introduced in chapter 3, ANAPSID query engine what we have is used for querying SPARQL endpoints. However, we didn't choose endpoints to store the data in NetCDF file because we didn't find any available endpoints on the web. As we mentioned before, we decided to choose MongoDB as the carrier instead of the endpoints to store the data in NetCDF file, so we need to redesign our query engine based on the query decomposer of ANAPSID query engine.

Upon receiving a user query, the Query Parser in the Make Plan Step first parses the query and translates it from a string text into an intermediate data structure. Then the query engine in the Create Plan Step groups the query as the data structure and try to build a tree structure according to this intermediate data structure. In the Interact Proxy Step, the query engine uses several processes concurrently to deal with these tree structure of queries and send them to the database. The database server processes the SQL query and returns the result to the Query Engine. When the import query comes into the query engine, there are 3 main steps of the MGMDQE query engine to deal with the query.

4.2.1 Make Plan Step

The first step is called the Make Plan Step. In this step, the MGMDQE query engine mainly uses a tool called PLY (Python Lex-Yacc) to deal with the query. PLY is an

implementation of lex and yacc parsing tools for Python. PLY consists of two files : lex.py and yacc.py. The ply.lex part deals with the lexical analysis part, and the ply.yacc module is for creating a parser. There is a more detailed introduction of PLY tool in chapter 2.

In the Make Plan Step, the MGMDQE query engine uses the PLY tool in the $run\_decompose$ function to parse the input query language. The lex.py module is used to decompose the input characters into a sequence of tokens through a series of regular expressions, and yacc.py recognizes programming language syntax through some context-free grammars. These two tools are designed to work together. Lex.py serves as an interface by providing the token() method to the outside. Each time the method returns the next valid token from the input. Yacc.py will continue to call this method to get tags and match the grammar rules. Parsing is based on a parser called LR-parsing[3]. It is much faster with high memory e□cient, and it is better suited to large grammars.

In a few words, this step parses the input query form a string text to a defined data structure which can be recognized and deal with by the query engine in the next two steps.

4.2.2 Create Plan Step

The second step is called the Create Plan Step. After the Make Plan Step, the import query has been parsed into a defined data structure. In this step, MGMDQE query engine groups the query as the data structure and try to build a tree structure of the import query according to this data structure.

In computer science, a tree is an abstract data type (ADT) or a data structure that implements this abstract data type to simulate a collection of data with a tree structure. It consists of n (n > 0) finite nodes to form a hierarchical relationship to show its architecture.

In the Create Plan Step, MGMDQE query engine builds a binary query tree that corresponds to a relational algebra expression. We can know from its definition that, the binary query tree represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes [28]. The figures 4.1 is the tree structure which we will use for a simple query later.

[3] https://en.wikipedia.org/wiki/LR_parser

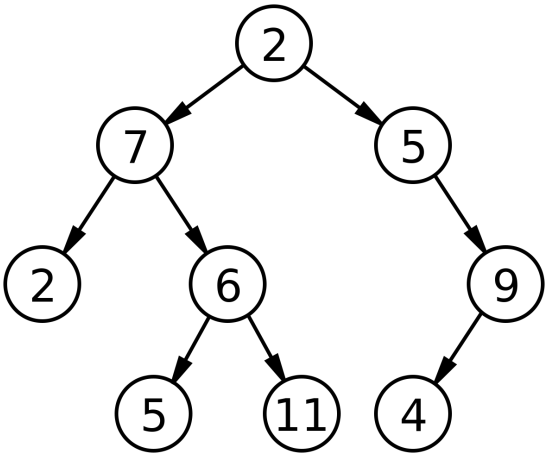

Figure 4.1: Binary Tree Structure

When the import query is complex enough, in this step, MGMDQE query engine will also try to break the complex entire query into several simple sub-queries first, and then execute several processes to deal with these sub-queries at the same time, which will greatly increase the e□ciency of the query engine. Each process deals with one sub-query, it tries to make one request for one sub-query to the next step.

4.2.3 Interact Proxy Step

The third step is called the Interact Proxy Step. MGMDQE query engine sends several HTTP requests by sockets concurrently to the MongoDB where stored the NetCDF files and listens to the responds.

A network socket[4] is an internal endpoint for sending or receiving data within a node on a computer network. Concretely, it represents the endpoints in networking software and it is just a data form of system resource.

[4] https://en.wikipedia.org/wiki/Network_socket

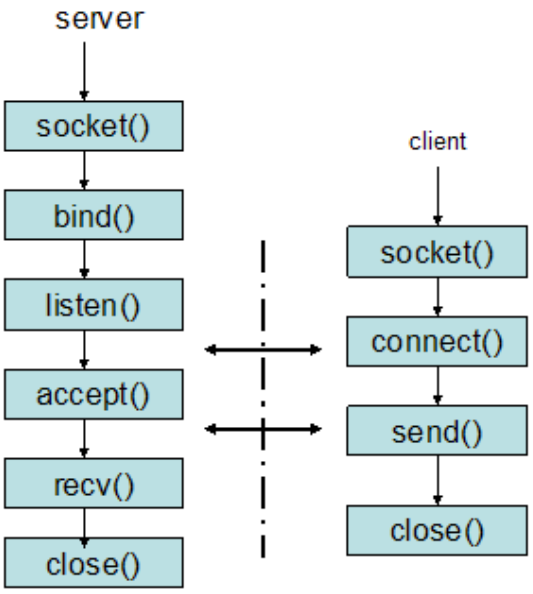

Figure 4.2: Sockets between server and client

The figure 4.2 shows how the network sockets work between the server side and the client side. First, the Client creates a socket and opens the socket and tries to connect to the server socket according to the server IP address and port number. Then the server socket receives the client socket request. At this time, the socket enters the blocking state. The blocking accept() method does not return until the client returns the connection information, and starts receiving the next client request. The client connects successfully and sends connection status information to the server and the server accepts method returns, the connection is successful. The client begins to write information to the socket and the server begins to read information. In the end, both the client side and the server side shut down.

In a word, after the Make Plan Step and the Create Plan Step, the import query has been the tree structure. In this step, according to the tree structures, query engine sends several requests which contain these tree structures by sockets to the MongoDB where stored the data in NetCDF files at the beginning. At last, it will get the response which contain the results back.

4.3 Display Di□erent Steps of Query Engine

In this section, it shows all the specific steps of the MGMDQE query engine. As what I have mentioned in the last section, here is a configuration of my query engine, the first step Make Plan Step, the second step Create Plan Step and the last step Interact Proxy Step.

4.3.1 Configuration of Query Engine

This section introduces the configuration part of the query engine. It is like the initialization of the query engine and It should be configured at the beginning.

Here is the configuration text:

```
-q
"/home/magiccream/Desktop/anapsid/queries/example"
-s
True
-p
ll
-o
False
-d
SSGM
-a
True
-w
False
-r
True
```

In this text, we can find some parameters about the configuration of the query engine at the beginning:

$-q$ is the absolute path of the text query file. In this file, it contains the prefix part at the beginning and the query body part. For example, the query file here we used:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT * WHERE {
  ?sub ?pred ?obj .
}
LIMIT 10
```

As we can find, there is a prefix part at the beginning, and the query body part shows the text query. $-s$ $-p$ $-o$ $-d$ $-a$ $-w$ $-r$ are some other system parameters in the query engine which are used to make sure the query engine work.

4.3.2 Step 1: Make Plan Step

This section introduces every step in the Make Plan Step of the MGMDQE query engine.

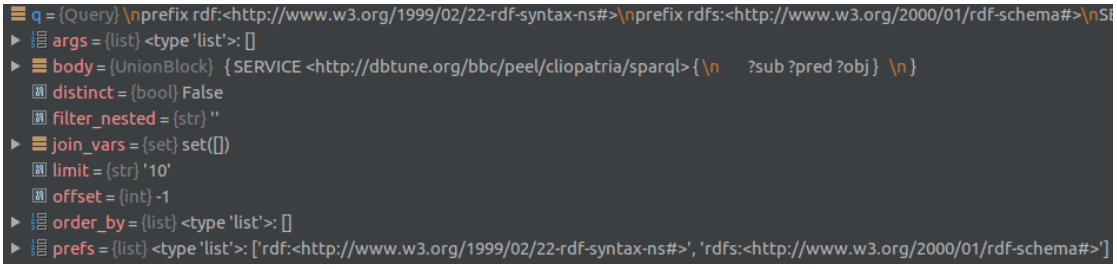

Figure 4.3: Query Structure

The figure 4.3 shows how is the structure of the test query after the configuration step. $q(Query)$ is the string of the test query. $limit(str)$ is the "LIMIT" condition in the body of the text query. $body(UnionBlock)$ is the main part of the test query. It is the original structure of the test query. Also this is the core step which the PLY (Python Lex-Yacc) tool works. $prefs(list)$ is a list to store the perfix part of the text query.

Let's get into the $body(UnionBlock)$ part.

Figure 4.4: Body UnionBlock

The figure 4.4 shows the original structure in the $body(UnionBlock)$. The most important parameter is the $triple(list)$.

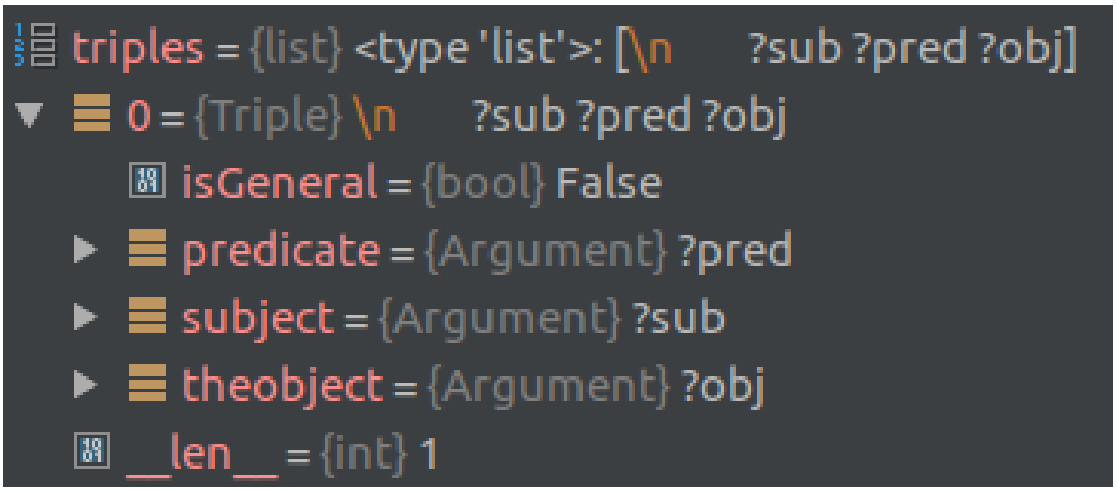

Figure 4.5: triple list

The figure 4.5 shows the $triple(list)$ in $body(UnionBlock)$. As we can see here, $triple(list)$ is a list to store the query language. $?pred$ is the $predicate(Argument)$ of the query. $?sub$ is the $subject(Argument)$ of the query, and $?obj$ is the $theobject(Argument)$ of the query.

Then, the PLY tool begins to work. It uses the lex.py module to decompose the input characters into a sequence of tokens through a series of regular expressions, and uses the yacc.py module to recognize the programming language syntax through some context-free grammars.

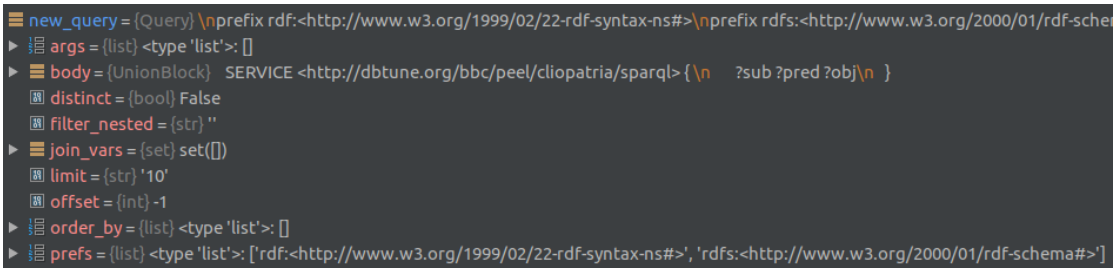

Figure 4.6: New structure

$New\_query(Query)$ in the figure 4.6 shows the new defined data structure of the text query after parsed by the PLY tool. $body(UnionBlock)$ is the core part of the new structure of the test query.

Let's see how the structure is in the new $body(UnionBlock)$ here.

Figure 4.7: New Body UnionBlock

The figure 4.7 shows a new data structure which has been parsed by the PLY tool in the Make Plan Step.

After this step, And it will be imported to the next part. The test query has been parsed by the PLY tools form its origin structure string text to a new defined data structure which can be recognized and deal with by the query engine in the next part.

4.3.3 Step 2: Create Plan Step

This section introduces every step in the Create Plan Step of the MGMDQE query engine.

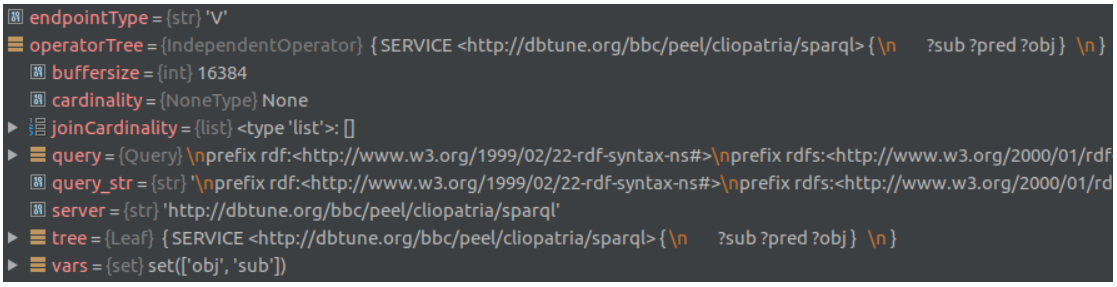

Figure 4.8: endpointType Structure

In the figure 4.8, $operaterTree(IndependentOperator)$ is the binary tree structure which is built following tree-build rules. The following $createPlan()$ function contains the di□erent rules to build the binary tree structure for di□erent queries.

```
def createPlan(query, adaptive, wc, buffersize, c, endpointType):
    endpType = endpointType
    operatorTree = OperatorsQuery(query, adaptive, wc, buffersize, c)
    if (len(query.order_by) > 0):
    if (query.args != []):
    if (query.distinct):
    if (query.offset != -1):
    if (query.limit != -1):
    return operatorTree
```

The following $TreePlan()$ represents a plan to be executed by the query engine. It is composed by a left node, a right node, and an operator node. The left and right nodes can be leaves to contact sources, or sub-trees.

In this $TreePlan()$, the operator node is a physical operator, provided by the engine. It creates a process for every node of the plan. The left node is always evaluated. If the right node is an independent operator or a sub-tree, it is evaluated.

```
class TreePlan(object):
    def __init__(self, operator, vars, left=None, right=None):
    def __repr__(self):
    def instantiate(self, d):
    def instantiateFilter(self, d, filter_str):
    def allTriplesLowSelectivity(self):
    def places(self):
    def constantNumber(self):
    def constantPercentage(self):
    def getCardinality(self):
    def getJoinCardinality(self, vars):
    def aux(self, n):
    def execute(self, outputqueue):
```

After this step, the new structure of the test query has become a binary tree structure that corresponds to a relational algebra expression by the query engine.

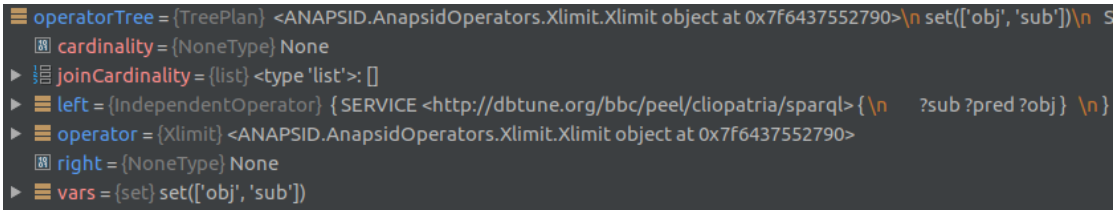

Figure 4.9: operatorTree Structure

The $operatorTree(TreePlan)$ in the figure 4.9 is the binary tree structure which has been created in the Create Plan Step. The $left$ and $tree$ shows all the nodes information which the tree structure has. After the Create Plan Step, it will be imported to the next step.

4.3.4 Step 3: Interact Proxy Step

This section introduces every step in the Interact Proxy Step of the MGMDQE query engine.

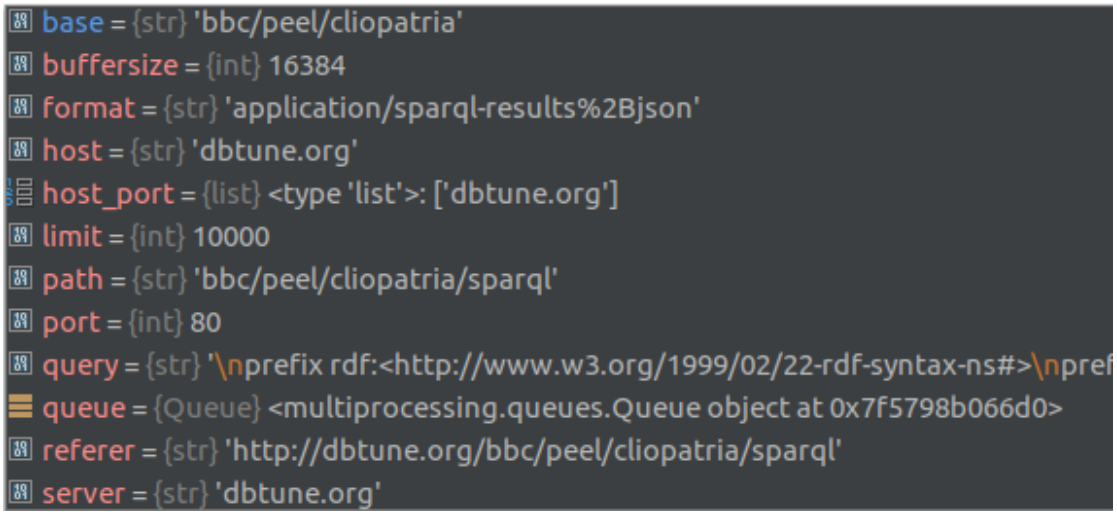

Figure 4.10: Connections parameters

Here are some parameters of connections in the figure 4.10, such as the port number which are used to connect the database in MongoDB.

Let see how the Interact Proxy Step sends several requests by processes to the database in MongoDB. Every tuple in the results is represented as Python dictionaries and is stored in a queue.

```
1  def contactProxy(server, query, queue, buffersize, limit):
2      mongo_endpoint = server.rsplit('//',2)
3      uri = urlparse(mongo_endpoint)
4      q = query.strip("\n").split('prefix')
5      target = query.split("WHERE").split("SELECT").strip().lstrip("?")
6      condition = query.split("WHERE").split("db").split()
7      where = query.split("WHERE").split("db").split()
8      where_float = float(where)
9      mongo_uri = 'mongodb:'.format(netloc, path, username, password)
10     client = pymongo.MongoClient()
11     return
```

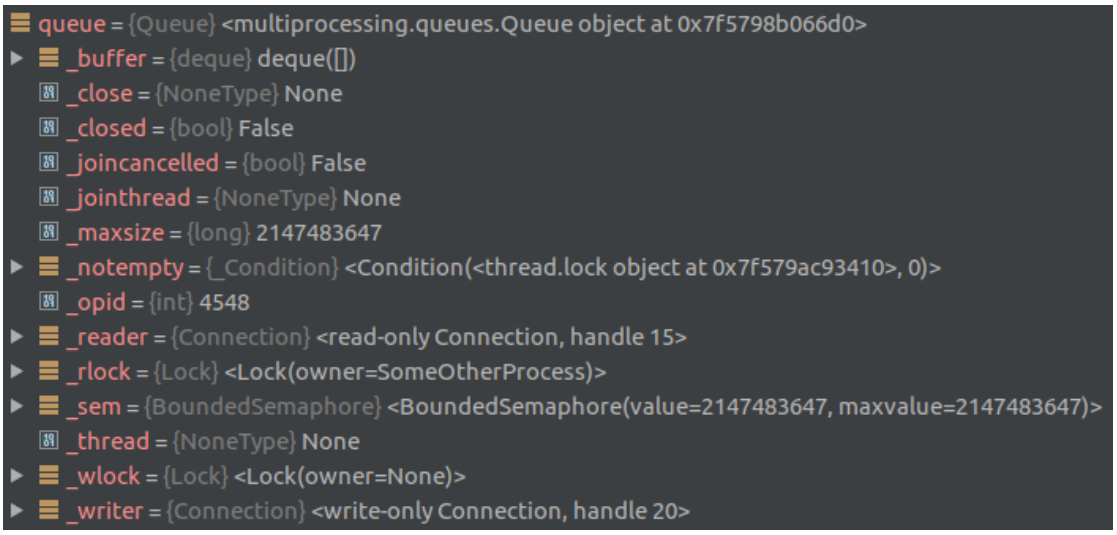

Figure 4.11: Query transfer Format

The figure 4.11 shows that the $Queue(Queue)$ is the test query which has already been transferred into the format by last two steps which can be called by the database on MongoDB. And it will be sent to the MongoDB by some concurrent processes using socket descriptor which is a type of handle. And let us take an example process in the figure 4.12 to see the structure of the sub-query in the process. There are some parameters in one of the example processes.

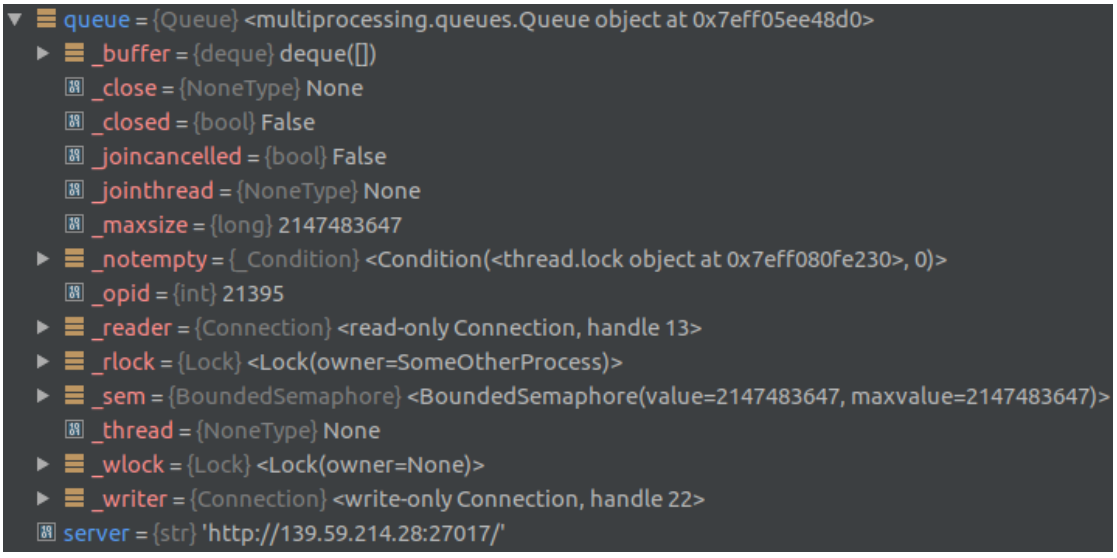

Figure 4.12: Subquery in process

At last, it shows Connected successfully and finally gets the results.

4.4 Metadata used in MGMDQE Query Engine

Metadata is the data information that provides information about other data. There are lots of metadata types, such as the descriptive metadata, structural metadata, administrative metadata, reference metadata and statistical metadata [29].

We can simply understand, the metadata is the smallest unit of data. Metadata is always used to describe data, such as its elements or attributes (name, size, data type, etc.), or its structure (length, field, data column),

Our problem is that being able to query or analyze scientific data without knowing about the underlying datasets are not at the moment possible. The metadata is the key to solving that problem. As we know, the metadata is a "data" which describes other data. That means if we can add the metadata between the second step Create Plan Step and the third step Interact Proxy Step, that we can analyze the components in the metadata, then we can get the useful information about the location of data, to guide the processes to the right position of the database in MongoDB in the Interact Proxy Step, then we can solve the problem. In that way, the query engine that will be able to query scientific datasets transparently, without being aware of the available datasets.

After finishing the whole three steps of the MGMDQE query engine, there are several processes take re-construct sub-queries, and before the query engine send these processes to all the database in MongoDB, there should exist the metadata, to guide the process should be sent to which exact database in MongoDB.

Here is the main idea of how to get all the information about the variables what we need in the metadata. The appendix B shows the excerpt of Example Metadata.ttl file. In the metadata.ttl file what we used, there is some information about the variable which contains where the variable of the data is in which database, which is most interests us. We use a tool called RDFLib [5] to get the information from metadata which we need here. RDFLib is a Python library for working with RDF, a simple yet powerful language for representing information as graphs.

```
1 def get_db_form_meta(path, var_name):
2     meta_path =
    q[2].split("<")[1].split(">")[0].rstrip("/").replace("file://","")
3     g = Graph()
4     g.parse(path, format='n3')
5     name = rdflib.term.Literal(var_name)
6     q = prepareQuery(
7         'SELECT ?s WHERE { ?s a <http:X> . ?s <http:Y> ?name.}'
```

[5] https://github.com/RDFLib/rdflib

```
    )
    rlt = []
    for row in g.query(q, initBindings={"name": name}):
        rlt.append(row.s.rsplit("/", 1)[1])
    return rlt
```

And then here shows how to compare the variables in sub-query to the variables in the metadata roughly.

```
dbs = get_db_form_meta(meta_path, target)
    rlt = []
    for record in dbs:
        db_name = record.rsplit("_",1)[0]
        collection_name = record.rsplit("_",1)[1]
        db = client[db_name]
        data = db[collection_name].find_one()
        for num in data["values"]:
            if num > where_float:
                rlt.append(num)
    res = {}
    res[target] = rlt
    queue.put(res)
    queue.put("EOF")
    return
```

After above steps, the query engine will get the information about the variables after step 2 Create Plan Step and put the information of the positions of the variables into their own processes in step 3 Interact Proxy Step, then the query engine can send these processes to their database directly after step 3.

Decided to use the metadata here must be one of the main contributions in my thesis, it basically solves the problems that most of the query engines can't solve now. That is the being able to query or analyze scientific data without knowing about the underlying datasets. Currently, the next work should be to optimize the frames which the query engine is using now and to improve the performance deeply.

4.5 Defects and Problems of MGMDQE Query Engine

In the last several sections, we have already known the Architecture of MGMDQE Query Engine introduces how is the query engine works based on the query decomposer part of ANAPSID query engine, and shows more details about every step of my query engine.

However, there exist some defects in the MGMDQE query engine. By using the keywords identification, the query engine has already decomposed the query into several simple sub-queries successfully. These sub-queries are the simple query text which has only one variable with one condition. Then the query engine can deal with and parser these sub-queries in a data structure into the second part of the query engine, Create Plan part and keep going. Then, the query engine tries to enter the last part of the query engine, Interact Proxy part. In this part, the query engine executes several concurrently executing processes which contain the sub-queries in a data structure, sends these processes to different datasets according to metadata, listens to the response from the datasets and gets the results back. However, there exists a serious problem. Because these several processes are executing concurrently, so the query engine must meet some troubles if some of the variables which have some relationship between these sub-queries.

At last, let's take an example to show how is the problem exists. If the users want to query the pressure of the time when the time is earlier than 24690.221296296295. And these two variables $PRES$ and $TIME$ are stored in two datasets:

Dataset1: "MO_201708_TS_MO_61277". Dataset2: "MO_201708_TS_MO_68422".

```
SELECT PRES
FROM "MO\_201708\_TS\_MO\_61277" and "MO\_201708\_TS\_MO\_68422"
WHERE PRES.time = TIME
      and
      TIME < 24690.221296296295;
```

Then it will get two sub-queries by the query engine, then we get the sub-query1 and the sub-query2, and these two sub-queries will be sent to the next steps of the query engine.

1. sub-query1:

```
SELECT PRES
FROM "MO\_201708\_TS\_MO\_61277" and "MO\_201708\_TS\_MO\_68422"
WHERE PRES.time = TIME;
```

2. sub-query2:

```
SELECT TIME
FROM "MO\_201708\_TS\_MO\_61277" and "MO\_201708\_TS\_MO\_68422"
WHERE TIME < 24690.221296296295;
```

These two sub-queries will be sent to the database in MongoDB by two processes which are executing concurrently in the Interact Proxy Step. But for the sub-query1, because

the value of b should be decided by the sub-query2 in another process which is executing concurrently, so for this sub-query, the value of b is unknown, then this process must get an error.

In a word, in the current situation, the query engine can only deal with some limited query types. The query must only have one variable with several conditions or several simple variables but without any relationship between these variables. However, I'm clear that this is not enough. We need to find a new solution for those complex queries which the query engine can't deal with how to improve the performance. That solution can be a very good and important supplement for the query engine.

4.6 Conclusion

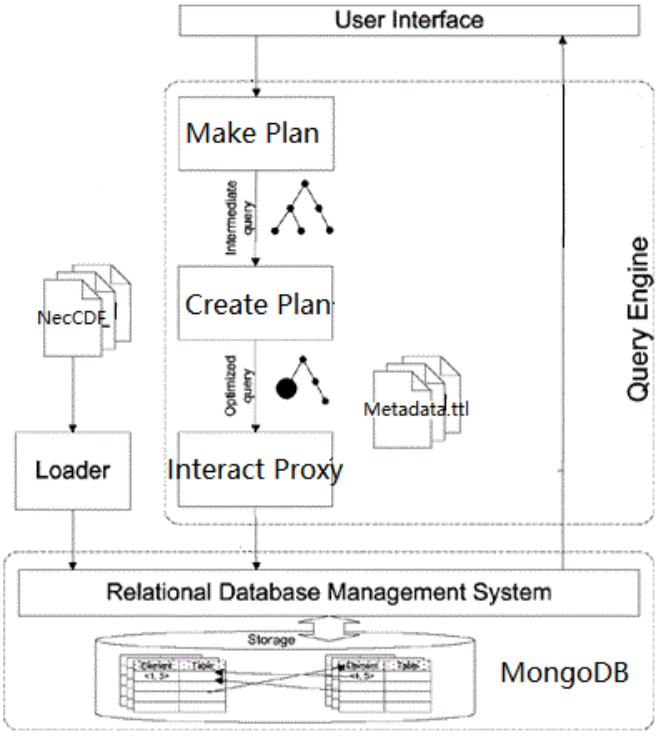

Figure 4.13: Structure of Query Engine

In this section, as we can see in the figure 4.13, we create a new query engine called MGMDQE query engine which is based on the query decomposer part of the ANAPSID query engine. It consists of three different steps, the first step is to parse the input query, the second step is to build a tree structure, the third step is to send the request to the database. In order to meet the requirements, we add the metadata before the third step. The query engine try to get the information in the metadata file and use the information to guide the requests.

However, there are still some problems and defects in the query engine such as the complex queries. So that we try to find a new idea to improve the query engine.

Chapter 5

GraphQLQE Query Engine

In this chapter, I will first explain the motivations for designing the new GraphQLQE query engine, then show the structure of the GraphQLQE query engine and explain the schema of the new query engine. Finally, use a simple example to illustrate the advantages and disadvantages of the new query engine. This is one of the most important contributions of my thesis.

5.1 Motivations of new query engine

As we mentioned in the last section of the Chapter 4, we find a defect of MGMDQE query engine we have when I do the experiment. When we try to test some complex queries, which have more than one variables and there exist some relationship between these variables, the MGMDQE query engine always meets some problems.

First, I try to find and solve the problems which MGMEQE query engine has now. However, because these processes are concurrently executed, so if one process needs the results from another process, then the problem comes. I try to add some control conditions to these processes. The new problem comes, if there are too many control conditions needed for the concurrent process, it will be a heavy burden for the query engine, which will a□ect the performance of the query engine.

I change my mind and decide to design a totally new query engine to solve the problem. In my plan, the new query engine should not only query scientific datasets transparently, without being aware of the available datasets but also meet almost all the types of queries.

5.2 Structure of GraphQLQE query engine

I choose Graphql as the input query language of my new query engine. As we know, GraphQL isn't tied to any specific database or storage engine and is instead backed by the existing code and data, so that the flexibility is one of the most advantages in GraphQL. After I finish parser the Graphql, I write several schemas of my own in order to query the data in the NetCDF files which are stored in the MongoDB. The ideas and the structures in the schemas are the cores of the new query engine.

Let's see how is the GraphQL look like. A GraphQL service is created by defining several types and fields on those types which are similar to Flow and TypeScript that are continuously evolved to the data. Then, it provides several functions for each field on each type. For example, a GraphQL service that tells us the project as well as that user's name who is using this project. It might look something like this:

```
type Project {
    name: String
    tagline: String
    contributors: [User]
}
    type User {
    name: String
    photo: String,
    friends: [User]
}
```

Next, we can use GraphQL's query language (Queries) as an object with no value and only attributes. The result returned is the object with the corresponding value, which is the standard JSON.

```
Which data we want // based on Queries {
    // find project whose name is GraphQL
    project (name: "GraphQL") {
    tagline
    }
}
Get predictable results {
    // return json
    "project": {
    "tagline": "A query language for APIs"
    }
}
}
```

Although the project defines three different fields in the type system, we (client) only need the tagline field, the server only returns this field. The (query) does not care about the User and its corresponding fields in the contributors in this query.

Meanwhile, the functions for each field on each type are like this:

```
function Queryme(request) {
  return request.auth.user;
}
function Username(user) {
  return user.getName();
}
```

As we can find, it is the primary entry point function for fulfilling GraphQL operations by parsing, validating and executing a GraphQL document alongside a GraphQL schema.

```
def execute_graphql(
    try:
        if backend is None:
        backend = get_default_backend()
        document = backend.document(schema, request_string)
        return document.execute(
            root=root,
            context=context,
            operation_name=operation_name,
            variables=variables,
            middleware=middleware,
        )
```

In the figure 5.1, we can see that a rough structure of the GraphQLQE query engine and know how the GraphQLQE query engine works. After it gets the users input GraphQL query, there are some schema stitchings in the GraphQL Proxy, which is the process of creating a single GraphQL schema from multiple underlying GraphQL APIs by its id or $name$. Then the GraphQL query is divided into several sub-GraphQL-server to query the data in different MongoDBs [30]. The results will gather by the GraphQL proxy again and get the result.

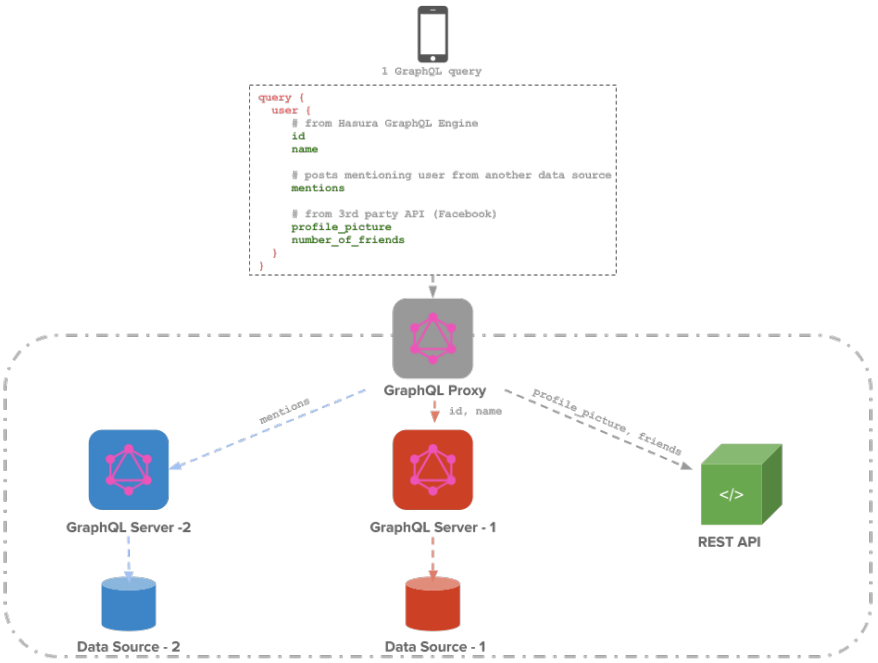

Figure 5.1: How GraphQl Works

5.3 Schemas of GraphQLQE query engine

After introduction how is the GraphQL as the input query for our query engine and how is the GraphQLQE query engine works, let's focus on the schemas in the GraphQLQE query engine, which is the most important part of the new query engine. A Schema in GraphQL is created by supplying the root types of each type of operation, query and mutation. These root types are totally optional so that they are self-designed according to di□erent input GraphQL query.

- Schema: The $Schema$ will be used by the GraphQL type system when there is a validating and executing query coming.
- requestString: The $requestString$ is a GraphQL language formatted string, which is used to represent the di□erent requested operations.
- rootValue: The value of the $rootValue$ is provided as the first argument to resolver functions. These resolver functions are always on the top level type such as the query object type.
- variableValues: The $variableValues$ is a mapping of variable name to runtime value which is used for all variables. These variable names are usually defined in the previous $requestString$.

- operationName: The $operationName$ is the name of the operation which is used in $requestString$ and if $requestString$ contains multiple possible operations in one GraphQL language formatted string. The $operationName$ can be omitted sometime if $requestString$ contains only one operation.

Every single GraphQL service defines a set of types by itself, which completely describes the set of possible data. The data can be queried on the service. Which means, when the queries are coming, they are validated and executed in a single schema. The server provides a GraphQL schema that defines the hierarchical relationships and types of available data, and the client only needs to query the data it needs.

The GraphQL service can be written in any language because we don't rely on the syntax of any particular language to communicate with the GraphQL schema. We define our own simple language, called "GraphQL schema language", which is similar to GraphQL's query language, allowing us to communicate with GraphQL schema without language differences.

Let's take an example schema to show how is the structure of the schema in the query engine. Here this function which lets the developer select a type in a given schema by accessing its $attrs$. Example: using schema. Query for accessing the "$Query$" type in the Schema

```
def __getattr__(self, type_name):
    _type = super(Schema, self).get_type(type_name)
    if _type is None:
    if isinstance(_type, GrapheneGraphQLType):
        return _type.graphene_type
    return _type
```

```
class GrapheneGraphQLType(object):
    def __init__(self, *args, **kwargs):
        self.graphene_type = kwargs.pop("graphene_type")
        super(GrapheneGraphQLType, self).__init__(*args, **kwargs)
```

And for our following example, there are two variables in our example, we need to find the variable $time$ and the variable $PresB$. Let's have a look at the structure of these schemas here.

Here is the scheme for the variable $time$:

```
class Time(graphene.ObjectType):
    name = graphene.String()
```

```
    values = graphene.List(graphene.Float)
    units = graphene.String()
    dimensions = graphene.List(graphene.String)
    indexes = graphene.List(graphene.Int)
```

```
class Query(graphene.ObjectType):
    time = graphene.Field(Time, dataSet):
    def resolve_time(self, info, **args):
        client = pymongo.MongoClient(mongoURI)
        db = client[args.get('dataSet')]
        coll = db["TIME"]
        f = {}
        s = {}
        doc = coll.find_one()
        vals = doc["values"]
        idxs_l = range(len(doc["values"]))
        print(idxs_l)
        if
        else;
        return Time(values= vals,
                    name= doc["name"],
                    units= doc["units"],
                    dimensions=doc["dimensions"],
                    indexes=idxs_l)
```

As we can find, the query engine will construction requests and find corresponding collections, documents and values in MongoDB according to the schemas. Here is another schema for the variable $PresB$:

```
class Pres(graphene.ObjectType):
    name = graphene.String()
    values = graphene.List(graphene.List(graphene.Float))
    units = graphene.String()
    dimensions = graphene.List(graphene.String)
```

```
class Query(graphene.ObjectType):
    pres = graphene.Field(Pres, dataSet):
        print(args)
        client = pymongo.MongoClient(mongoURI)
        db = client[args.get('dataSet')]
        coll = db["PRES"]
        f = {}
        s = {}
```

```
        doc = coll.find_one()
        vals = doc["values"]
        if
        return Pres(values=vals,
                    name=doc["name"],
                    units=doc["units"],
                    dimensions=doc["dimensions"])
```

In general, as we can find, a single schema here is not very complex, but we need a lot of different kinds of schemas according to different kinds of input queries, meanwhile, we can also add other more schemas in the future.

5.4 Advantage and Disadvantages of new query engine

In my opinion, the GraphQLQE query engine can handle almost all kinds of different queries, not only the simple query which has only one variable or several variables without any relationships, but also the complex query which has several variables with a lot of relationships. The only thing what we need to do is to parser the GraphQL query first, and to check if our query engine can deal with this kind of query now. If not, we need to write some new schemas into the decomposer part here according to the structure of GraphQL after parsing the query to deal with new queries.

Let's take the same example in Chapter 4 to show how is my new query engine works. If the users want to query the pressure of the time when the time is earlier than 24690.221296296295. And these two variables $PRES$ and $TIME$ are stored in two datasets: the Dataset "MO_201708_TS_MO_61277" and the Dataset "MO_201708_TS_MO_68422"

```
result = execute(query2, variable = {"time", query1})

WHERE
query1{
    time("MO\_201708\_TS\_MO\_68422", lt:24690.221296296295){
    values
    indexes}
}

AND
query2{
    pres("MO\_201708\_TS\_MO\_61277", time;$time,$){
    name
    values}
```

```
15 }
```

Here we can find the test query is divided into two sub-queries, the sub-query $query1$ is just like a condition query which is the condition of the test query, it queries all the results of time which is less than B in the $dataset2$. And the sub-query $query2$ likes the main query of the test query, it queries all the results of A in $database1$ whose time is equal to the time value in $query1$.

The final results of the example query are shown in the figure 5.2.

```
'pres', OrderedDict([('name', 'PRES'), ('values', [[2.0999999046325684, 3.0,
```

Figure 5.2: Results of example

5.4.1 Advantages of new query engine

We can easily find the biggest advantage of the new query engine from the above example, which is the maximum flexibility and the scalability. In my opinion, the flexibility and the scalability here can be divided into the following two aspects.

On the one hand, flexibility is for the user. Users can query almost all the types of the query through our GraphQLQE query engine. No matter how complicated the query is, as long as the user can sort out the relationship of the sub-queries in the query, translate their query language to GraphQL and use this GraphQL query as the input query for the GraphQLQE query engine, then the user can get the results from the query engine.

Scalability is another advantage for the GraphQLQE query engine. We can write di□erent new query schemas in the query engine according to the di□erent types of queries what the query engine get, to update the schemas in our query engine. We can improve the performance of the GraphQLQE query engine so that the query engine can satisfy almost all types of queries, it makes the scalability of the query engine.

5.4.2 Disadvantages of new query engine

However, flexibility and scalability also mean relative complexity. And this complexity is precisely the biggest disadvantage of the GraphQLQE query engine. This complexity can also be divided into the following two aspects.

On the one hand, this complexity is for the users. Before using our new query engine, users need to be familiar with and try to use GraphQL as their input query language, which means it requires the users to convert other query languages into GraphQL as the

input to the new query engine by themselves before, that would be not very easy for a lot of users.

On the other hand, this complexity also applies to the GraphQLQE query engine. Since the types of queries are di□erent, this requires us to constantly add new patterns and schemas based on the new sub-query relationships included in the GraphQL, thus improving our query engine, allowing the new query engines to handle all kinds of queries, which must be a long and complicated work for us.

And the biggest disadvantage of the new query here is, as we can find in the example in the last section because we haven't use metadata here, so the users still have to know which database the variables are stored. However, this problem has returned to the original problem. So in my plan, we should combine the idea in the GraphQLQE query engine with the MGMDQE query engine which has been introduced in Chapter 4.

5.4.3 Conclusion

In the section, We find a new idea about how to deal with some complex queries and build a new query engine called GraphQLQE query engine according to this idea. The new query engine uses GraphQL as the input query engine because of its flexibility and scalability. In the new query engine, there are a lot of self-designed schemas, so that it can be used to query almost all kinds of query types. However, the metadata file has not been used in the new query engine yet, so it goes back to the main problem again.

In my opinion, we need to pay more attention to this new query engine in the future because of its advantages and better performance.

Chapter 6

Experiments

6.1 Introduction of Experiment

The experiment was performed on a laptop, which equipped with an Intel(R) Core(TM) i7-4700HQ CPU @2.40GHz, Samsung SSD 850 EVO 500GB, 8.0 GB RAM, running on the Ubuntu 18.04.1 LTS which equipped with 4.0GM RAM and 20GB SCSI. It is connected via the Gigabit Ethernet. The experiment is composed of two parts according to two di□erent kinds of query engine in my thesis.

The experiments were conducted 6 NetCDF.nc files downloaded from Copernicus data repository[1] and 1 metadata.tll file. The Query Templates is provided by the Waterloo SPARQL Diversity Test Suite (WatDiv)[2] that covers all di□erent query shapes and thus allows us to test the performance of our 2 query engines in a more fine-grained way. WatDiv comes with a set of 20 predefined query templates called Basic Testing use case which can be grouped in four categories according to their shape: star (S), linear (L), snowflake (F) and complex (C).

Because we have small data of only 6 datasets in our experiment, so we just choose 2 linear queries, 1 star query, 1 snowflake query as "simple queries" and 1 complex query according to the query templates in WatDiv randomly. The query templates which are used for the experiment are shown in the table 6.1. There will be more details of the query templates which are used in the experiment in Appendix C. We do the experiment in a laptop with the properties which are introduced in the beginning.

In the experiment, we will do the experiment five times for each query template with 5 di□erent variables, and to record the accuracy of results and the spend time of two query

[1] http://www.copernicus.eu/
[2] https://dsg.uwaterloo.ca/watdiv/

	Linear	Star	Snowflake	Complex
Query Templates	L3,L4	S6	F5	C3

Table 6.1: Query Templates

engines. The resulting accuracy and the spend time of the query engine are the results in the experiment what we need.

6.2 Experiment

Although the query language of the two query engines is different, this does not affect us to do our experiment in the same way. Therefore, we put the experimental results of the two query engines together to make a more intuitive comparison. We design a simple table to record the results of two query engines in the experiment.

Query Templates	MGMDQE Query Engine	GraphQLQE Query Engine
Linear Query L3	✓	✓
Linear Query L4	✓	✓
Star Query S6	✓	✓
Snowflake Query F5	✓	✓
Complex Query C3	×	✓

Table 6.2: Results of Query Engine deal with query templates

In the table 6.2, it shows the results that if the query engine can deal with different query templates or not. As we can see in the table, Both of two query engines in the experiment can deal with the linear query, the star query end the snowflake query. However, for the complex query, two query engines perform total differently. The MGMDQE query engine will get no result and report an error, but the GraphQLQE query engine can still deal with it.

Query Templates	MGMDQE Query Engine	GraphQLQE Query Engine
Linear Query L3	430	367
Linear Query L4	428	398
Star Query S6	535	326
Snowflake Query F5	653	405
Complex Query C3	ERROR	507

Table 6.3: Average Time Spend of Query Engine deal with query templates (in ms)

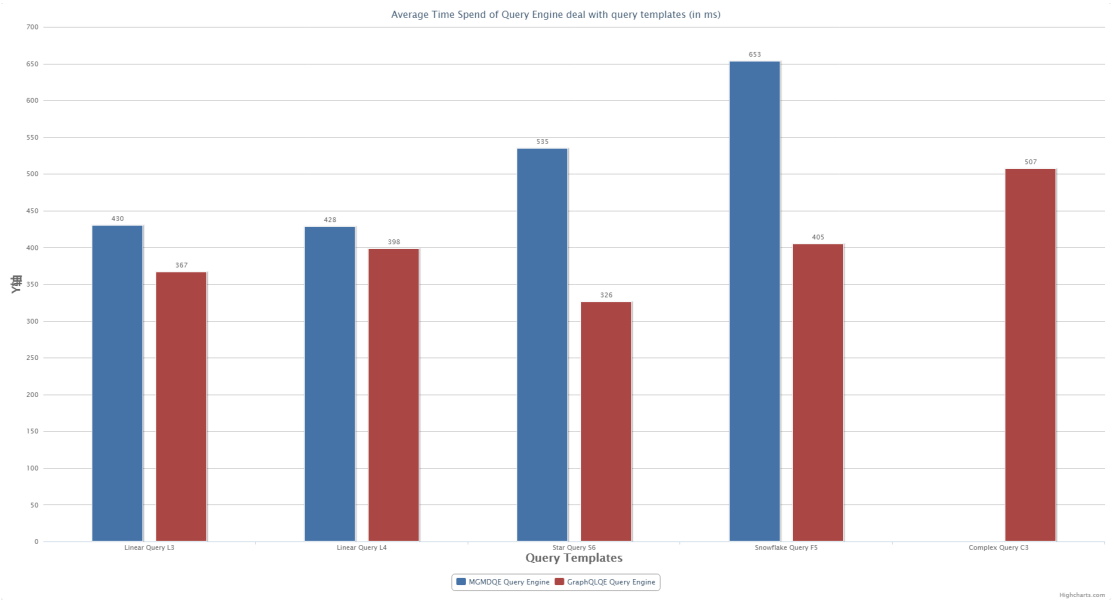

Figure 6.1: Average Time Spend of Query Engine deal with query templates (in ms)

In the table 6.3 and the figure 6.1, they show the average time (in ms) of five times that does the query engine can deal with different query templates. As we can see in the table, because the experiment is in a very simple way, which only have 5 query templates and 6 datasets, so the time spent of MGMDQE query and GraphQLQE query engine are very close. We need to do the experiment with more query templates and datasets to get the results about the performance of the two query engines.

6.3 Results and Conclusions

In the experiment, we can easily find that the performance of the MGMDQE query engine and the GraphQLQE query engine is acceptable which is in the previous expectation before the experiment. For several simple queries such as the linear query, the star query and the snowflake query, both of MGMDQE query engine and GraphQLQE query engine can deal with. However, for the complex query, MGMDQE query engine gets an error but GraphQLQE query engine can still get a result. That is the biggest difference between the two query engines. Generally, the results in the experiment basically met our guess before the experiment. Meanwhile, the experiment gives us a lot of help and also tells us what we need to do to improve the performance of both two query engines in the future.

However, there exist some problems in the experiment. The main problem in the experiment is that, in a sense, this experiment is too simple. We only have 6 NetCDF files as datasets and one simple metadata.ttl file to use in the experiment. In my opinion, I try

to do the experiment in a perfect way with more NetCDF files but it must have some limitations because the less of the experimental data. Then another huge problem comes, when I try to add or delete a NetCDF file as new datasets, the metadata.ttl file always needs to be changed, so it's really difficult for me to get the metadata file constantly. It must be another important work for me in the future.

Chapter 7

Conclusion and Future Work

7.1 Summary

In this thesis, we first build a query engine called MGMDQE query engine based on the query decomposer part of the ANAPSID query engine. For solving the main problem, we optimize the three steps of the query decomposer and decide to add a metadata file into my query engine. The information of the data and the variables in metadata can guide the processes in the MGMDQE query engine to be sent to which database in MongoDB. So that the user can use the MGMDQE query engine to query or analyze scientific data without knowing about the underlying datasets in MongoDB.

However, during testing the MGMDQE query engine, I find a defect when I try to query some complex queries by it. Then I change my mind and try to build a totally new query engine called GraphQLQE query engine. It uses GraphQL as the query language, and there are a lot of self-designed schemas in the new query engine, which are used to query different types of input queries. The GraphQLQE query engine can be used to query almost all types of queries. However, it hasn't used the metadata file in the query engine so that the main problem of our thesis comes again.

Our experimental results basically verify our thoughts and understanding of both two query engines before the experiment. For several simple queries, both of the MGMDQE query engine and the GraphQLQE query engine can deal with. However, for those complex queries, the MGMDQE query engine gets an error but the GraphQLQE query engine can still get a result correctly. According to the results in the experiment, We need to solve the exist problems and improve the performance of both two query engines in the future.

7.2 Further Work

We have designed two new query engines in our thesis, for future work, all is about in order to solve the problems, improve the performance and extend the functionality of both the MGMDQE query engine and the GraphQLQE query engine. Therefore, our work should be developed in a number of ways in the future:

- Make MGMDQE query engine work for complex query: As we can see the results in the experiment, MGMDQE query engine can only deal with the simple queries, but not the complex queries. So the most important work in the future for MGMDQE query engine is to make it work for both simple queries and complex queries. We can adopt the main idea in GraphQLQE query engine which is the flexibility. So that the MGMDQE query engine may get a much better performance dealing with complex queries in the future.
- Add metadata to GraphQLQE query engine: Also as we have found in the results of the previous experiment, the performance of the GraphQLQE query engine in the experiment is acceptable for us. It can not only deal with simple queries, but also the complex queries. However, the main issues here is that it doesn't solve the main problem in our thesis. So the most important work in the future for GraphQLQE query engine is to add the metadata file and make use of it just like what we have done in the MGMDQE query engine. Meanwhile, we can extend the input query language from only GraphQL to some other query languages, which must be more convenient for all the users. Furthermore, we can make the GraphQLQE query engine customization for all the users, which means that users can write their new schemas according to their new input query types by themselves. In this way, it can maximize the flexibility and scalability of the query engine.
- Evaluate two update query engines again: After the previous future work on two new query engines, we should do a new evaluation of these two updated query engines again in the same way as what we have done before. Compared to previous experiments, we should make this new experiment more complete, which means we must have more NetCDF files as datasets, one metadata file, and more query templates in the experiment. We should do the same experiment on some other existed query engines too so that it can make sure the results in the experiment are more credible. After the experiment, we can make a more complete evaluation of these two updated query engines to find which query engine has better performance in which query template.

- Pay more attention to GraphQLQE query engine: In my opinion, we should pay more attention to the GraphQLQE query engine in the future, not only because of its advantages and better performance, but also the new idea.

Appendix A

Excerpt of NetCDF file

In this appendix, here shows a simple excerpt of one NetCDF.nc file which is read by MATLAB: NetCDF file: "MO_201708_TS_MO_61277.nc" (Removed some duplicate parts).

```
Source:
           D:\Studys\Master Thesis\nc_datasets\MO_201708_TS_MO_61277.nc
Format:
           classic
Global Attributes:
           data_type                 = 'OceanSITES time-series data'
           format_version            = '1.2'
           platform_code             = 'E1M3A'
           platform_name             = ' '
           ices_platform_code        = ' '
           institution               = 'HELLENIC CENTER FOR MARINE RESEARCH (HCMR)'
           institution_edmo_code     = '164'
           date_update               = '2017-09-01T10:35:59Z'
           site_code                 = 'E1M3A'
           wmo_platform_code         = '61277'
           source                    = 'BUOY/MOORING: SURFACE, MOORED: observation'
           history                   = '2017-09-01T10:35:59Z : Creation'
           data_mode                 = 'R'
           quality_control_indicator = '6'
           quality_index             = 'A'
           references                = 'http://www.oceansites.org'
           comment                   = ' '
           Conventions               = ''
           netcdf_version            = '3.5'
           title                     = 'Med Sea - NRT in situ Observations'
           summary                   = ' '
           naming_authority          = 'OceanSITES'
           id                        = 'MO_201708_TS_MO_61277'
           cdm_data_type             = 'Time-series'
           area                      = 'Mediterranean'
           geospatial_lat_min        = '35.729'
           geospatial_lat_max        = '35.729'
           geospatial_lon_min        = '25.1202'
```

```
            geospatial_lon_max         = '25.1202'
            geospatial_vertical_min    = '-3.0'
            geospatial_vertical_max    = '1000.0'
            time_coverage_start        = '2017-08-01T00:00:00Z'
            time_coverage_end          = '2017-08-31T21:00:00Z'
            institution_references     = 'http://www.hcmr.gr'
            contact                    = 'achalk@hcmr.gr, cmems-service@hcmr.gr'
            author                     = 'Antonis Chalkiopoulos'
            data_assembly_center       = 'HCMR'
            pi_name                    = 'Leonidas Perivoliotis'
            distribution_statement     = ''
            citation                   = ''
            update_interval            = 'daily'
            qc_manual                  = 'OceanSITES User's Manual v1.2'
            last_latitude_observation  = '35.729'
            last_longitude_observation = '25.1202'
            last_date_observation      = '2017-08-31T21:00:00Z'
            wmo_inst_type              = ' '
Dimensions:
            TIME      = 248   (UNLIMITED)
            LATITUDE  = 248
            LONGITUDE = 248
            POSITION  = 248
            DEPTH     = 19
Variables:
    TIME
            Size:       248x1
            Dimensions: TIME
            Datatype:   double
            Attributes:
                        long_name     = 'Time'
                        standard_name = 'time'
                        units         = 'days since 1950-01-01T00:00:00Z'
                        valid_min     = 0
                        valid_max     = 90000
                        QC_indicator  = 1
                        QC_procedure  = 1
                        uncertainty   = ' '
                        comment       = ' '
                        axis          = 'T'
    LATITUDE
            Size:       248x1
            Dimensions: LATITUDE
            Datatype:   single
            Attributes:
                        long_name                 = 'Latitude of each location'
                        standard_name             = 'latitude'
                        units                     = 'degree_north'
                        _FillValue                = 99999
                        valid_min                 = -90
                        valid_max                 = 90
                        QC_indicator              = 1
                        QC_procedure              = 1
                        uncertainty               = ' '
                        comment                   = ' '
```

```
                    axis                        = 'Y'
                    reference                   = 'WGS84'
                    coordinate_reference_frame  = 'urn:ogc:crs:EPSG::4326'
    LONGITUDE
           Size:       248x1
           Dimensions: LONGITUDE
           Datatype:   single
           Attributes:
                    long_name                   = 'Longitude of each location'
                    standard_name               = 'longitude'
                    units                       = 'degree_east'
                    _FillValue                  = 99999
                    valid_min                   = -180
                    valid_max                   = 180
                    QC_indicator                = 1
                    QC_procedure                = 1
                    uncertainty                 = ' '
                    comment                     = ' '
                    axis                        = 'X'
                    reference                   = 'WGS84'
                    coordinate_reference_frame  = 'urn:ogc:crs:EPSG::4326'
    DEPH
           Size:       19x248
           Dimensions: DEPTH,TIME
           Datatype:   single
           Attributes:
                    long_name     = 'Depth'
                    standard_name = 'depth'
                    units         = 'm'
                    _FillValue    = -9999.9902
                    axis          = 'Z'
                    positive      = 'down'
    ATMS
           Size:       19x248
           Dimensions: DEPTH,TIME
           Datatype:   single
           Attributes:
                    long_name     = 'Atmospheric pressure at sea level'
                    standard_name = 'air_pressure_at_sea_level'
                    units         = 'hPa'
                    _FillValue    = -9999.9902
    DOX1
           Size:       19x248
           Dimensions: DEPTH,TIME
           Datatype:   single
           Attributes:
                    long_name     = 'Dissolved oxygen'
                    standard_name = 'volume_fraction_of_oxygen_in_sea_water'
                    units         = 'ml l-1'
                    _FillValue    = -9999.9902
    DRYT
           Size:       19x248
           Dimensions: DEPTH,TIME
           Datatype:   single
           Attributes:
```

```
                long _name        = 'Air temperature       in  dry  bulb'
                standard _name = 'air  _temperature'
                units             = 'degrees     _C'
                _FillValue      = -9999.9902
FLU2
        Size:          19x248
        Dimensions:  DEPTH,TIME
        Datatype:     single
        Attributes:
                long _name        = 'Chlorophyll-a  fluorescence'
                standard _name = ''
                units             = 'mg  m-3'
                _FillValue      = -9999.9902
GSPD
        Size:          19x248
        Dimensions:  DEPTH,TIME
        Datatype:     single
        Attributes:
                long _name        = 'Gust  wind  speed'
                standard _name = 'wind  _speed _of _gust'
                units             = 'm  s-1'
                _FillValue      = -9999.9902
HCDT
        Size:          19x248
        Dimensions:  DEPTH,TIME
        Datatype:     single
        Attributes:
                long _name        = 'Current to direction          relative true        north'
                standard _name = 'direction     _of _sea _water  _velocity'
                units             = 'degree'
                _FillValue      = -9999.9902
HCSP
        Size:          19x248
        Dimensions:  DEPTH,TIME
        Datatype:     single
        Attributes:
                long _name        = 'Horizontal  current  speed'
                standard _name = 'sea _water  _speed'
                units             = 'm  s-1'
                _FillValue      = -9999.9902
PHPH
        Size:          19x248
        Dimensions:  DEPTH,TIME
        Datatype:     single
        Attributes:
                long _name        = 'Ph'
                standard _name = 'sea _water  _ph_reported   _on_total   _scale'
                units             = '1'
                _FillValue      = -9999.9902
PRES
        Size:          19x248
        Dimensions:  DEPTH,TIME
        Datatype:     single
        Attributes:
                long _name        = 'Sea  pressure'
```

```
                    standard_name = 'sea_water_pressure'
                    units         = 'dbar'
                    _FillValue    = -9999.9902
                    axis          = 'Z'
                    positive      = 'down'
    PSAL
           Size:       19x248
           Dimensions: DEPTH,TIME
           Datatype:   single
           Attributes:
                    long_name     = 'Practical salinity'
                    standard_name = 'sea_water_practical_salinity'
                    units         = '0.001'
                    _FillValue    = -9999.9902
    TEMP
           Size:       19x248
           Dimensions: DEPTH,TIME
           Datatype:   single
           Attributes:
                    long_name     = 'Sea temperature'
                    standard_name = 'sea_water_temperature'
                    units         = 'degrees_C'
                    _FillValue    = -9999.9902
    TUR4
           Size:       19x248
           Dimensions: DEPTH,TIME
           Datatype:   single
           Attributes:
                    long_name     = 'Turbidity'
                    standard_name = 'sea_water_turbidity'
                    units         = '1'
                    _FillValue    = -9999.9902
    VHM0
           Size:       19x248
           Dimensions: DEPTH,TIME
           Datatype:   single
           Attributes:
                    long_name        = 'Spectral significant wave height (Hm0)'
                    standard_name    = 'sea_surface_wave_significant_height'
                    units            = 'm'
                    _FillValue       = -9999.9902
                    type_of_analysis = 'spectral analysis'
    VMDR
           Size:       19x248
           Dimensions: DEPTH,TIME
           Datatype:   single
           Attributes:
                    long_name        = 'Mean wave direction from (Mdir)'
                    standard_name    = 'sea_surface_wave_from_direction'
                    units            = 'degree'
                    _FillValue       = -9999.9902
                    type_of_analysis = 'spectral analysis'
    VTM02
           Size:       19x248
           Dimensions: DEPTH,TIME
```

```
        Datatype:   single
        Attributes:
                    long_name        = 'Spectral moments (0,2) wave period (Tm02)'
                    standard_name    = ''
                    units            = 's'
                    _FillValue       = -9999.9902
                    type_of_analysis = 'spectral analysis'
    VTPK
        Size:       19x248
        Dimensions: DEPTH,TIME
        Datatype:   single
        Attributes:
                    long_name        = 'Wave period at spectral peak'
                    standard_name    = 'wave_at_variance_spectral_density_maximum'
                    units            = 's'
                    _FillValue       = -9999.9902
                    type_of_analysis = 'spectral analysis'
    WDIR
        Size:       19x248
        Dimensions: DEPTH,TIME
        Datatype:   single
        Attributes:
                    long_name     = 'Wind from direction relative true north'
                    standard_name = 'wind_from_direction'
                    units         = 'degree'
                    _FillValue    = -9999.9902
    WSPD
        Size:       19x248
        Dimensions: DEPTH,TIME
        Datatype:   single
        Attributes:
                    long_name     = 'Horizontal wind speed'
                    standard_name = 'wind_speed'
                    units         = 'm s-1'
                    _FillValue    = -9999.9902
    GPS_LATITUDE
        Size:       19x248
        Dimensions: DEPTH,TIME
        Datatype:   single
        Attributes:
                    long_name     = 'GPS Latitude of each location'
                    standard_name = ' '
                    units         = 'degree_north'
                    _FillValue    = -9999.9902
    GPS_LONGITUDE
        Size:       19x248
        Dimensions: DEPTH,TIME
        Datatype:   single
        Attributes:
                    long_name     = 'GPS Longitude of each location'
                    standard_name = ' '
                    units         = 'degree_east'
                    _FillValue    = -9999.9902
```

Appendix B

Excerpt of Example Metadata.ttl file

In this appendix, here shows the simple excerpt of the Metadata.ttl file which describes some variables in the NetCDF file: "MO_201708_TS_MO_61277.nc" (Removed some duplicate parts): Metadata.ttl file: "example_datasets TTL.ttl"

```
<http://bigdataocean.eu/bdo/MO_201708_TS_MO_61277_TIME>
        a       <http://bigdataocean.eu/bdo/BDOVariable> ;
        <http://purl.org/dc/terms/identifier>
                "TIME" ;
        <http://www.w3.org/2002/07/owl#sameAs>
                <http://vocab.nerc.ac.uk/collection/P07/current/CFSN0115/> ;
        <http://www.w3.org/2004/02/skos/core#prefLabel>
                "time"@en .

<http://bigdataocean.eu/bdo/MO_201708_TS_MO_61277_LATITUDE>
        a       <http://bigdataocean.eu/bdo/BDOVariable> ;
        <http://purl.org/dc/terms/identifier>
                "LATITUDE" ;
        <http://www.w3.org/2002/07/owl#sameAs>
                <http://vocab.nerc.ac.uk/collection/P07/current/CFSN0600/> ;
        <http://www.w3.org/2004/02/skos/core#prefLabel>
                "latitude"@en .

<http://bigdataocean.eu/bdo/MO_201708_TS_MO_61277_LONGITUDE>
        a       <http://bigdataocean.eu/bdo/BDOVariable> ;
        <http://purl.org/dc/terms/identifier>
                "LONGITUDE" ;
        <http://www.w3.org/2002/07/owl#sameAs>
                <http://vocab.nerc.ac.uk/collection/P07/current/CFSN0554/> ;
        <http://www.w3.org/2004/02/skos/core#prefLabel>
                "longitude"@en .

<http://bigdataocean.eu/bdo/MO_201708_TS_MO_61277_DEPH>
        a       <http://bigdataocean.eu/bdo/BDOVariable> ;
```

```
        <http://purl.org/dc/terms/identifier>
                "DEPH" ;
        <http://www.w3.org/2002/07/owl#sameAs>
                <http://bigdataocean.eu/bdo/cf/parameter/manually_entered_depth> ;
        <http://www.w3.org/2004/02/skos/core#prefLabel>
                "manually_entered_depth"@en .

<http://bigdataocean.eu/bdo/MO_201708_TS_MO_61277_TEMP>
        a       <http://bigdataocean.eu/bdo/BDOVariable> ;
        <http://purl.org/dc/terms/identifier>
                "TEMP" ;
        <http://www.w3.org/2002/07/owl#sameAs>
                <http://vocab.nerc.ac.uk/collection/P07/current/CFSN0335/> ;
        <http://www.w3.org/2004/02/skos/core#prefLabel>
                "sea_water_temperature"@en .
```

Appendix C

Query Templates in Experiments

We choose 2 linear queries, 1 star query, 1 snowflake query and 1 complex query as the query templates for the experiment. In this appendix, here shows the details about the 5 query templates in the experiment.

- Linear Query: L3

```
SELECT ?v0 ?v1 WHERE {
        ?v0     wsdbm:likes     ?v1 .
        ?v0     wsdbm:subscribes        %v2% .
}
```

Listing C.1: SPARQL query

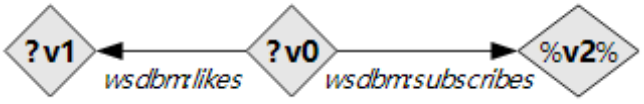

Figure C.1: Query Graph of Linear Query: L3

- Linear Query: L4

```
SELECT ?v0 ?v2 WHERE {
        ?v0     og:tag  %v1% .
        ?v0     sorg:caption    ?v2 .
}
```

Listing C.2: SPARQL query

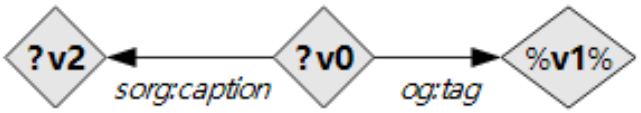

Figure C.2: Query Graph of Linear Query: L4

- Star Query: S6

```
SELECT ?v0 ?v1 ?v2 WHERE {
        ?v0     mo:conductor    ?v1 .
        ?v0     rdf:type        ?v2 .
        ?v0     wsdbm:hasGenre  %v3% .
}
```

Listing C.3: SPARQL query

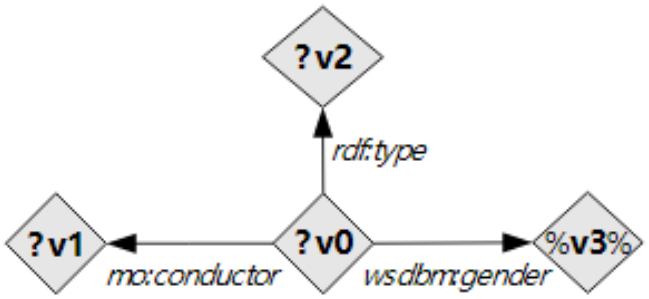

Figure C.3: Query Graph of Star Query: S6

- Snowflake query: F5

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
        ?v0     gr:includes     ?v1 .
        %v2%    gr:offers       ?v0 .
        ?v0     gr:price        ?v3 .
        ?v0     gr:validThrough ?v4 .
        ?v1     og:title        ?v5 .
        ?v1     rdf:type        ?v6 .
}
```

Listing C.4: SPARQL query

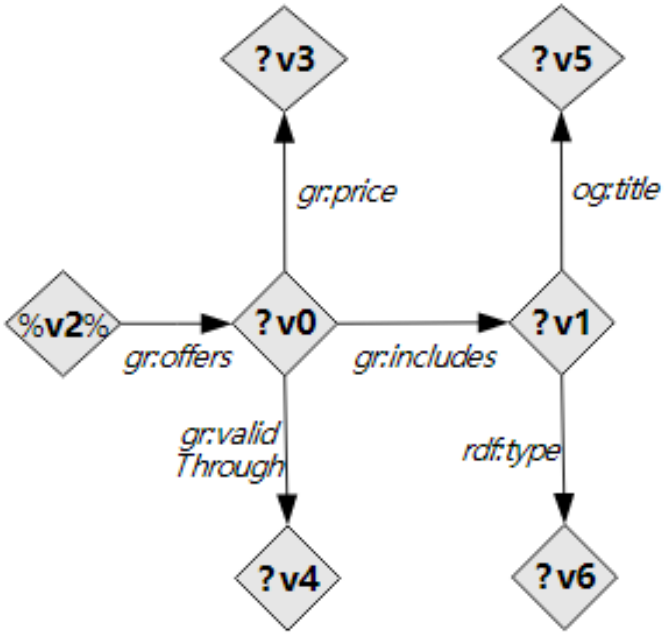

Figure C.4: Query Graph of Snowflake Query: F5

- Complex query: C3

```
:
SELECT ?v0 WHERE {
        ?v0     wsdbm:likes     ?v1 .
        ?v0     wsdbm:friendOf  ?v2 .
        ?v0     dc:Location     ?v3 .
        ?v0     foaf:age        ?v4 .
        ?v0     wsdbm:gender    ?v5 .
        ?v0     foaf:givenName  ?v6 .
}
```

Listing C.5: SPARQL query

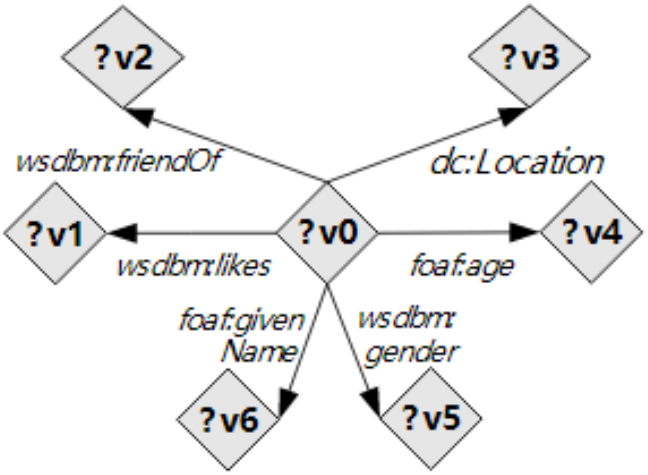

Figure C.5: Query Graph of Complex Query: C3

List of Figures

List of Tables

Bibliography

[1] Jeremy Ginsberg, Matthew H Mohebbi, Rajan S Patel, Lynnette Brammer, Mark S Smolinski, and Larry Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, 457(7232):1012, 2009.

[2] Craig A Statchuk. Building a data query engine that leverages expert data preparation operations, June 7 2018. US Patent App. 15/370,839.

[3] Peter A Cooper and Blimat Ahmed Omar. Knowledge-based fast web query engine using nosql. In *Digital Forensic and Security (ISDFS), 2018 6th International Symposium on*, pages 1–5. IEEE, 2018.

[4] Jim Gray, David T Liu, Maria Nieto-Santisteban, Alex Szalay, David J DeWitt, and Gerd Heber. Scientific data management in the coming decade. *Acm Sigmod Record*, 34(4):34–41, 2005.

[5] Ben Domenico, Stefano Nativi, John Caron, Lorenzo Bigagli, and Ethan Davis. A standards-based, web services gateway to netcdf datasets. In *Proc. of AMS–22nd IIPS Conference, Atlanta, Georgia, abstr*, number 8.1, 2006.

[6] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[7] Frank Manola, Eric Miller, Brian McBride, et al. RDF primer. *W3C recommendation*, 10(1-107):6, 2004.

[8] Wikipedia contributors. Sparql — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=SPARQL&oldid=874272095, 2018. [Online; accessed 27-February-2019].

[9] Eric Prud Hommeaux, Andy Seaborne, et al. SPARQL query language for RDF. *W3C recommendation*, 15, 2008.

[10] W3C SPARQL Working Group et al. SPARQL 1.1 overview. https://www.w3.org/TR/sparql11-overview/, 2013.

[11] David Beazley. Ply (python lex-yacc). *See http://www. dabeaz. com/ply*, 2001.

[12] Jeremy G Siek. Assignment 2: Parsing with ply (python lex-yacc).

[13] Frank DeRemer and Thomas Pennello. E□cient computation of lalr (1) look-ahead sets. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4): 615–649, 1982.

[14] Michael E Lesk and Eric Schmidt. Lex: A lexical analyzer generator, 1975.

[15] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[16] H He and AK Singh. Graphql: Query language and access methods for graph databases. Technical report, Technical report, Department of Computer Science at University of California, Santa Barbara, 2007.

[17] ABM Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

[18] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[19] Wikipedia contributors. Nosql — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=NoSQL&oldid=884298289, 2019. [Online; accessed 27-February-2019].

[20] Wikipedia. Category:scientific databases. https://en.wikipedia.org/wiki/Category:Scientific_databases, 2018.

[21] Lars George. *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[22] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

[23] Kristina Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O'Reilly Media, Inc.", 2013.

[24] Zachary Parker, Scott Poe, and Susan V Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, page 5. ACM, 2013.

[25] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. Mongodb vs oracle–database comparison. In *2012 third international conference on emerging intelligent data and web technologies*, pages 330–335. IEEE, 2012.

[26] Richard Kemp. Legal aspects of managing big data. *Computer Law & Security Review*, 30(5):482–491, 2014.

[27] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. Anapsid: an adaptive query processing engine for sparql endpoints. In *International Semantic Web Conference*, pages 18–34. Springer, 2011.

[28] Wikipedia contributors. Binary tree — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Binary_tree&oldid=879968802, 2019. [Online; accessed 27-February-2019].

[29] Wikipedia contributors. Metadata — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Metadata&oldid=884396848, 2019. [Online; accessed 27-February-2019].

[30] Rishichandra Wawhal. The ultimate guide to schema stitching in graphql. https://blog.hasura.io/the-ultimate-guide-to-schema-stitching-in-graphql-f30178ac0072, 2018.