



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

ASSIGNMENT OF MASTER'S THESIS

Title: Dataflow analysis of Google BigQuery scripts
Student: Bc. Kyrlo Bulat
Supervisor: Ing. Jan Trávníček, Ph.D.
Study Programme: Informatics
Study Branch: Web and Software Engineering
Department: Department of Software Engineering
Validity: Until the end of winter semester 2021/22

Instructions

Study a Google BigQuery technology. Study the syntax and semantics of Google BigQuery query language.

Get familiar with the Manta project and its approach to dataflow representation.

Analyze whether a dataflow analysis can be obtained from a static analysis of scripts in a Google BigQuery language.

Design an approach to extract metadata needed for dataflow analysis of the Google BigQuery scripts.

Design a way to analyze and represent the source codes in the Google BigQuery language, document the dataflow relevant declarations and statements.

Design analysis of source codes in the Google BigQuery language to detect dataflow between its data structures.

Implement a proof of concept tool to extract dataflow from a set of Google BigQuery scripts to the Manta system.

Design and perform appropriate testing of your proof of concept.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague May 28, 2020



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Master's thesis

Dataflow analysis of Google BigQuery scripts

Bc. Kyrylo Bulat

Department of Software Engineering
Supervisor: Ing. Jan Trávníček, Ph.D

January 5, 2021

Acknowledgements

I would like to thank the supervisor of this master thesis, Ing. Jan Trávníček, Ph.D., for all genuinely useful advice and help. My appreciation goes to all the colleagues of Manta team, especially to Mgr. Jiří Toušek for his patience and dedication to every question I had. All of this would be impossible without the support of my family and friends.

Declaration

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the “Metodický pokyn o etické přípravě vysokoškolských závěrečných prací”.

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the “Copyright Act”), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs (“software”), in accordance with Article 2373 of the Act No. 89/2012 Coll., the Civil Code, I hereby grant a nonexclusive and irrevocable authorisation (license) to use this software, to any and all persons that wish to use the software. Such persons are entitled to use the software in any way without any limitations (including use for-profit purposes). This license is not limited in terms of time, location and quantity, is granted free of charge, and also covers the right to alter or modify the software, combine it with another work, and/or include the software in a collective work.

In Prague on January 5, 2021

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2021 Kyrylo Bulat. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Bulat, Kyrylo. *Dataflow analysis of Google BigQuery scripts*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce je zaměřena na analýzu datových toků v Google BigQuery skriptech a jejich reprezentaci. Nejprve popisuje přístupy, které se používají pro data lineage, analýzu zdrojového kódu a vizualizaci toku dat v systému Manta. Poté zkoumá technologii Google BigQuery, její databázové objekty a syntaxi jejího SQL dialektu. Pokračuje popisem architektury a návrhu implementovaného prototypu. Poslední kapitoly této práce jsou věnovány testování a prezentaci výstupů vytvořeného prototypového řešení.

Klíčová slova Google BigQuery, data lineage, Manta, analýza zdrojového kódu, datové toky

Abstract

This thesis is focused on the analysis of data flows for Google BigQuery scripts and their representation. Firstly, it describes the possible approaches to data lineage, source code analysis, and data flow visualization in the Manta system. It then examines the Google BigQuery technology, its database objects and SQL dialect syntax. It continues with design and architecture, which are used during the implementation of the prototype. The last chapters of this work are dedicated to testing and presenting the outputs of the created solution.

Keywords Google BigQuery, data lineage, Manta, source code analysis, data flows

Contents

Introduction	1
Goals	2
1 Basic concepts	3
1.1 Data lineage	3
1.2 Static code analysis	3
1.2.1 Lexical analysis	4
1.2.2 Syntax analysis	5
1.2.3 Semantic analysis	6
1.3 Data dictionary	7
1.3.1 Metadata extraction	7
1.3.2 Database specific metadata	7
1.4 Dataflow analysis	7
1.5 Manta Flow	8
2 Analysis	9
2.1 Google BigQuery	9
2.1.1 Google Cloud Platform	10
2.1.2 High-level architecture of BigQuery	11
2.1.3 Database objects' structure	12
2.2 Metadata extraction	13
2.2.1 Ways to extract	13
2.2.1.1 INFORMATION_SCHEMA views	14
2.2.1.2 SDK	14
2.2.1.3 REST API	14
2.2.2 Required privileges	15
2.2.3 Metadata to extract	15
2.2.3.1 Project	16
2.2.3.2 Dataset	16

2.2.3.3	Table	16
2.2.3.4	View	17
2.2.3.5	Function	18
2.2.3.6	Procedure	18
2.3	Analysis of BigQuery SQL	19
2.3.1	Standard and legacy SQL	19
2.3.2	Data types	20
2.3.3	Lexical Structure	21
2.3.3.1	Identifiers	21
2.3.3.2	Literals	22
2.3.3.3	Case sensitivity	23
2.3.3.4	Keywords	23
2.3.4	Query syntax	23
2.3.4.1	EXCEPT clause	24
2.3.4.2	FOR SYSTEM_TIME AS OF clause	25
2.3.4.3	REPLACE clause	26
2.3.4.4	OMIT RECORD IF clause	26
2.3.5	Data manipulation language	27
2.3.5.1	INSERT statement	27
2.3.5.2	DELETE statement	27
2.3.5.3	TRUNCATE TABLE statement	28
2.3.5.4	UPDATE statement	28
2.3.5.5	MERGE statement	28
2.4	Requirements analysis	29
2.4.1	Functional requirements	29
2.4.2	Non-functional requirements	30
3	Design	33
3.1	Architecture	33
3.1.1	Extractor	33
3.1.2	Parsing and resolving	35
3.1.3	Dataflow	36
3.1.4	Other modules	37
3.2	Tools	37
3.2.1	Java	37
3.2.2	Spring	37
3.2.3	ANTLR	38
3.2.4	Testing tools	38
3.2.5	Maven	38
4	Implementation	41
4.1	Extractor	41
4.1.1	CredentialService	41
4.1.2	Retrieving and converting metadata	42

4.1.3	DictionaryWriter	44
4.1.4	DdlScriptGenerator	44
4.1.5	DdlWriter	44
4.1.6	Data dictionary persisting	44
4.1.7	BigQueryExtractor	45
4.2	Parsing and resolving	45
4.2.1	Parsing	45
4.2.1.1	BigQueryLexer	45
4.2.1.2	BigQuery parser grammar	46
4.2.2	Resolving	47
4.2.2.1	AST nodes	48
4.2.2.2	Context concept	48
4.3	Dataflow	49
4.3.1	Dataflow graph	49
4.3.2	FlowVisitor	49
5	Testing	51
5.1	Extractor	51
5.1.1	Unit and integration tests	51
5.1.2	Memory management and performance tests	52
5.2	Parsing and resolving	52
5.2.1	Parsing	52
5.2.2	Resolving	53
5.3	Dataflow	54
6	Result examples	55
6.1	CREATE TABLE statement	55
	Conclusion	57
	Bibliography	59
	A Acronyms	63
	B Contents of enclosed USB	65

List of Figures

1.1	AST for SELECT statement	5
1.2	Demo visualization results generated by Manta Flow	8
2.1	Google BigQuery web UI console with an executed query for the number of COVID-19 cases in the United States based on the public dataset data	10
2.2	A high-level architecture of BigQuery [5]	12
2.3	Resource hierarchy in BigQuery	13
3.1	BigQuery extractor module class diagram	34
3.2	BigQuery parsing and resolving module class diagram	35
3.3	BigQuery dataflow module class diagram	36
5.1	AST for INSERT statement	53
6.1	Dataflow graph for CREATE TABLE AS statement	56

List of Tables

2.1	Data types in BigQuery legacy and standard SQL	20
2.2	Data type properties in BigQuery standard SQL	21
2.3	Case sensitivity in BigQuery	23
2.4	SELECT clauses in BigQuery	24

Introduction

Nowadays, everything is about the data and the ways it is being processed. Organizations of different sizes, all the electronic devices and we ourselves produce, receive and process data daily. The amount of it has significantly increased over the past few years and it continues to grow.

Such a large amount of data brings up questions that need to be answered: how to process, store, analyze, and, more importantly, understand it? The answer to these questions becomes critical in the enterprise domain, where proper data management is the key to success.

Modern companies can have hundreds of different systems running, where each is working with data in its own way using different tools. In such an interlinked environment tracing the origin of data and its destination with all the transformations along the way becomes very complicated. This is where having a proper data lineage solution, such as Manta, becomes useful.

Manta focuses on analysis and visualization of how data flows inside the enterprise. This type of analysis provides information about data and helps the organization handle situations such as migration of the projects, data consolidation and virtualization, impact analysis, and others.[1]

With modern trends of using cloud technologies in software development and data analysis, Manta must stay up to date and cover this area too. One of such tools that is gaining popularity is Google BigQuery.

Google BigQuery is a serverless, highly scalable data warehouse that comes with a built-in query engine. It has a set of features that allow quick and easy integration to other services and at the same time provide performance on the same level as complex data warehouse solutions or even better.

This thesis is focused on implementing the module for dataflow analysis of Google BigQuery scripts, which will be used as a part of the Manta system.

Goals

This thesis aims to implement a proof of concept tool to extract dataflow from a set of Google BigQuery scripts to the Manta system. In order to achieve this goal, the work is divided into the following steps:

- analysis of Google BigQuery technology and the way dataflow analysis can be performed on its scripts,
- design and implementation of the module for extraction of metadata needed for dataflow analysis,
- implementation of the module responsible for executing static code analysis on a set of Google BigQuery scripts,
- representation of the dataflow according to the standards of the Manta system.

In the end, the appropriate testing needs to be designed and executed for the implemented proof of concept.

Basic concepts

This chapter describes the basic terms and concepts in data lineage and static code analysis of SQL scripts domains. It also covers an introduction to the Manta system and its approach to dataflow analysis.

1.1 Data lineage

Data lineage shows the origin of the data, where it moves or flows to in the environment, and what transformations are applied to it along the way.[1] With the amount of data and complex architectures of modern enterprise systems, data lineage utilization is an important part of analytics processes.

There are several approaches to data lineage, including manual lineage, lineage by tagging, parsing lineage, etc.[1] Parsing lineage belongs to the group of approaches that can be automated. It involves a programmatical analysis of the program and representing it in an understandable form. There are three main ways of performing program analysis automatically:

- static program analysis – without executing the target program,
- dynamic programming analysis – during runtime of the program,
- combination of both of them.

Static program analysis is often performed on source code of the program. This work executes static code analysis of SQL scripts before dataflow analysis phase.

1.2 Static code analysis

Static code analysis is used in various tools, including integrated development environments, compilers, or security applications. The compiler case is very

similar to the approach used in this thesis and it consists of the following phases:

- lexical analysis,
- syntax analysis,
- semantic analysis.

The following subsections discuss each of them.

1.2.1 Lexical analysis

The first phase of static code analysis is lexical analysis, or also known as scanning. Lexical analyzer reads the stream of characters making up the source program and groups them into meaningful sequences called lexemes. For each lexeme, the analyzer produces a token on its output, which is passed to later phases of code analysis.

The tokens are recognized based on a set of patterns, which are called regular expressions. Regular expression is a notation used to describe regular language — a language that is accepted by finite automaton. “*A finite automaton is a formalism for recognizers that has a finite set of states, an alphabet, a transition function, a start state, and one or more accepting states.*”[2] The concept of finite automaton lies in the core of the lexical analysis.

The patterns used to recognize tokens may specify additional logic of creating tokens, such as skipping whitespace or comments, that are irrelevant to further processing phases. Examples of regular expressions can be seen in the following code 1.1.

```
1  SELECT = "SELECT";
2  FROM = "FROM";
3  COMMA = ",";
4  SEMICOLON = ";";
5  SPACE = " ";
6  WHITESPACE = SPACE , { SPACE } ;
7
8  DIGIT = "0".."9";
9  LETTER : "a".."z" | "A".."Z";
10
11 UNSIGNED_INT = DIGIT , { DIGIT };
12 ID = LETTER , { LETTER | DIGIT };
```

Listing 1.1: Token patterns specification in EBNF

This set of patterns would be sufficient to execute lexical analysis on the following statement 1.2. The patterns are simplified for the purpose of example and do not cover the SQL `SELECT` statement’s full syntax.

```
1  SELECT 1, b FROM table1;
```

Listing 1.2: `SELECT` statement for lexical analysis

Such lexical analysis would produce the list of tokens 1.3, where each can be represented as a pair consisting of the name and value itself.

```
1 <SELECT, 'SELECT'> <UNSIGNED_INT, '1'> <COMMA, ','> <ID, 'b'> <
  FROM, 'FROM'> <ID, 'table1'> <SEMICOLON, ';'>;
```

Listing 1.3: Result of lexical analysis

1.2.2 Syntax analysis

The token stream generated in the previous step serves as an input to the syntax analysis phase, also known as parsing. “*The parser derives a syntactic structure for the program, fitting the words into a grammatical model of the source programming language. If the parser determines that the input stream is a valid program, it builds a concrete model of the program for use by the later phases of compilation.*”[2]

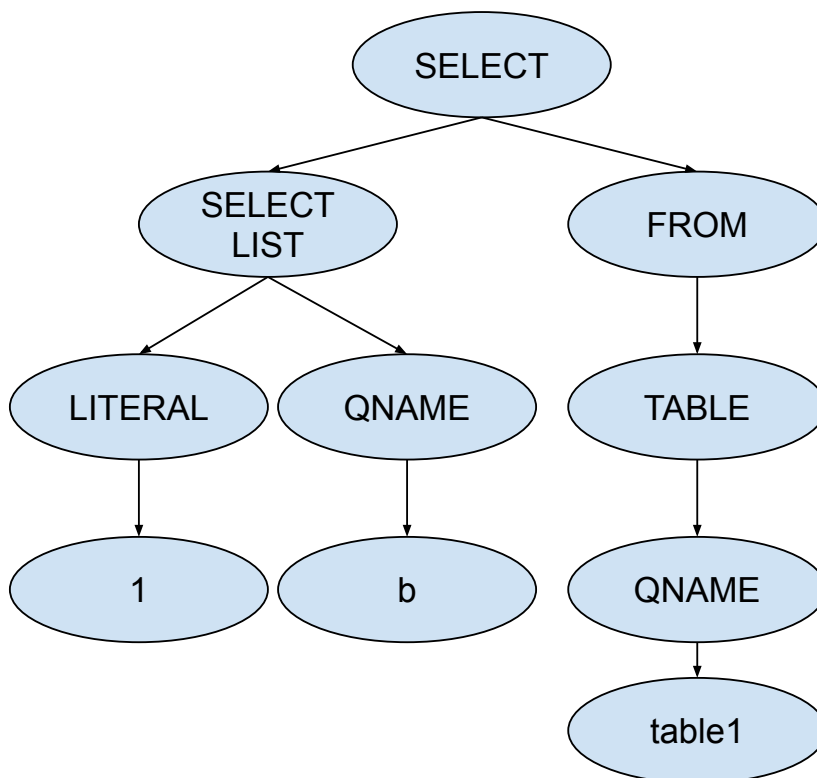


Figure 1.1: AST for SELECT statement

Abstract syntax tree, also known as AST, is often used as a “concrete model of the program” on the output of the parsing. It is a tree-like structure, that provides a high-level view of the structure of the input program. Figure 1.1 shows an example of an abstract syntax tree constructed for the statement 1.2.

At first, parse tree is constructed according to the rules defined in context-free grammar, which consists of terminal and nonterminal symbols, set of productions, also known as production rules, and start symbol. A grammar is used to specify the syntax of the language, so one of its applications is to describe the hierarchical structure of most programming languages. Parse tree contains all the tokens provided to the parser on the input and also nodes representing the used rules of the grammar. It serves as a conceptual basis for building abstract syntax tree, which may skip some tokens and have different structure. Usually, tokens that do not play any difference in further processing of AST are skipped, such as semicolons between statements in SQL.

1.2.3 Semantic analysis

The next phase of static code analysis is semantic analysis, which ensures that statements of the program are semantically correct according to the language definition. Among its tasks belong type checking, resolution of identifiers, and others.

In this thesis, this phase’s core goal is to find what all references in the source script refer to. For this purpose, a semantic analysis tool uses the abstract syntax tree constructed in the previous step and information in the symbol table — a data structure used to hold information about source-program constructs.[3] For example, the semantic analysis of statement 1.2 would include understanding what is referenced by identifiers “table1” and “a” and checking that column “a” is indeed present in “table1”.

This thesis also includes deduction analysis, which is not a standard part of static code analysis. It is a heuristic used in the Manta system, whenever semantic analysis tool fails to resolve all the references. It is a process of “guessing” what is referred by the references based on the knowledge of the language structure. It is often used in cases where the symbol table does not contain enough information. For example, in the case of statement 1.2 and assuming not having information about “table1” in the symbol table, one of the guesses could be that “table1” is either table or view and “a” is a column of this table or view.

The semantic analysis phase’s output is again AST but with checked semantic and nodes referencing the entities in the symbol table. As a result, the symbol table structure is also enriched with data found or deduced during this phase.

1.3 Data dictionary

The symbol table mentioned in 1.2.3 plays a significant role during the semantic phase of static code analysis. It allows to find objects and entities that are referenced in the script, as well as to register a new one. That is why it is essential to collect as much information as possible about analyzed source codes and entities before starting a static code analysis.

1.3.1 Metadata extraction

SQL is a programming language mostly used to operate and manage data stored in relational or other types of database systems. Data in database systems is usually contained in structures and objects, such as tables, views, etc. Manta system benefits from the nature of the database environment and executes extraction of metadata from the target database instance as preparation before static code analysis.

The extraction phase aims to collect information about target database objects and save it to a data dictionary, which is equivalent to a symbol table in Manta terminology. Thanks to the collected data, the semantic phase of static code analysis can search for referenced entities in the data dictionary and update it with new information found in the source code. This process helps to increase the accuracy of resolving and in the result data lineage itself.

1.3.2 Database specific metadata

Except for extracted metadata, the data dictionary can also be updated with database-specific information, such as object hierarchy, built-in functions, procedures, and variables. That heuristic is also part of the Manta approach to the data lineage process.

1.4 Dataflow analysis

Based on the output of static code analysis, it is possible to construct a dataflow graph, which is essentially one of the data lineage representations of the input program. It is an oriented graph, where nodes represent data, data transformations, or a more abstract containers for the data, and edges between them represent how data flows.

In this thesis, the graph's edges are labeled, which means they are divided into two groups: direct and filter.

Direct edges represent the direct dependency of data from one node to another. For example, in statement 1.2, the direct edge will be created from column "a" in the table "table1" to the column "a" in the result.

1. BASIC CONCEPTS

The filter edge is used to express filtering conditions, which may affect the data contained in target node. An example could be the LIMIT clause of the SELECT statement, which affects the amount of data in the result.

1.5 Manta Flow

Manta Flow is one of the applications developed in the Manta company, and it focuses on the visualization of data lineage based on the analysis of source code in a business intelligence environment. Manta technology supports and integrates with various technologies in database, reporting and analysis, modeling, data integration, and programming languages domains.[4] The example of demo dataflow visualization created by Manta Flow can be seen in figure 1.2.

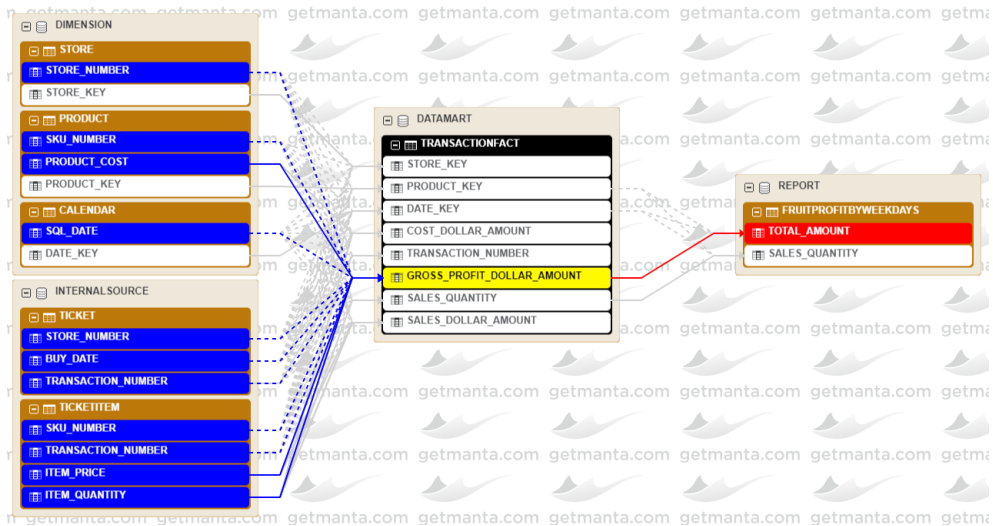


Figure 1.2: Demo visualization results generated by Manta Flow

The steps described in this chapter are part of how Manta Flow approaches the source code analysis and generation of dataflow graphs.

Analysis

This chapter focuses on describing what Google BigQuery and Google Cloud Platform is. It also covers the BigQuery object hierarchy and metadata available for extraction. It provides a high-level understanding of syntax and possibilities of BigQuery standard and legacy SQL dialects.

2.1 Google BigQuery

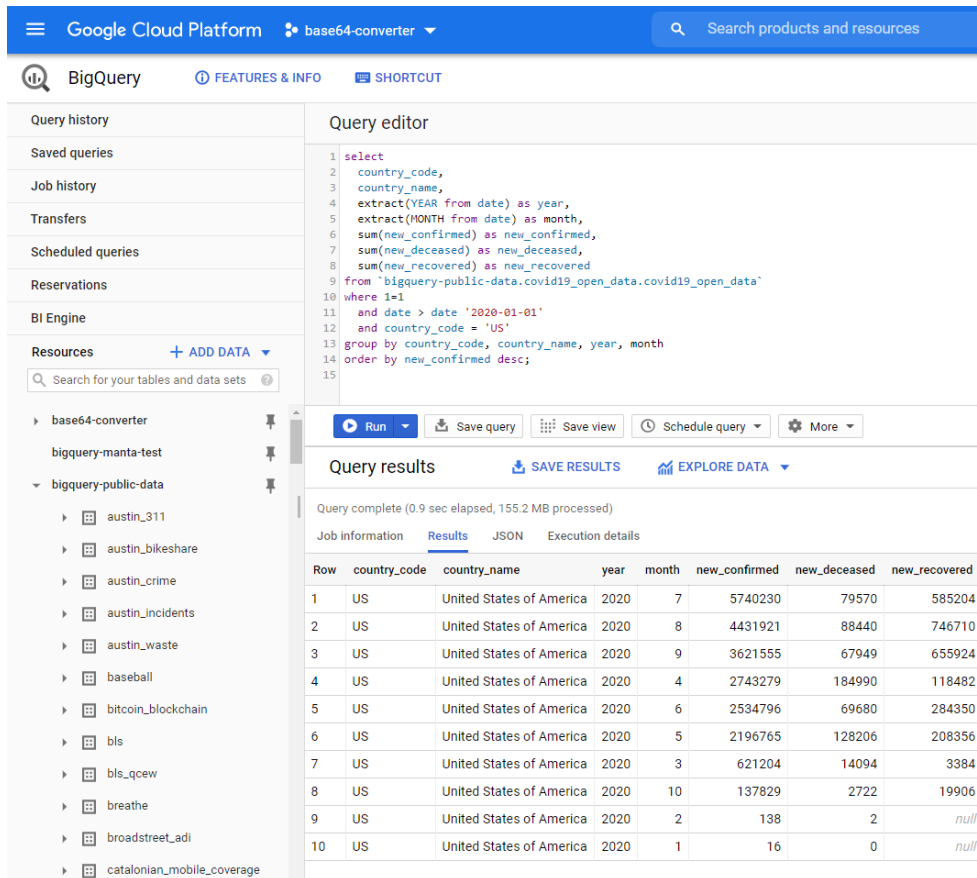
Google BigQuery is a serverless, highly scalable data warehouse that comes with a built-in query engine. It was first launched as a service in 2010 with general availability in November 2011.[5] The query engine can run SQL queries on terabytes of data in a matter of seconds and petabytes in only minutes. Such performance comes without the need to manage any infrastructure and without the need to create or rebuild indexes.[6] Key features of this technology are the following: [7]

- Serverless – with serverless data warehousing, Google does all resource provisioning behind the scenes, so end users can focus on data and analysis rather than worrying about upgrading, securing, or managing the infrastructure.
- Petabyte scale – BigQuery gives great performance on users' data, with the ability to scale seamlessly to store and analyze petabytes to exabytes of data with ease.
- Standard SQL – BigQuery supports a standard SQL dialect that is ANSI:2011 compliant. BigQuery also provides ODBC and JDBC drivers at no cost to ensure easy integration of existing applications with its powerful engine.

Google BigQuery provides multiple ways to access its resources. To start experimenting, it is enough to login to Google Cloud Platform, choose BigQuery service, and run the query in web UI console. Figure 2.1 shows part

2. ANALYSIS

of Google BigQuery console with entered SQL query and the result of the execution.



The screenshot displays the Google Cloud Platform BigQuery interface. The top navigation bar includes the Google Cloud Platform logo, the user's account name 'base64-converter', and a search bar. Below the navigation bar, there are links for 'BigQuery', 'FEATURES & INFO', and 'SHORTCUT'. The main interface is divided into two main sections: 'Query editor' and 'Query results'.

The 'Query editor' section shows the following SQL query:

```
1 select
2   country_code,
3   country_name,
4   extract(YEAR from date) as year,
5   extract(MONTH from date) as month,
6   sum(new_confirmed) as new_confirmed,
7   sum(new_deceased) as new_deceased,
8   sum(new_recovered) as new_recovered
9 from `bigquery-public-data.covid19_open_data.covid19_open_data`
10 where 1=1
11   and date > date '2020-01-01'
12   and country_code = 'US'
13 group by country_code, country_name, year, month
14 order by new_confirmed desc;
15
```

The 'Query results' section shows the following table:

Row	country_code	country_name	year	month	new_confirmed	new_deceased	new_recovered
1	US	United States of America	2020	7	5740230	79570	585204
2	US	United States of America	2020	8	4431921	88440	746710
3	US	United States of America	2020	9	3621555	67949	655924
4	US	United States of America	2020	4	2743279	184990	118482
5	US	United States of America	2020	6	2534796	69680	284350
6	US	United States of America	2020	5	2196765	128206	208356
7	US	United States of America	2020	3	621204	14094	3384
8	US	United States of America	2020	10	137829	2722	19906
9	US	United States of America	2020	2	138	2	null
10	US	United States of America	2020	1	16	0	null

Figure 2.1: Google BigQuery web UI console with an executed query for the number of COVID-19 cases in the United States based on the public dataset data

2.1.1 Google Cloud Platform

Google BigQuery is available for users as a part of Google Cloud Platform (GCP). GCP is a collection of products and services available in the form of web services, which allows users to use some of Google's infrastructure. This collection includes many things that are common across all cloud providers, such as on-demand virtual machines via Google Compute Engine or object storage for storing files via Google Cloud Storage.[8] When comparing Google BigQuery to other database and data warehouse solutions, it is important to consider easy integrations to other Google Cloud Platform products. The list of BigQuery features include:

- federated queries that allow users to run queries against data held in Google Cloud Storage, Cloud SQL (a relational database), Bigtable (a NoSQL database), Spanner (a distributed database), or Google Drive (which offers spreadsheets).
- The combination of Cloud Pub/Sub, Cloud Dataflow and BigQuery allows creating ETL tool with customized steps and transformations based on the user's needs.
- With Dataproc and Dataflow, BigQuery provides integration with the Apache big data ecosystem, allowing existing Hadoop/Spark and Beam workloads to read or write data directly from/to BigQuery using the Storage API.[7]

2.1.2 High-level architecture of BigQuery

BigQuery service is built on top of Dremel technology, which has been in production internally in Google since 2006.[9] Dremel is a scalable, interactive ad-hoc query system for the analysis of read-only nested data, capable of running aggregation queries over trillion-row tables in seconds. Except for Dremel, BigQuery also uses other Google's technologies:

- Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across several clusters, each with up to tens of thousands of machines.[10]
- Colossus is a distributed file system that is the successor to the Google File System.[11]
- Jupiter is Google's network that can deliver 1 Petabit/sec of total bi-section bandwidth, allowing efficient and quick distribution of large workloads.[12]

Figure 2.2 shows the high-level architecture of BigQuery with the technologies mentioned above. In this figure, the compute (Borg) and storage (Colossus) parts are separated, allowing BigQuery to scale both components independently based on the current users' demand.

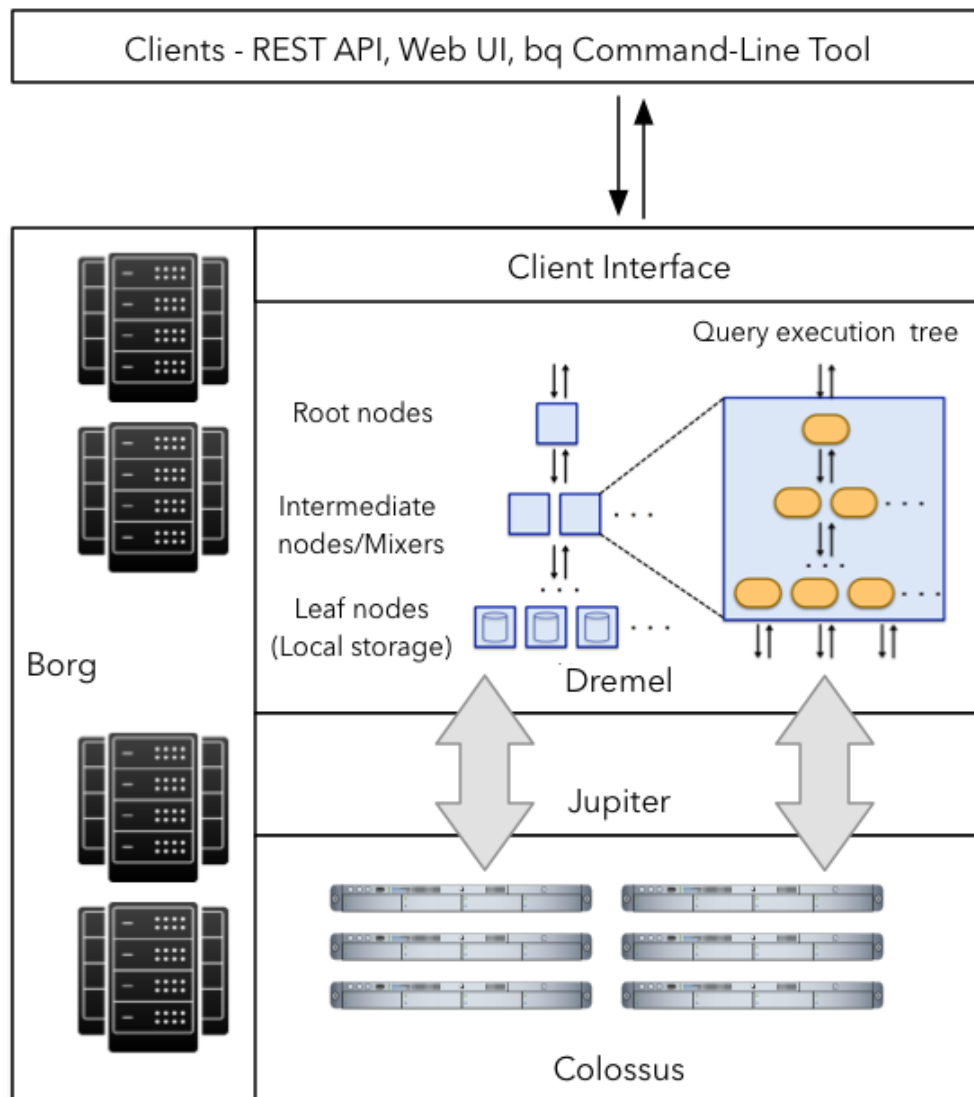


Figure 2.2: A high-level architecture of BigQuery [5]

2.1.3 Database objects' structure

In comparison to other known databases, BigQuery has a slightly different structure and terminology for its resources. Additionally, all the database resources are grouped into Organizations, the root node of the Google Cloud resource hierarchy. This entity provides control over all resources that belong to an organization or company. The following figure 2.3 shows a tree representation of resource hierarchy in BigQuery.

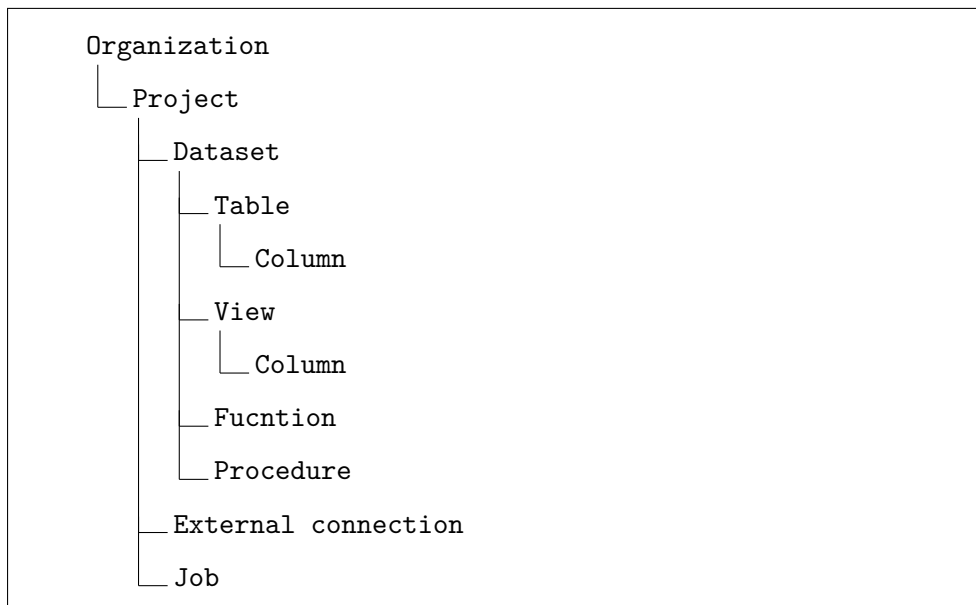


Figure 2.3: Resource hierarchy in BigQuery

In the hierarchy a project comes after the Organization resource and it serves as the container for the datasets, external connections, and jobs. A single project can be compared to a database instance in other database technologies. Dataset is the parent of the tables, user-defined functions, procedures, and views. Dataset can be considered as a schema in a database like PostgreSQL. The lowest level of objects hierarchy take tables, views, functions and procedures, which are similar to objects in classical databases. An external connection is an entity that is used during the federated queries in BigQuery. Federated queries allow users to execute SQL queries against tables stored in Cloud SQL without copying or moving data.[13] Its functionality is similar to “dblink” feature in PostgreSQL or Oracle databases. A job resource is automatically created, scheduled, and run whenever the user loads, exports, queries, or copies data in the BigQuery platform.[14]

2.2 Metadata extraction

This section describes ways to extract information about stored objects in BigQuery and required privileges for it. It also focuses on a detailed description of object types, its attributes, and available metadata.

2.2.1 Ways to extract

There are three main approaches to interaction with BigQuery and extraction of metadata:

- INFORMATION_SCHEMA views,
- SDK,
- REST API.

2.2.1.1 INFORMATION_SCHEMA views

INFORMATION_SCHEMA is a series of views that provide access to: [15]

- dataset metadata,
- routine metadata,
- table metadata,
- view metadata.

This way of extraction allows accessing metadata in a programmatical way with the help of SQL queries. The problem is that list above does not contain an ability to extract metadata about projects which are required for a successful and complete process of extraction. The INFORMATION_SCHEMA feature was in the beta release phase when working on this thesis and might be changed in future releases.

2.2.1.2 SDK

Google Cloud provides development tools and libraries in different programming languages for interacting with its services and products. BigQuery also has its project on GitHub. The SDK allows access to BigQuery service on the same level as REST API with some known limitations.

This approach's advantages are the ready-for-use objects class model, team of developers that work on improvement of development tools, and online community of people using the same library and sharing their experience. The disadvantage of this approach is that BigQuery SDK does not provide the functionality of accessing metadata about projects.

2.2.1.3 REST API

BigQuery service exposes access to its functionalities with the help of an application programming interface (API). Within API, users can find resources for interacting with core resources such as datasets, tables, jobs, and routines.[16] Although Google documentation recommends using this service through the libraries described in 2.2.1.2, it does not give such flexibility as in the case of direct API usage.

The API is accessible with HTTPS and follows the REST principles. This approach gives at least the same amount of functionalities as in previously

described ways to extract metadata and the ability to list projects and its metadata.

Considering all the previously mentioned pros and cons of different approaches, the BigQuery API approach was chosen for implementation.

2.2.2 Required privileges

To be able to extract metadata about objects, the prototype application has to obtain access to them. Ideally, the implemented solution has to have enough privileges to retrieve metadata about objects but not to be able to retrieve objects' data itself, i.e., not be able to access records in tables.

Thanks to Cloud Identity and Access Management tool, it is possible to define and manage fine-grained access to Google Cloud Platform resources. This service's core concept is the definition of who (identity) has what access (role) for which resources.[17]

The role is a collection of permissions. Permissions determine what operations are allowed on a resource. There are a set of predefined roles that are available for users of Google Cloud Platform, and one of them is BigQuery Metadata Viewer that contains the following permissions:

- `bigquery.datasets.get`
- `bigquery.datasets.getIamPolicy`
- `bigquery.models.getMetadata`
- `bigquery.models.list`
- `bigquery.routines.get`
- `bigquery.routines.list`
- `bigquery.tables.get`
- `bigquery.tables.getIamPolicy`
- `bigquery.tables.list`
- `resourcemanager.projects.get`
- `resourcemanager.projects.list`

Giving this role to proof of concept application ensures that only metadata about database objects will be extracted and data itself will stay inaccessible.

2.2.3 Metadata to extract

The following subsection focuses on a detailed description of object types, its attributes, and metadata available in the BigQuery platform. It also covers the importance of extracted metadata for later use in building dataflows.

2.2.3.1 Project

The project resource is the base-level organizing entity. It is not used solely for purposes of BigQuery, but for Google Cloud platform in general. Creation of a project is required to use Google Cloud services, managing APIs, billing, and permissions.[18] A project has the following attributes that we need to extract:

- id – unique identifier for the project,
- name – a human-readable name for the project.

A detailed list of projects is retrieved with the help of an HTTP GET request from the following resource <https://bigquery.googleapis.com/bigquery/v2/projects>.

2.2.3.2 Dataset

Datasets are top-level containers used to organize and control access to tables, views, functions, and procedures. Each dataset is contained within a specific project. It has the following attributes that need to be extracted:

- id – an identifier for the dataset that is unique per project,
- description (optional) – a description for the dataset,
- friendly name (optional) – a descriptive name for the dataset.

List of datasets and detail about every dataset is retrieved from the following resources:

- <https://bigquery.googleapis.com/bigquery/v2/projects/{projectId}/datasets>
- <https://bigquery.googleapis.com/bigquery/v2/projects/{projectId}/datasets/{datasetId}>

2.2.3.3 Table

A table contains individual records organized in rows. Each record is composed of columns (also called fields). Every table is defined by a schema that describes the column names, data types, and other information. Two table types are supported in BigQuery:

- native tables – tables backed by native BigQuery storage,
- external tables – tables backed by storage external to BigQuery.

Metadata of both of these table types contains the following information:

- name – a name for the table that is unique per dataset,
- description (optional) – a description for the table,
- friendly name (optional) – a descriptive name for the table,
- schema (optional) – a definition of the schema of the table that contains the following information about every column of the table:
 - name – a name for the column,
 - type – a data type for the column,
 - mode (optional) – the column mode that tells whether the column is nullable, required, or repeated,
 - description (optional) – a descriptive name for the column.

In addition to the mentioned attributes, external table metadata also provides information about source URIs and source format for the data.

External tables are part of external data sources functionality in BigQuery. It is a data source that can be queried directly even though the data are not stored in BigQuery. Instead of loading or streaming the data, the table references the external data source. Four data source types are supported: Cloud Bigtable, Cloud Storage, Google Drive, and Cloud SQL[19]. Every time user queries an external data source, the data are loaded from the specified location and then processed in BigQuery.

Metadata about the table resource also contains information about encryption configuration, clustering, range partitioning, and others. However, this information is not relevant for building dataflows and it is not going to be saved.

2.2.3.4 View

A view is a virtual table defined by a SQL query, and user queries it in the same way as a table.[20] Two types of views are supported in BigQuery:

- view,
- materialized view.

Materialized views are precomputed views that periodically cache results of a query for increased performance and efficiency.[21] A materialized view is relatively new functionality in BigQuery. It was added in a beta release on the 8th of April 2020.[22] That is why the extraction of metadata about this resource is not covered by this master thesis and will not be implemented in a proof of concept application.

Metadata about views are very similar to the metadata about tables, and even its extraction is done with the help of the same resources. To distinguish

a view from a table when querying metadata, the “type” attribute is used. This attribute can have the following values: `TABLE`, `EXTERNAL`, `VIEW` and `MATERIALIZED_VIEW`. Except for table metadata attributes, the view definition attribute is extracted too. This attribute contains a query string used to define the view, and it is useful to understand the source of data in the view.

2.2.3.5 Function

BigQuery supports user-defined functions and lets users define functions with SQL expressions or JavaScript code. The following metadata about function are extracted:

- name – a name for the function that is unique per dataset,
- function definition – SQL or JavaScript body of the function,
- return type (optional) – data type of the returned value,
- language (optional) – specification of the programming language used to a define function, either SQL or JavaScript,
- imported libraries (optional) – stores the path of the imported JavaScript libraries,
- parameters (optional) – list of function arguments. Every argument has its metadata:
 - name – name of the argument,
 - data type – a data type of the argument.

JavaScript functions in BigQuery allow users to write function definition in JavaScript programming language. It also allows using other JavaScript libraries that have to be predefined during the creation of function with the help of imported libraries attribute. List of user-defined functions and metadata about them are retrieved from the following resources:

- <https://bigquery.googleapis.com/bigquery/v2/projects/{projectId}/datasets/{datasetId}/routines>
- <https://bigquery.googleapis.com/bigquery/v2/projects/{projectId}/datasets/{datasetId}/routines/{routineId}>

2.2.3.6 Procedure

Procedures are another type of resource that is available to BigQuery users. In its core procedure is a block of statements that can be called from other statements. Procedures are similar to user-defined functions, except it can be written only in SQL language, there is no return type, and its arguments have

a mode. Mode allows to specify whether it is input, output, or both input and output argument. Procedures metadata are retrieved with the same resource as user-defined functions in BigQuery. In order to distinguish between both entities, the attribute “type” is used. It can have a “SCALAR_FUNCTION” or “PROCEDURE” value.

2.3 Analysis of BigQuery SQL

BigQuery has its own SQL dialects, which are similar to SQL dialects of other known database systems such as PostgreSQL or Oracle but with some differences and additional clauses. This section focuses on describing BigQuery SQL dialects and their syntax.

2.3.1 Standard and legacy SQL

BigQuery supports two SQL dialects: standard SQL and legacy SQL. Legacy SQL was initially named BigQuery SQL and was renamed to legacy SQL after the release of the new dialect. Standard SQL is ANSI:2011 compliant and has extensions that support querying nested and repeated data.

The standard SQL is the preferred SQL dialect for querying data stored in BigQuery. New features added to the BigQuery platform are not being backported to legacy SQL. The documentation for standard SQL covers more details and is maintained better than for the legacy dialect. Default dialect depends on the tool that is being used for interaction with BigQuery: [23]

- in the Cloud Console and the client libraries, standard SQL is the default,
- in the classic BigQuery web UI, the bq command-line tool, and the REST API, legacy SQL is the default.

Even though the default dialect is set, the user may change it with the provided configurations. There is also the ability to set preferred dialect directly through the SQL with `#legacySQL` and `#standardSQL` prefixes.

The proof of concept application supports script analysis for statements written both in standard and legacy SQL. The following subsections focus on discussing statements supported in standard SQL dialect with remarks regarding differences in legacy SQL.

2.3.2 Data types

Table 2.1: Data types in BigQuery legacy and standard SQL

Standard SQL	Legacy SQL	Description
BOOL	BOOLEAN	Boolean values are represented by the keywords TRUE and FALSE.
INT64	INTEGER	Numeric values that do not have fractional components.
FLOAT64	FLOAT	Double precision (approximate) decimal values.
NUMERIC	NUMERIC	Decimal values with 38 decimal digits of precision and 9 decimal digits of scale.
STRING	STRING	Variable-length character (Unicode) data.
BYTES	BYTES	Variable-length binary data.
STRUCT	RECORD	Container of ordered fields each with a type (required) and field name (optional).
ARRAY	REPEATED	An ordered list of zero or more elements of non-ARRAY values.
TIMESTAMP	TIMESTAMP	Representation of an absolute point in time.
DATE	DATE	Representation of a logical calendar date, independent of time zone.
TIME	TIME	Representation of time independent of a specific date and timezone.
DATETIME	DATETIME	Representation of date and time with the range 0001-01-01 00:00:00 to 9999-12-31 23:59:59.999999.
GEOGRAPHIC	-	A collection of points, lines, and polygons, which is represented as a point set, or a subset of the surface of the Earth.

BigQuery supports both simple and complex data types such as `ARRAY` or `STRUCT`. Table 2.1 contains a list of all data types with their names in both of the dialects. `GEOGRAPHY` is supported only in standard SQL. There is also limited support for data types such as `DATE`, `TIME`, `DATETIME`, and `TIMESTAMP` in legacy SQL.

Table 2.2 shows a list of data type properties, which implies restrictions on storing and querying data with specific types.

Table 2.2: Data type properties in BigQuery standard SQL

Property	Description	Data types
Nullable	Value can be null.	All data types except <code>ARRAY</code> s can be nullable.
Orderable	Allowed in <code>ORDER BY</code> .	All data types except for <code>ARRAY</code> , <code>STRUCT</code> , <code>GEOGRAPHY</code> .
Groupable	<code>PARTITION/GROUP BY</code> or <code>DISTINCT</code> can use it.	All data types except for <code>ARRAY</code> , <code>STRUCT</code> , <code>GEOGRAPHY</code> .
Comparable	Values of the same data type can be compared.	All data types except <code>ARRAY</code> s and <code>GEOGRAPHY</code> s. <code>STRUCT</code> data type has limited support for comparison.

2.3.3 Lexical Structure

This subsection describes the lexical structure of BigQuery statements. BigQuery statement consists of a series of tokens separated by whitespace or comments. Tokens include identifiers, quoted identifiers, literals, keywords, operators, and special characters.[24]

2.3.3.1 Identifiers

Identifiers are names that are associated with columns, tables, and other database objects. They can be unquoted or quoted.[24] Backticks are used in standard and brackets in legacy SQL for quoted identifiers. Below are examples of the same query using quoted identifiers written in standard and legacy SQL:

```

1 #standardSQL
2 SELECT * FROM
3 `bigquery-public-data.covid19_open_data.covid19_open_data`;

```

Listing 2.1: Quoted identifiers in standard SQL

```
1 #legacySQL
2 SELECT * FROM
3 [bigquery-public-data:covid19_open_data.covid19_open_data];
```

Listing 2.2: Quoted identifiers in legacy SQL

Quoted identifiers may contain special characters, such as spaces or symbols, but cannot be empty. Unquoted identifiers can contain letters, numbers and underscores but cannot begin with a number.

There are multiple syntactically correct ways to quote an identifier in standard SQL. The identifier can consist of multiple name segments. In listing 2.1, the identifier consists of three name segments: name of the project at the start, name of the dataset in the middle and name of the table in the end. Every segment could be quoted separately, or only part of the identifier, that uses symbols that have to be quoted, may be quoted.

There are additional restrictions to identifiers used for objects such as table, view, column, etc.

2.3.3.2 Literals

A literal is a representation of a constant value of a built-in data type. Most of the literals in BigQuery have the same syntax as in other known database systems. `ARRAY` and especially `STRUCT` data types are somehow unique to BigQuery technology and their literals have special syntax.

Syntax of struct literal is the following:

```
1 (expr, expr [, ...])
```

Listing 2.3: Struct literal syntax

Where “`expr`” is an element in the struct. There must be at least two expressions specified or otherwise, it is indistinguishable from an expression wrapped in parenthesis. The literal above will output a struct with anonymous fields in which data types are inferred from the input expressions. Having anonymous fields means that none of the elements can be referenced. There is also a syntax for naming elements of the struct and explicitly entering the struct literal data type.

`ARRAY` data type is not unique to BigQuery, and its literal syntax is similar to other database technologies.

```
1 [ ARRAY[<data_type> ] "[" [ expr [, ...] ] ] "]"
```

Listing 2.4: Array literal syntax

`ARRAY` keyword is optional, and explicit specification of the array elements’ data type is also optional but cannot exist without the `ARRAY` keyword. Then follows a list of expressions enclosed in square brackets. Arrays can be empty.

The `GEOGRAPHY` data type mentioned earlier cannot be expressed as literal. In order to work with it, BigQuery provides built-in functions. Some of them are in the following list:

- `ST_GEOGPOINT`, `ST_MAKELINE`, `ST_MAKEPOLYGON` – build new geography values from coordinates,
- `ST_GEOGFROMGEOJSON`, `ST_GEOGFROMWKB` – create geographies from an external format such as WKB and GeoJSON.

2.3.3.3 Case sensitivity

Case sensitivity of BigQuery identifiers differs based on the category of the object it is used for. Following table 2.4 provides an overview of known and documented rules.

Table 2.3: Case sensitivity in BigQuery

Category	Is case sensitive?
Keywords	No
Built-in Function names	No
User-Defined Function names	Yes
Table names	Yes
Column names	No
Aliases within a query	No

Whether the identifier is quoted or unquoted, it does not affect its case-sensitivity.

2.3.3.4 Keywords

Keywords are other building blocks of every programming language. They have special meaning in the language, and in BigQuery there is only one restriction that is put on their usage: keywords cannot be used as identifiers unless they are enclosed in backticks. The full list of reserved keywords can be found in BigQuery documentation.^[24]

2.3.4 Query syntax

Query statements are intended to scan one or more tables or expressions and provide the result rows. In BigQuery, the query statement is represented with the `SELECT` statement. `SELECT` statement syntax both in standard and legacy SQL is similar SQL dialect in other databases such as PostgreSQL. BigQuery supports the following clauses in the `SELECT` statement:

Table 2.4: SELECT clauses in BigQuery

Clause	Standard SQL	Legacy SQL
SELECT	Yes	Yes
FROM	Yes	Yes
JOIN	Yes	Yes
WHERE	Yes	Yes
GROUP BY	Yes	Yes
HAVING	Yes	Yes
ORDER BY	Yes	Yes
LIMIT	Yes	Yes
WITH	Yes	No
WINDOW	Yes	No
OFFSET	Yes	No
EXCEPT	Yes	No
FOR SYSTEM_TIME AS OF	Yes	No
REPLACE	Yes	No
OMIT RECORD IF	No	Yes

Table of the supported clauses 2.4 contains some clauses that are not very popular among other technologies or may be known under a different name. To better understand the concepts of query statements, some of them will be discussed in the following subsections.

2.3.4.1 EXCEPT clause

EXCEPT clause is defined as a part of SELECT * EXCEPT statement.

```

1  #standardSQL
2  WITH actors AS (SELECT 'Ivan' first_name, 'Trojan' last_name,
3                     125 movie_cnt)
4  SELECT * EXCEPT(First_Name) FROM actors;
5  +-----+-----+
6  | last_name | movie_cnt |
7  +-----+-----+
8  | Trojan   | 125      |
9  +-----+-----+

```

Listing 2.5: EXCEPT clause example

Code 2.5 shows an example of usage of this clause. It aims to specify one or more columns to exclude from the resultset. Columns to exclude are specified with its names that are case insensitive in BigQuery.

It is important to know that this clause cannot exclude columns that do not have names. So for example, code in the listing 2.6 cannot be executed due to compilation error.

```

1 #standardSQL
2 SELECT * EXCEPT(First_Name) FROM (SELECT 1, 2, 3);
3 -- Column First_Name in SELECT * EXCEPT list does not exist

```

Listing 2.6: EXCEPT clause error example

2.3.4.2 FOR SYSTEM_TIME AS OF clause

The FOR SYSTEM_TIME AS OF is an optional clause that can be specified after the name of the table in FROM clause. This allows to reference the historical versions of the table definition and data in it.

```

1 #standardSQL
2 SELECT count(*) AS count
3 FROM bigquery-public-data.crypto_ethereum.tokens;
4 +-----+
5 | count |
6 +-----+
7 | 193488 |
8 +-----+
9
10
11 SELECT count(*) AS count
12 FROM bigquery-public-data.crypto_ethereum.tokens
13 FOR SYSTEM_TIME AS OF
14 TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 6 day);
15 +-----+
16 | count |
17 +-----+
18 | 192676 |

```

Listing 2.7: FOR SYSTEM_TIME AS OF clause example

In code example 2.7, the second statement selects historical data of the same table used in the first statement. There are two restrictions on the value of the timestamp expression:

- it cannot have value more than seven days before the current timestamp,
- it cannot be in the future.

The FOR SYSTEM_TIME AS OF clause is also supported in other data warehouse solutions. For example, in Snowflake, it is known as AT or BEFORE clause[25].

2.3.4.3 REPLACE clause

This clause is used as a part of the `SELECT * REPLACE` statement. It contains one or more `expression AS identifier` clauses where the `identifier` is a column name. It replaces the value of the matching column in the resultset with the value of the `expression`. An example of the `REPLACE` clause can be found in 2.8.

```

1  #standardSQL
2  WITH actors AS (
3    SELECT 'Jiri' first_name, 'Machacek' last_name,
4           85 movie_cnt, 2 movie_in_progress_cnt
5  )
6  SELECT * REPLACE(
7     movie_cnt + movie_in_progress_cnt AS movie_cnt,
8     0 AS movie_in_progress_cnt)
9  FROM actors;
10
11 +-----+-----+-----+-----+
12 | first_name | last_name | movie_cnt | movie_in_progress_cnt |
13 +-----+-----+-----+-----+
14 | Jiri       | Machacek  | 87        | 0                      |
15 +-----+-----+-----+-----+

```

Listing 2.8: REPLACE clause example

2.3.4.4 OMIT RECORD IF clause

The `OMIT RECORD IF` clause is a construct that is unique to BigQuery. Its behavior is similar to the `WHERE` clause — it allows to filter records based on the condition.

```

1  #legacySQL
2  SELECT repo_name FROM
3     [bigquery-public-data:github_repos.languages]
4  OMIT RECORD IF
5     COUNT(language.name) < 160
6  +-----+
7  | repo_name          |
8  +-----+
9  | polyrabbit/polyglot |
10 +-----+

```

Listing 2.9: OMIT RECORD IF clause example

In comparison to the `WHERE` clause, it has two differences. The records will be omitted from the resultset if the condition is evaluated to true, and kept otherwise. The second difference is the ability to use scoped aggregation functions in its condition, which may be useful when working with repeated and nested data types.

In example 2.9, names of repositories that have less than 160 languages used in it are filtered. In this case, the “language” field is an array of records with one of the sub-fields named “name”.

2.3.5 Data manipulation language

BigQuery is primarily a data warehouse into which data are loaded or streamed and left unmodified.[6] However, BigQuery supports data manipulation language (DML), which enables to insert, update, and delete records in tables. The statements of this type can be executed only with standard SQL. It is important to note that each DML statement is executed in an implicit transaction, which means that changes are committed automatically in case of success. There is no support for multi-statement transactions.

In the following subsections, a brief overview of the available DML statements will be presented.

2.3.5.1 INSERT statement

The `INSERT` statement’s behavior is the same as in other databases — it adds records to the target table. There is an ability to use a list of values or `SELECT` statement to specify what has to be inserted. Insertion of multiple rows at once within the `VALUES` clause and the `SELECT` statement is supported. In code 2.10, there is an example of an `INSERT` statement in BigQuery.

```

1 #standardSQL
2 INSERT actors(first_name, last_name, movie_cnt)
3   VALUES ('Jiri', 'Machacek', 85),
4           ('Ivan', 'Trojan', 125);

```

Listing 2.10: `INSERT` statement example

The specification of the list of columns in the target table is optional.

2.3.5.2 DELETE statement

The `DELETE` statement is used to remove records from the target table. The `WHERE` clause specifying rows to remove is mandatory. To delete all rows from the table `WHERE true` can be used.

```

1 #standardSQL
2 DELETE actors AS a
3   WHERE a.first_name = 'Ivan';

```

Listing 2.11: `DELETE` statement example

As in example 2.11, an alias for the target table can be specified and later used in the query. It is important to note that after specifying the alias, the original target table’s name can no longer be used in qualified names of the columns in `WHERE` clause.

2.3.5.3 TRUNCATE TABLE statement

Except for DELETE, the TRUNCATE TABLE statement can be used to delete rows in the target table. This statement deletes all the rows from the table but leaves schema, description and labels of the target intact.

```
1 #standardSQL
2 TRUNCATE TABLE actors;
```

Listing 2.12: TRUNCATE TABLE statement example

2.3.5.4 UPDATE statement

The UPDATE statement is used to update existing rows in the table. BigQuery supports the following clauses of the UPDATE statement:

- SET – list of columns which have to be updated with specified expression value,
- FROM – a list of tables, whose columns can appear in WHERE or SET clause,
- WHERE – mandatory clause to specify the condition that row has to satisfy in order to be updated.

Nested and repeated fields also can be updated with the help of this statement. In the case of the column of type struct, the full path to the sub-field has to be specified.

```
1 #standardSQL
2 UPDATE actors
3 SET movie_cnt = movie_cnt + 1, movies = ARRAY(
4     SELECT movie FROM UNNEST(movies) AS movie
5     UNION ALL
6     SELECT ('Charlatan', CAST('2020-08-20' AS DATE))
7 )
8 WHERE first_name = 'Ivan' and last_name = 'Trojan';
```

Listing 2.13: UPDATE statement example

In example 2.13, for actors with the first name “Ivan” and last name “Trojan”, the number of movies is increased by one and to the array of their movies a movie with name “Charlatan” and release date August 20, 2020 is added.

2.3.5.5 MERGE statement

A MERGE statement is an atomic combination of INSERT, UPDATE, and DELETE operations, that runs (and succeeds or fails) as a single statement.[6] In BigQuery, this statement consists of the following building blocks:

- `target_name` – the name of the table that is going to be changed,

- `source_name` – table or subquery that is used as a source of the data to be updated or inserted,
- `merge_condition` – boolean expression that is used by the JOIN to match rows in source and target tables,
- `when_clause` – allows specifying WHEN MATCHED, WHEN NOT MATCHED and WHEN NOT MATCHED BY SOURCE options and INSERT, UPDATE or DELETE statements. Each MERGE statement must contain at least one `when_clause`,
- `search_condition` – the optional clause that can be combined with the `when_clause`. The `when_clause` statement is executed for a row only if `merge_condition` and `search_condition` are satisfied.

One of the limitations of the MERGE statement is the inability to use correlated subqueries within a `when_clause`, `search_condition`, UPDATE or INSERT statement inside the `when_clause`.

```

1  #standardSQL
2  MERGE actors a
3  USING
4  (SELECT 'Ivan' first_name, 'Trojan' last_name, 86 movie_cnt) s
5  ON a.first_name = s.first_name AND a.last_name = s.last_name
6  WHEN MATCHED THEN
7     UPDATE
8     SET movie_cnt = s.movie_cnt
9  WHEN NOT MATCHED BY TARGET THEN
10     INSERT(first_name, last_name, movie_cnt)
11     VALUES(first_name, last_name, movie_cnt)
12  WHEN NOT MATCHED BY SOURCE THEN
13     DELETE;

```

Listing 2.14: MERGE statement example

In example 2.14, MERGE statement updates, inserts or deletes data from table “actors” based on the USING clause subselect.

2.4 Requirements analysis

This section lists functional and non-functional requirements for the proof of concept application.

2.4.1 Functional requirements

FR1: Metadata extractor for BigQuery

The implemented solution has to be able to extract metadata about specified BigQuery projects and all related database objects in it. It also includes:

- adding extracted metadata into the data dictionary,

- persisting metadata,
- extracting BigQuery specific information such as database objects hierarchy, built-in functions, and system variables,
- generating DDL scripts based on the extracted metadata.

FR2: BigQuery standard and legacy SQL parsing

Reading and parsing the SQL script in a file or a string is part of the prototype implementation. Parsing of `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE` statements in standard SQL and `SELECT` statement in legacy SQL have to be supported. Because of the continuous work and improvements of the standard SQL language by the BigQuery team, some changes released after the start of the work on the master thesis are expected not to be supported by the implemented prototype.

FR3: Building AST

The result of the parsing should be in the form of an abstract syntax tree. Its structure will allow the convenient traversing and processing during the later phases of implementation.

FR5: Semantic analysis

The implemented prototype has to be able to process the abstract syntax tree created during the parsing phase. It has to recognize declarations of new database objects and add this information to the data dictionary. It also has to find references to database objects and map them to existing data dictionary entities.

FR6: Deduction

The implemented solution must be capable of handling references that are not pointing to any object in the previously created data dictionary. In this case, it has to deduce the referenced entity and register it in the data dictionary.

FR7: Dataflow

Proof of concept application has to create dataflow for the provided BigQuery script with statements listed in second functional requirement and represent it in the form of a graph. Created dataflow graph should also be available in the format that would allow further processing and visualization.

2.4.2 Non-functional requirements

NFR1: Execution time

The implemented solution should extract metadata and parse the provided scripts in time that would be the same or at least close to other solutions in the Manta portfolio. However, the speed of extraction metadata in the case of

BigQuery can be affected by the chosen extraction approach. Every extractor call requires the initialization of a new HTTPS connection, which may slow down the performance.

NFR2: Memory management

The prototype is required to run and succeed under different circumstances and with available resources. One of the resources that may be limited is memory. The prototype, especially the extractor part, should manage used memory correctly and be able to handle the situation, where all the metadata will not fit into the memory.

NFR3: Usage of existing technologies and formats

The proof of concept application will be part of the Manta system, so it should comply with the existing architecture. To achieve this, it should use provided interfaces and classes where possible, save data to the same storage as other technologies, and represent processing results in the format that can be consumed and processed by other components of the Manta system.

NFR4: Implementation quality

The created implementation has to be production quality and follow the programming standards in Manta:

- review – results have to be reviewed by at least one member of the team (not including the developer himself),
- unit-testing – code should be covered with unit tests that make sense,
- development tracking – development has to be tracked and reported regularly with the existing Manta tools,
- extensibility – implemented solution has to be designed in a way that allows adding new features and improvements in the future.

Design

This chapter describes the architecture of the created solution and tools used during the implementation.

3.1 Architecture

The implemented prototype functionalities can be logically separated into three parts:

- extraction part – connects to the target database, extracts the metadata from it, and saves them for later use by other modules,
- parsing and resolving part – is responsible for parsing the input scripts, building the AST and resolving the references in the script,
- dataflow part – uses the results of previous steps to build a graph representation of dataflow and saves it for future processing.

3.1.1 Extractor

As mentioned before, this part is focused on extracting metadata about objects in the target database and saving them for later use. This process consists of the following steps:

- connecting to the target database – includes retrieval of the access token that is required for authorization of every request for metadata from BigQuery API services,
- retrieving the metadata and its processing – includes execution of requests for metadata to the BigQuery platform and converting the raw representation into the designed data model,
- saving metadata into data dictionary – includes mapping of the extracted metadata and adding it to the data dictionary,

3. DESIGN

- generating the DDL – includes generation of data definition language scripts based on the extracted metadata,
- saving DDLs – includes saving the generated DDL created in the previous step,
- persisting the data dictionary – includes persisting of the data in the data dictionary into the persistence storage.

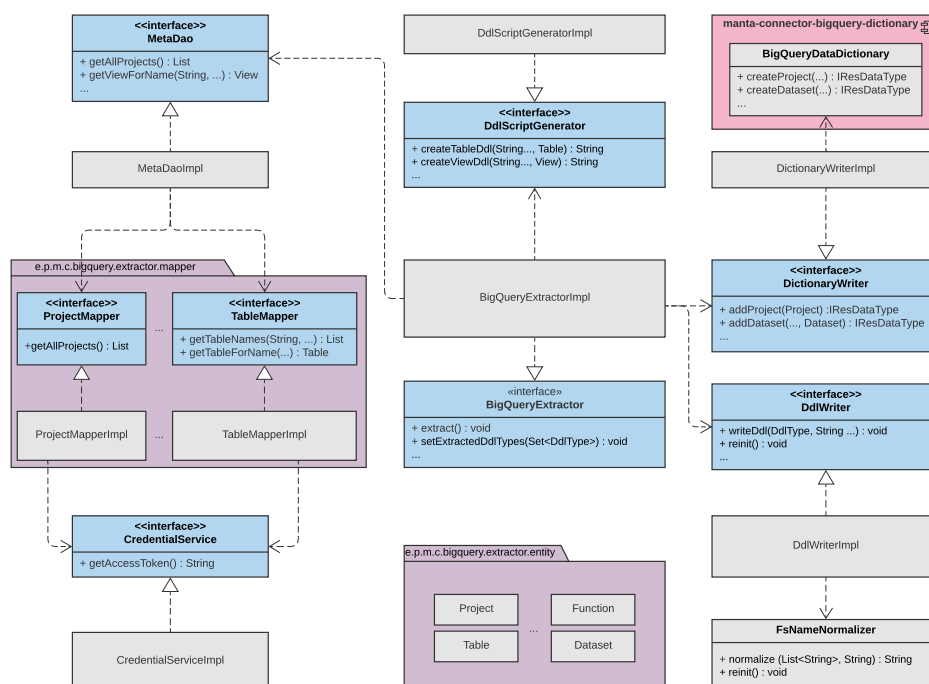


Figure 3.1: BigQuery extractor module class diagram

`manta-connector-bigquery-dictionary-extractor` module contains the extractor functionality. Its classes and interfaces are captured in figure 3.1. The presented class diagram contains only high-level concepts of classes, its attributes and methods, and does not reveal all the details.

The `BigQueryExtractor` is an API that can be used by other modules to run the process of the extraction. Its implementation manages and controls the whole extraction process. It uses the `MetaDao` interface in order to retrieve metadata. The `MetaDao` implementation delegates calls to `*Mapper` interfaces from `eu.profnit.manta.connector.bigquery.extractor.mapper` package, where every interface is focused on retrieving metadata for the concrete type of the object. Retrieving of access token required for authorization

of every request is done with the help of the `CredentialService` interface. The `eu.profnit.manta.connector.bigquery.extractor.entity` package contains the data model of retrieved metadata. After the metadata is extracted and converted, `BigQueryExtractor` uses the `DictionaryWriter` interface to save the data into the data dictionary. `DdlScriptGenerator` is responsible for generating DDL based on the metadata, and `DdlWriter` provides a unified interface for saving the generated script to the file system. Persisting the data dictionary is executed with the help of API available in the data dictionary object.

3.1.2 Parsing and resolving

The following figure 3.2 provides an overview of classes and interfaces in `manta-connector-bigquery-resolver` module.

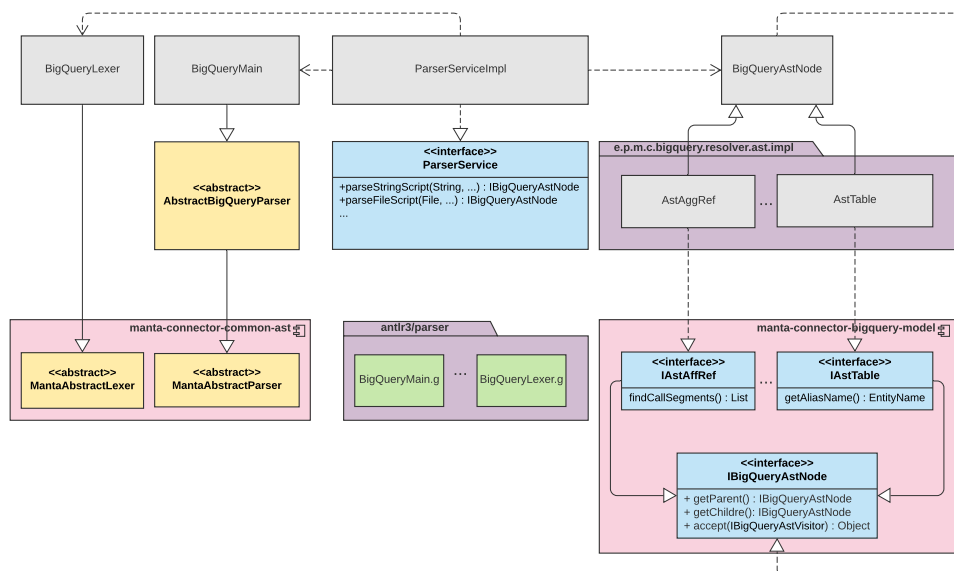


Figure 3.2: BigQuery parsing and resolving module class diagram

The `manta-connector-bigquery-resolver` module contains the implementation of parsing and resolving parts. The parsing part is mainly represented with interface `ParserService` and its implementation. It provides an API for other modules to parse input script in a string or file format and returns the built AST. The main parsing logic is represented by the ANTLR

files located in `antlr3/parser` package. It consists of lexer and parser grammar files. These files are later compiled with ANTLR program and moved into `eu.profinit.manta.connector.bigquery.resolver.parser` package.

The `ParserService` is also responsible for initiating resolving of AST, but the core logic of it is implemented in classes located in `eu.profinit.manta.connector.bigquery.resolver.ast` package and `impl` subpackage. All of the `Ast*` classes extend `BigQueryAstNode` class and implement their own logic of resolving.

3.1.3 Dataflow

Dataflow part is implemented in `manta-dataflow-generator-bigquery` module and its core logic is located in class `FlowVisitor`. `FlowVisitor` class consists of `process` methods responsible for the processing of AST and resulting in a dataflow graph. It also uses an instance of `BigQueryGraphHelper`, which contains some common processing logic. The high-level class diagram of the dataflow module can be seen in figure 3.3.

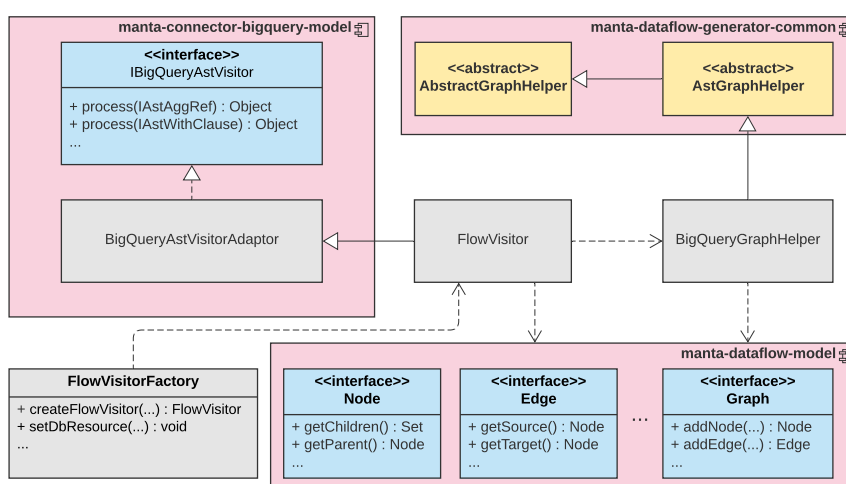


Figure 3.3: BigQuery dataflow module class diagram

The dataflow module mostly depends on three other modules:

- `manta-dataflow-generator-common` – contains interfaces and classes for simplified construction of resulting graph,
- `manta-dataflow-model` – contains building blocks such as `Node`, `Edge`, `Graph` for dataflow graph,

- `manta-connector-bigquery-model` – serves as a connecting point to other implemented modules. It is described in detail in 3.1.4.

3.1.4 Other modules

There are three more modules that have been created during the implementation of the proof of concept application:

- `manta-connector-bigquery-model` – defines interfaces for AST nodes and its processing,
- `manta-connector-bigquery-dictionary` – implements data dictionary for BigQuery,
- `manta-connector-bigquery-dictionary-mapping` – contains the identification of data dictionaries and their configurations.

As shown in 3.2 and 3.3, both the dataflow and the parsing modules depend on the `manta-connector-bigquery-model` module. This module contains interfaces of AST nodes created during the parsing, which will later be processed by the dataflow module. This creates an additional level of abstraction and avoids strict coupling of `manta-connector-bigquery-resolver` and `manta-dataflow-generator-bigquery` modules. This module also defines the interface for the BigQuery data dictionary, which again allows other modules not to depend on the concrete implementation. Except for defining interfaces for AST nodes, the model module defines `IBigQueryAstVisitor`, an interface for traversing AST.

3.2 Tools

The proof of concept application was implemented with the help of tools and technologies that are already used in the Manta system. The following subsections describe some of them.

3.2.1 Java

The implementation is in the programming language Java version 8. Java is a general-purpose, concurrent, class-based and object-oriented language with a long history and good community support.[26] This language is chosen to implement the BigQuery prototype to be consistent and follow design and architecture principles in the Manta system.

3.2.2 Spring

For convenient and fast development of the prototype, Spring framework major version 5 was used. Spring is one of the most popular technologies used in

the Java programming language world. Its first version was released in the year 2002, and it is being developed until now. The framework covers many different areas of programming, and it consists of 20 different modules, which do not have to be used all at once.[27] Spring Context, Spring Beans, Spring Core, Spring Test and Spring JDBC were used to implement the application.

3.2.3 ANTLR

The big part of static code analysis for scripts written in BigQuery SQL language implemented using ANTLR tool version 3. The acronym ANTLR is decoded as ANother Tool for Language Recognition, and it provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions in different programming languages.[28]

During the development of the BigQuery prototype, the following features of the ANTLR tool were used:

- definition of BigQuery lexer, which enables to define rules for lexical analysis,
- definition of BigQuery grammar, which is responsible for syntax analysis,
- definition of rewrite rules as part of parser definition, which allow to design the structure of abstract syntax tree as needed.

3.2.4 Testing tools

To follow Manta software development standards, the implemented functionality was covered with the unit and manual tests. “*A unit test is an automated piece of code that invokes the unit of work being tested, and then checks some assumptions about a single end result of that unit*”.[29] For purposes of unit testing, the JUnit major version 4 framework was used. It is again well known in Java programming language community and perfectly satisfies the testing needs of the implemented prototype.

In combination with JUnit technology, the Mockito version 3 testing tool was used. It allows to simulate behavior of some components the code under test is dependent on. It simplifies the process of testing software that depends a lot on other components or environmental factors. For example, it allows to test the metadata extraction logic itself without being dependent on the real response of BigQuery API service with the help of mocking its responses.

3.2.5 Maven

As discussed in section 3.1, the implemented solution is separated into different modules, and it also depends on other modules existing in the Manta system. For better maintenance, control, and build of application modules, Apache Maven is used. All the definitions of the module, its dependencies, and build

steps are defined within the `pom.xml` file. Maven is later used in combination with the Jenkins tool, which allows using benefits of continuous integration and delivery.

Implementation

This chapter describes the implementation details of the modules described in chapter 3, and provides insight into problems encountered during the development process.

4.1 Extractor

Section 3.1.1 lists six main steps the metadata extraction process consists of. The following subsections describe these steps in more detail.

4.1.1 `CredentialService`

`CredentialService` is the interface that provides a single method called `getAccessToken()`. Other components of the extractor module use this interface to retrieve the access token required for authorization of request for metadata to BigQuery API services.

Because BigQuery API is a part of a more generic Google API system, all rules that apply to Google API also apply to the BigQuery API. Google APIs use the OAuth 2.0 protocol for authentication and authorization. The standard Google API OAuth 2.0 scenario for server applications consists of obtaining service account credentials from the Google API console, requesting an access token from the Google Authorization Server, and sending the access token to the Google API that is needed to be accessed.[30] A service account is a particular type of Google account that belongs to an application instead of an individual end-user.

A service account comes with the public/private key pair. The private key has to be provided to the implementation of `CredentialService` in the form of application properties before the start of extraction. This key is used to sign JSON Web Token that is sent to Google Authorization Service with a request for obtaining the access token. If the JWT token is formed correctly

and the service account has required permissions, the access token response is returned. An example of an access token response can be seen in figure 4.1.

```
1 {
2   "access_token": "1/8xbJqa0ZXSUZbHL15E0tu1pxz3
3     fmmetKx9W8CV4t79M",
4   "scope": "https://www.googleapis.com/auth/bigquery"
5   "token_type": "Bearer",
6   "expires_in": 3600
7 }
```

Listing 4.1: Access token response example

Except for the token value itself, it also contains a token expiration time, which is used for caching purposes of `CredentialService`. Instead of requesting the token every time the request is made, it is requested once and then reused until it expires and then requested again.

4.1.2 Retrieving and converting metadata

Retrieving metadata is done with the help of BigQuery API services, which are accessible with HTTPS protocol. BigQuery API services follow the REST design principle, and metadata to extract are represented as resources, which are described more in detail in subsection 2.2.3.

Package `eu.profinit.manta.connector.bigquery.extractor.mapper` consists of interfaces with suffix `Mapper`, where every interface is dedicated to the extraction of concrete type of database objects metadata. For example, `FunctionMapper` is designed to provide access to metadata about functions. Like most other interfaces in the same package, the `FunctionMapper` defines two methods:

- `getFunctionNames(projectId, datasetName)` – returns list of function names or empty list if there are no functions found in the dataset in the project,
- `getFunctionForName(projectId, datasetName, functionName)` – returns retrieved function metadata or null in case requested function is not found.

The naming of methods in other interfaces is designed accordingly to the type of database object they extract.

Package `eu.profinit.manta.connector.bigquery.extractor.mapper.impl` contains the implementation of the interfaces mentioned above. In most cases, the implementation consists of executing the request for metadata and processing the response. The request for metadata part is very similar for most of the metadata resources and its common part was implemented in

class `BigQueryApiCaller`. This class provides convenient methods for the extraction of a collection and a single metadata object. It is also responsible for recording the amount of time spent on HTTP requests and the number of calls made for statistical purposes. After successful execution of the request, the response is returned and further processing of it is executed in one of the implementations of Mapper interfaces.

```

1  {
2    "etag": "ABeXQI4azA2CQY20+nHXhw==",
3    "routineReference": {
4      "projectId": "bigquery-manta-test",
5      "datasetId": "test",
6      "routineId": "f1"
7    },
8    "routineType": "SCALAR_FUNCTION",
9    "creationTime": "1604250002155",
10   "lastModifiedTime": "1604250002155",
11   "language": "SQL",
12   "arguments": [
13     {
14       "name": "x",
15       "dataType": {
16         "typeKind": "INT64"
17       }
18     }
19   ],
20   "definitionBody": "pow(x, 2)"
21 }

```

Listing 4.2: BigQuery API response for procedure metadata

In figure 4.2, there is an example of response for metadata of function with name `f1`, which returns the value of input integer raised to the power of two. The response is returned in JSON format, where every field of the response and its structure is documented within BigQuery online documentation. The processing of this response is done with the help of Jackson library, which enables to parse the response and get the needed fields using well-designed API methods.

After the response is processed, its information is saved into the designed data model, represented by classes in the `eu.profnit.manta.connector.bigquery.extractor.entity` package. These classes were designed based on the structure of the metadata response for each type of database object. It was also limited only to a subset of metadata fields that are important for further processing. In some cases, additional classes, such as `Column` or `SqlDataType`, were created to have a better data model.

4.1.3 DictionaryWriter

After the successful extraction of metadata, it has to be saved in the data dictionary. `DictionaryWriter` is an interface that defines a set of methods for saving the metadata of all database objects. `DictionaryWriterImpl` is its single implementation, and it handles the mapping of extracted metadata to the internal representation of the data dictionary. It is also responsible for counting the number of saved objects into the data dictionary for statistic purposes.

4.1.4 DdlScriptGenerator

This interface defines a set of methods used to generate DDL scripts for database objects based on the extracted metadata. BigQuery defines data definition language only for a subset of database object types. Therefore `DdlScriptGeneratorImpl` implements the generation of DDL scripts only for tables, views, functions and procedures. Because only a subset of metadata is extracted during the extraction, the original DDL can differ from the generated one, but this fully satisfies the requirements of the implemented prototype. For example, DDL of the `CREATE TABLE` statement can contain `kms_key_name` option, which describes the Cloud KMS encryption key used to protect BigQuery table, but it is not essential for further processing of the implemented prototype and it would be missing in the generated DDL.

4.1.5 DdlWriter

`DdlWriterImpl` implements `DdlWriter` and is focused on saving the generated DDL to the file system. Its method `writeDdl()` has quite a simple logic: creating the name and path in the file system for the DDL script and saving it. The name of the file for the DDL script has to reflect the name of the object defined in the script. There are also some limitations from the file system side: the file's name has to be unique and should not contain symbols that are not allowed. Before creating the name of the file, based on the identifier of BigQuery object, it has to be normalized, which is done with the help of `FsNameNormalizer` class. This class solves the described problems of normalization and returns the file's path and name based on the provided parameters.

4.1.6 Data dictionary persisting

`BigQueryDataDictionary` represents a data dictionary for saving extracted metadata of BigQuery objects. It extends the functionality defined in the class `AbstractDataDictionary`, a part of the common modules defined in the Manta system. Thanks to this extension, it can persist saved metadata to

the persistent storage without any further implementation. The persistence is implemented with the method `persist()` in `AbstractDataDictionary`.

4.1.7 `BigQueryExtractor`

As mentioned earlier in 3.1.1, `BigQueryExtractor`'s `extract()` method uses all components mentioned in this section and orchestrates the process of extraction and saving the metadata. Its implementation `BigQueryExtractorImpl` is configurable with the following methods:

- `setDictionary` – allows to set the instance of data dictionary which is going to be used for saving the extracted metadata,
- `setExtractedDdlTypes` – allows to specify types of objects that are going to be extracted,
- `setOutputDdlTypes` – allows to configure types of database objects, whose DDL are going to be created and saved,
- `setIncludedProjectsDatasets` – allows to specify BigQuery projects and datasets which have to be extracted,
- `setExcludedProjectDatasets` – allows to specify BigQuery projects and datasets which must not be extracted.

It is also responsible for handling exceptions that may occur and collecting the statistics from the used components to process them at the end of the extraction process.

4.2 Parsing and resolving

The section describes the implementation of parsing and resolving parts.

4.2.1 Parsing

Parsing logic is mainly realized with the help of ANTLR tool, and it is located in files with `.g` extension in `antlr3/parser` package. During the phase of building the project, files from this package are processed by the ANTLR tool, and their `java` representations are generated and located into `eu.profinit.manta.connector.bigquery.resolver` package.

4.2.1.1 `BigQueryLexer`

File `BigQueryLexer.g` describes the rules that are used for lexical analysis. The generated Java equivalent is `BigQueryLexer` class, which extends `MantaAbstractLexer`. In turn, this class extends `org.antlr.runtime.Lexer` class and contains some additional error processing logic.

4. IMPLEMENTATION

BigQuery lexer consists of definitions for different types of tokens, such as string and number literals, reserved and non-reserved keywords, operators and special characters. It also contains rules for whitespace and single- or multiline- comments. For example, the rule for single-quoted string literal looks like:

```
1 fragment ANY_CHAR : ~'a' | 'a';
2 fragment SINGLE_QUOTED_STRING : '\\'' (~('\\'' | '\\\\') | ( '\\\
  ANY_CHAR ) ) * '\\'';
```

Listing 4.3: BigQuery single quoted string lexer rule

Section 2.3.3.1 described the specialty of quoted identifiers in standard SQL dialect, which can contain multiple name segments. The `nextToken()` method from the parent class had to be overridden to handle these segments' parsing correctly. The simplified version of this method is presented in figure 4.4

```
1 boolean inBacktickQuotedID = false;
2
3 public Token nextToken() {
4     if(input.LA(1) == '`') {
5         inBacktickQuotedID = !inBacktickQuotedID;
6         mBACKTICK(); // process the backtick character
7     }
8
9     if(inBacktickQuotedID) {
10        while (true) {
11            if(input.LA(1) == '.') {
12                mPERIOD(); // process period character
13            } else {
14                mQuotedFragment(); // process name segment
15            }
16            emit();
17            return state.token;
18        }
19    } else {
20        return super.nextToken();
21    }
22 }
```

Listing 4.4: Overriden `nextToken()` method in `BigQueryLexer`

4.2.1.2 BigQuery parser grammar

BigQuery parser grammar is represented within three files: `BigQueryMain.g`, `BigQueryExpressions.g` and `BigQueryNonReservedKW.g`. This structure was inspired by existing parsers for other SQL dialects in the Manta system.

`BigQueryNonReservedKW.g` mainly consists of `non_reserved_words` and `reserved_words` rules, which help separate reserved and non-reserved keywords into two separate groups. This separation is required because reserved keywords cannot be used as identifiers unless enclosed in backticks. It later

affects the grammar rules focused on parsing parts of the language, which can contain identifiers.

`BigQueryExpressions.g` contains almost the whole parsing logic of `SELECT` statement in `BigQuery` and also some general building blocks, which are used in other statements. These building blocks include expressions, data types, aggregated references and qualified names.

`BigQueryMain.g` is a starting point for parsing every statement, and it also contains parsing logic for data definition and modification language. An example of the rule that recognizes the type of input statement can be seen in figure 4.5:

```

1 common_table_expression_statement
2 @init{
3   registerNewSymbolsScope(ResScope.SELECT_SCOPE);
4 }
5 :
6 (with_clause? (SELECT | LEFT_PAREN)) => s = select_statement
7 -> ^(AST_SELECT_STATEMENT<AstStandaloneSelect>[contextState] $s)
8 |
9 (KW_INSERT) => s = insert_statement
10 -> ^(AST_INSERT_STATEMENT<AstStandaloneInsert>[contextState] $s)
11 |
12 (KW_DELETE) => s = delete_statement
13 -> ^(AST_DELETE_STATEMENT<AstStandaloneDelete>[contextState] $s)
14 |
15 (KW_UPDATE) => s = update_statement
16 -> ^(AST_UPDATE_STATEMENT<AstStandaloneUpdate>[contextState] $s)
17 ;
18 finally {
19   SYMBOLTABLESCOPE_stack.pop();
20 }

```

Listing 4.5: Common table expression statement rule

The figure above 4.5 contains rewrite rules that follow the `->` symbol. This operator of `ANTLR` language allows to define how to generate the output, which is the `AST` in this case. It allows to perform different kinds of transformations, such as reordering nodes, deleting nodes or creating imaginary nodes. In figure 4.5, imaginary nodes `AST_SELECT_STATEMENT`, `AST_INSERT_STATEMENT` and others are created to identify the root of the subtree for the processed statement.

4.2.2 Resolving

The second part of the `manta-connector-bigquery-resolver` is the resolving of the references used in the input script.

4.2.2.1 AST nodes

`BigQueryAstNode` is an ancestor class for every node created in AST. Some of the AST nodes have special meaning and in this case, a special class and interface are defined for their representation. All the classes of AST nodes are put into the `eu.profnit.manta.connector.bigquery.resolver.ast.impl` package. In most cases, these classes contain resolving logic and helper methods that will allow easier traversal of AST and finding required elements in it.

`BigQueryAstNode` implements `IBigQueryAstNode` that declares methods for accessing its parent and children in the AST. Additionally, `BigQueryAstNode` defines `resolve()` method, which in the default implementation does not have any special logic — it just delegates the resolving to its children. This method may be overridden in case the special class is created for an AST node. An example of such class is `AstCreateTable`, representing the root for the `CREATE TABLE` statement in AST.

`resolve()` is the starting point of the whole process of resolving. The result of parsing is represented with an instance of `BigQueryAstNode` pointing to the root of the abstract syntax tree. In case the resolving is requested, the `ParserServiceImpl` calls `resolve()` method on the AST root node, which delegates it to its children.

`ParserServiceImpl` is the implementation of `ParserService`, which declares several methods for parsing and resolving the input scripts.

4.2.2.2 Context concept

Many AST node classes implement the `AstInteranlResolve` interface in addition to implementing the `resolve()` method. This interface defines method `resolveInternal(Stack<Map<EntityName, IResObject>>)` with one parameter called `context`. This allows to execute resolving of the node with additional information that may be defined in the different part of the statement and restrict the lookup of references to a special range of variables available at this moment. An example of its usage can be seen in the following statement: 4.6

```
1  SELECT
2    (SELECT c2
3     FROM t2
4     WHERE CAST(c1 AS INT64) = t1.c1
5    )
6  FROM t1;
```

Listing 4.6: SELECT statement with scalar subquery

Assuming that column `c1` exists in table `t2`, unqualified column name `c1` inside the subselect statement must be resolved to the column of table `t2` even though the column with the same name exists in table `t1`. If at the

moment of entering the subselect, the new layer with variables referencing the columns of table `t2` was not created in the context, `c1` could reference columns in both tables `t1` and `t2`.

4.3 Dataflow

This section describes the `manta-dataflow-generator-bigquery` module implementation. `FlowVisitor` and `BigQueryGraphHelper` are two classes that contain most of the dataflow building logic.

4.3.1 Dataflow graph

The dataflow phase accepts the resolved AST on input and produces a dataflow graph. Dataflow graph is mainly represented with interfaces `Graph`, `Node` and `Edge`, which are part of the `manta-dataflow-model` module. The `Graph` interface defines methods for adding, removing and changing nodes and edges in the graph.

The implemented prototype does not work with the mentioned interfaces directly but instead uses an abstraction, which is represented by helper classes such as `AstGraphHelper` and `AbstractGraphHelper`. Some of the methods that they provide are the following:

- `addNode(String name, NodeType type, Node parent):Node` – adds a node to the graph with the specified name, type and parent node,
- `addDirectFlow(Node source, Node target):Edge` – adds a direct edge between two nodes,
- `addFilterFlow(Node source, Node target):void` – adds a filter edge between two nodes.

`BigQueryGraphHelper` extends abstract class `AstGraphHelper` and adds helper methods that are common for the processing of different statements. It is important to understand the dataflow graph's underlying model even though it is not used directly.

4.3.2 FlowVisitor

The visitor pattern is used for processing nodes of AST created during the parsing phase. `FlowVisitor` extends `BigQueryAstVisitorAdaptor` class which implements `IBigQueryAstVisitor` interface. This interface declares methods with signature `process(IAst*)` for processing AST nodes, which are important for creating the dataflow graph. Classes of AST nodes that have to be processed have to implement the `accept(IBigQueryAstVisitor)` method to be correctly processed by the visitor.

4. IMPLEMENTATION

The goal of `FlowVisitor` is to process input AST nodes and generate a related part of the dataflow graph. The logic of creating the dataflow graph was mostly inspired by other existing SQL dialects in the Manta system.

Testing

This chapter describes the process of testing the proof of concept application. To ensure the correctness of the implemented functionality, unit, integration and performance tests were created. Unit and integration tests can be executed automatically, and in combination with continuous integration tools, allow the development of new features with a lower risk of breaking existing functionalities.

5.1 Extractor

The extractor part covers the extraction and saving of the metadata of objects from the target database. The extractor module is responsible not only for integration to the target platform but also for converting, mapping and saving the metadata.

5.1.1 Unit and integration tests

For most of the components described in 4.1, unit tests were created in the following classes:

- `AbstractRoutineMapperImplTest`, `FunctionMapperImplTest` – cover the more complex logic of mapping the JSON response to the data model,
- `DictionaryWriterImplTest` – verifies the `DictionaryWriterImpl` service, which saves metadata to the data dictionary,
- `DdlScriptGeneratorImplTest` – compares the generated DDL scripts with expected ones,
- `DdlWriterImplTest` – checks that files with generated DDL are saved to the expected location and with the expected content,

- `BigQueryExtractorImplTest` – tests different scenarios of extraction, such as error handling in case of an unexpected error, or handling of different configurations of the extractor.

Except for unit tests, integration tests were also implemented in the extractor module. `MetaDaoImplTest` tests the process of connecting to the target database and requesting the metadata from it. It also tests the part of the converting and mapping logic of the metadata to the metadata model. The previously mentioned `BigQueryExtractorImplTest` also contains tests of the complete extraction process with integration to the BigQuery testing account created for these purposes.

5.1.2 Memory management and performance tests

To be compliant with memory management and time execution requirements, a set of manual tests were executed on the extractor module.

Memory management test aims to verify that extractor works correctly even with limited available resources. This test uses `-Xmx` Java option that limits Java heap size. This test execution showed that the BigQuery extractor could handle the extraction and persisting of 3250 different database objects with Java heap size set to 70, 65 and 60 megabytes. These results satisfy the requirements put on the prototype.

Performance test executes extraction process on big enough target instance of BigQuery. For this test, five projects were created, and each of them consists of 1000 tables, 1000 views and 1000 functions. The average time of multiple executions of this test is 52 minutes, which means that almost five objects are processed per second. This result is slower compared with other technologies in the Manta system. The main reason for worse performance might be using HTTPS protocol in combination with the REST API service of BigQuery instead of JDBC, which is usually used in the Manta system for extraction purposes.

5.2 Parsing and resolving

The `manta-connector-bigquery-resolver` module focuses on parsing input scripts and resolving references in built AST. This module is covered by unit tests and it logically can be divided into two groups: tests of parsing and tests of resolving.

5.2.1 Parsing

Parsing tests verify that input scripts can be parsed without errors and that the AST structure is built according to the design. These tests are executed with class `AnnotatedFilesResolverTest`, which iterates over the set of input

scripts in directory `test/resources/SimpleTests` and does parsing for each of them. At the end of the parsing, the test checks that there are no error or warning messages in the log output, verifying that the parsing did not fail. The output AST representation is not compared to any expected result but was checked manually during implementation.

Manual tests were done with the tool `GraphViz`. This tool takes AST, converted to DOT format, and prints it as an image to the specified file. An example of such image can be seen in figure 5.1, which shows the AST for a simple `INSERT` statement with a `VALUES` clause.

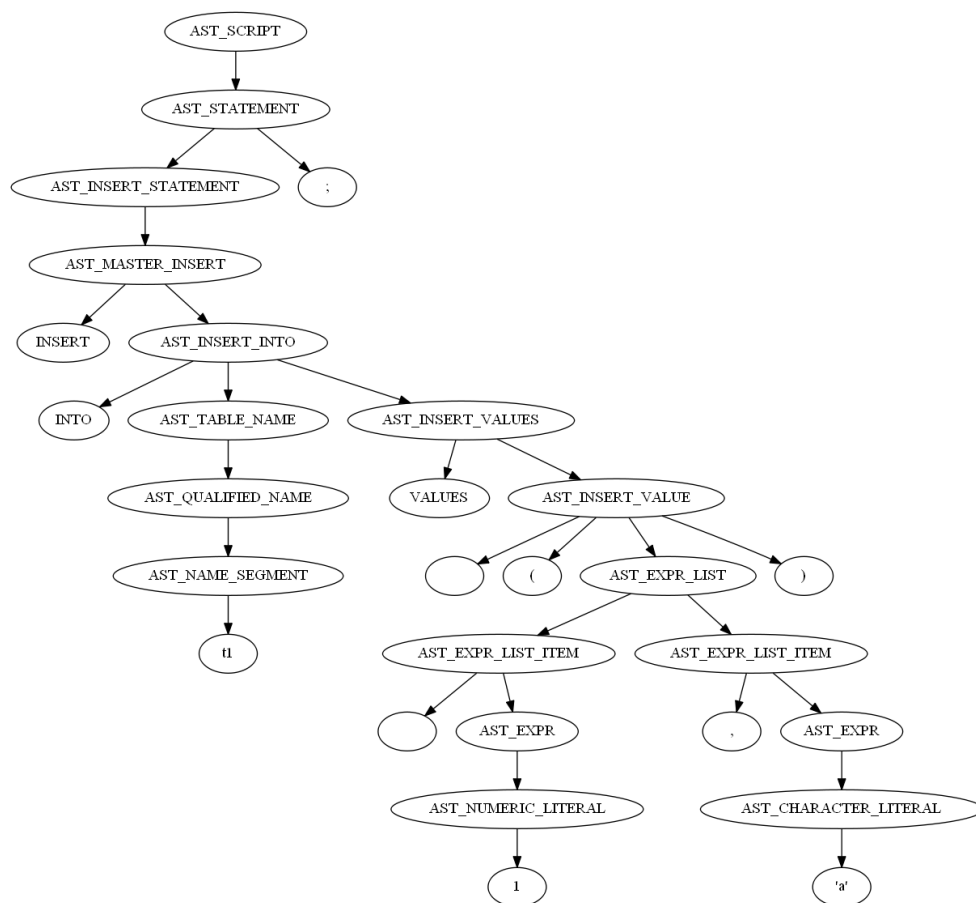


Figure 5.1: AST for `INSERT` statement

5.2.2 Resolving

Resolving tests are executed in the same manner as parsing tests but with an additional step checking the resolved references. Manta system has developed

5. TESTING

its own set of utilities, which simplify the process of verification the results of resolving.

These utilities take SQL script on input, which is annotated by multi-line comments. With the help of these annotations, the following properties of a resolved entity can be verified: it is correctly resolved, i.e., it references the expected object, it is not deduced, it has the correct type, it has the right parent, it has the correct name. Also, names of the columns of the resultset and structure of result AST can be checked.

```
1 #standardSQL
2 CREATE TABLE t1 /* name = "T1" */ /*= t1*/ (
3   a /*= t1a */
4     /* parentEntity = t1 */
5     /* parentEntity.hasProperty(DB_TABLE) */
6     INT64,
7   b STRING
8 );
9
10 SELECT
11 a /*= t1a */ /* hasProperty(COLUMN) */ /* !deduced() */
12 AS aAlias /* `ancestor::AST_ALIAS` != null */
13 FROM t1 /*= t1 */;
14
15 SELECT * FROM t1;
16 /* a | b */
```

Listing 5.1: Resolving tests with annotations example

In listing 5.1, there is an example of the usage of almost all types of annotations that can be applied.

5.3 Dataflow

The functionality of this module is also covered with unit tests. Test class `AstFilesFlowTest` works similarly to tests in parsing and resolving modules. It takes input scripts, parses and resolves them, and then generates dataflow graph. Created dataflow graph can be serialized to a string, which allows a simple comparison of it with the expected result. Serialization of dataflow graph is done with `toString(Graph)` method of `GraphUtils` class from the `manta-common-testutils` module. This class is also capable of deserializing the string back into `Graph` instance.

Input scripts and expected results are located in `test/resources/flow` directory. `AstFilesFlowTest` is configured to accept files with extension `sql` and compare the generated graphs with the content of files with the same name but suffixed with `_expected.txt`.

During the development process, manual tests were also executed. Dataflow graph is converted to DOT format and is printed as an image with `GraphViz` tool. Output examples can be seen in chapter 6.

Result examples

This chapter contains an example of processing a simple input SQL script and its dataflow graph. It also describes the purpose of nodes and edges in the resulting graph.

6.1 CREATE TABLE statement

Dataflow graph will be generated for BigQuery CREATE TABLE statements shown in code example 6.1. CREATE TABLE statement for the table named `t1` does not generate dataflow itself, but the statement for the table named `t2` does. In BigQuery, CREATE TABLE AS statement allows to create a new table based on the resultset of SELECT statement. In this case, the SELECT statement will work with table `t1`.

```
1 #standardSQL
2 CREATE TABLE t1 (
3     a INT64,
4     b INT64,
5     c INT64
6 );
7
8 CREATE TABLE t2 AS
9     SELECT b, c
10    FROM t1
11    WHERE a = 42;
```

Listing 6.1: CREATE TABLE and CREATE TABLE AS examples

In figure 6.1, there is the dataflow graph generated for these statements, which is limited to the subset of nodes that play a role in this case. Every node in the figure consists of the following elements:

- position in the script, where it is used. This information is captured inside `<>` symbols, usually at the left hand side of the node,

6. RESULT EXAMPLES

- name of the node, usually on the left hand side or right after the position in the script information,
- node type inside brackets,
- parent of the node inside the parenthesis.

Nodes are connected with labeled edges, which represent the flow of the data in the statement. There are two categories of edge labels used in this graph: D stands for direct and F for filter flow. For example, between column **b** in table τ_1 and column **b** in table τ_2 , there is a path that consists of direct edges and nodes related to the representation of the **SELECT** statement. This path describes that data from column **b** in table τ_1 are used as the origin of data in column **b** in τ_2 . On the other hand, there is a $\langle 11,5 \rangle$ Where node in the middle of the graph, which has filter edges to 1 **b** and 2 **c** **ResultSetColumn** nodes. This connection describes that resultset columns of the **SELECT** statement might be affected by **WHERE** clause.

This way graph represents dataflows for the **CREATE TABLE AS** statement with detailed information about building blocks of the newly created table.

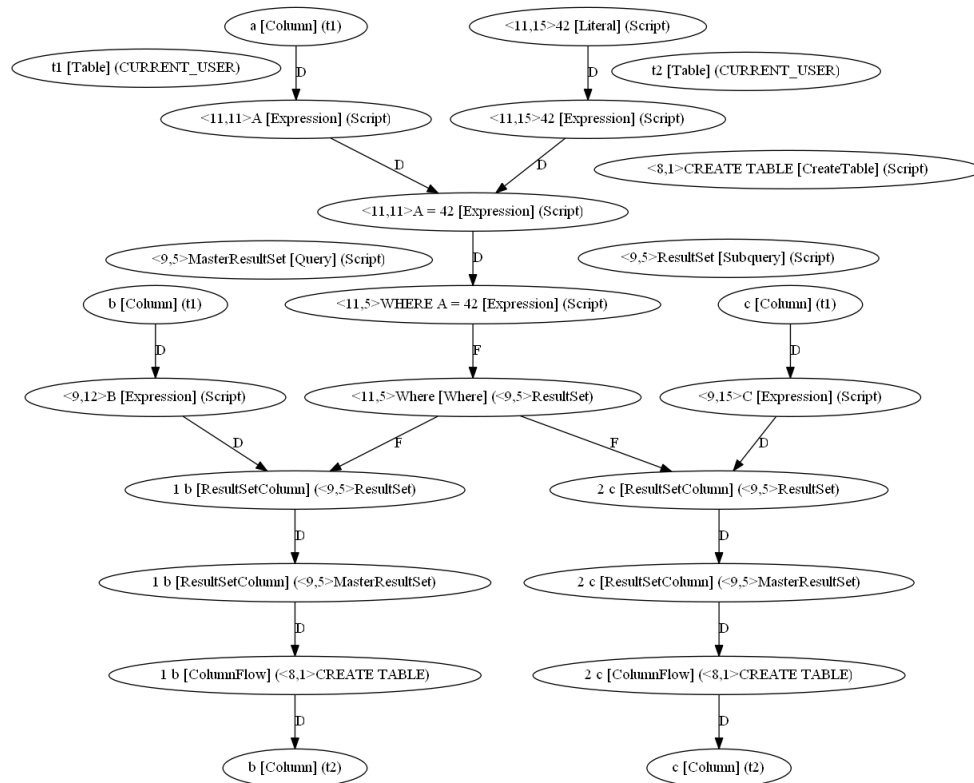


Figure 6.1: Dataflow graph for **CREATE TABLE AS** statement

Conclusion

The objective of this master thesis was to analyze BigQuery technology and semantics of its query language, learn about the Manta system and its approach to static code analysis and data lineage. It was required to implement the extraction of metadata needed for dataflow analysis and design a way to analyze and represent the source codes in the Google BigQuery language. The final goal was to implement a proof of concept application that will extract dataflow from a set of Google BigQuery scripts to the Manta system and cover it with proper quality tests.

All the mentioned objectives were accomplished. A way to analyze the source code of Google BigQuery language was found, and relevant dataflow declarations and statements were detected. The proof of concept application was implemented, and it was extensively tested.

The implemented application does not cover analysis of all the statements existing in Google BigQuery language, but it is able to process most essential constructs and concepts in it. It also does not provide the full support for non-atomic objects of `STRUCT` data type due to the complexity of its processing and representation. The design and implementation of the proof of concept application are done in a way that allows extending current functionalities and adding support for other statements of the BigQuery language.

The implemented solution will be integrated into the Manta system and added to the portfolio of supported technologies.

Bibliography

- [1] Understand Your Data: The Ultimate Guide to Data Lineage. [online], [cit. 2020-10-03]. Available from: <https://getmanta.com/ultimate-guide-to-data-lineage/>
- [2] Cooper, K. D.; Torczon, L. *Engineering a compiler*. Amsterdam [Etc.] Morgan Kaufmann, second edition, 2013.
- [3] Aho, A. V.; Lam, M. S.; et al. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, second edition, 2007.
- [4] Supported technologies. [online], [cit. 2020-10-03]. Available from: <https://getmanta.com/technologies/databases/>
- [5] A Deep Dive Into Google BigQuery Architecture. [online], [cit. 2020-10-03]. Available from: <https://panoply.io/data-warehouse-guide/bigquery-architecture/>
- [6] Lakshmanan, V.; Tigani, J. *Google BigQuery: the Definitive Guide: Data Warehousing, Analytics, and Machine Learning at Scale*. O'Reilly Media, Incorporated, 2019, ISBN 9781492044468. Available from: <https://books.google.cz/books?id=LRF7xgEACAAJ>
- [7] BigQuery. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery>
- [8] Geewax, J. *Google Cloud Platform in Action*. Manning Publications, 2018, ISBN 9781617293528. Available from: <https://books.google.cz/books?id=N7YVvgAACAAJ>
- [9] Melnik, S.; Gubarev, A.; et al. Dremel: Interactive Analysis of Web-Scale Datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, 2010, pp. 330–339. Available from: <http://www.vldb2010.org/accept.htm>

BIBLIOGRAPHY

- [10] Verma, A.; Pedrosa, L.; et al. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.
- [11] Corbett, J. C.; Dean, J.; et al. Spanner: Google’s Globally-Distributed Database. In *OSDI*, 2012.
- [12] Singh, A.; Ong, J.; et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network. In *Sigcomm ’15*, 2015.
- [13] Cloud SQL federated queries. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/docs/cloud-sql-federated-queries>
- [14] Introduction to BigQuery jobs. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/docs/jobs-overview>
- [15] Introduction to BigQuery INFORMATION SCHEMA. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/docs/information-schema-intro>
- [16] BigQuery APIs and Libraries Overview. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/docs/reference/libraries-overview>
- [17] Identity and Access Management overview. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/iam/docs/overview>
- [18] Resource hierarchy. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/resource-manager/docs/cloud-platform-resource-hierarchy>
- [19] Introduction to external data sources. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/external-data-sources>
- [20] Introduction to views. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/docs/views-intro>
- [21] Introduction to materialized views. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/docs/materialized-views-intro>
- [22] Release notes. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/docs/release-notes>

- [23] Switching SQL dialects. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/docs/reference/standard-sql/enabling-standard-sql>
- [24] Lexical structure and syntax in Standard SQL. [online], [cit. 2020-10-03]. Available from: <https://cloud.google.com/bigquery/docs/reference/standard-sql/lexical>
- [25] AT — BEFORE. [online], [cit. 2020-10-03]. Available from: <https://docs.snowflake.com/en/sql-reference/constructs/at-before.html>
- [26] Gosling, J.; Joy, B.; et al. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, first edition, 2014, ISBN 013390069X.
- [27] Overview of Spring Framework. [online], [cit. 2020-10-03]. Available from: <https://docs.spring.io/spring-framework/docs/5.0.0.RC2/spring-framework-reference/overview.html>
- [28] ANTLR v3. [online], [cit. 2020-10-03]. Available from: <https://www.antlr3.org/>
- [29] Osherove, R. *The Art of Unit Testing*. Manning Publications, second edition, 2013, ISBN 9781617290893.
- [30] Using OAuth 2.0 to Access Google APIs. [online], [cit. 2020-10-03]. Available from: <https://developers.google.com/identity/protocols/oauth2>

Acronyms

EBNF	Extended Backus–Naur Form
AST	Abstract Syntax Tree
SQL	Structured Query Language
ANSI	American National Standards Institute
ODBC	Open Database Connectivity
JDBC	Java Database Connectivity
UI	User Interface
COVID-19	Coronavirus disease 2019
GCP	Google Cloud Platform
NoSQL	non-SQL
ETL	Extract Transform Load
API	Application Programming Interface
REST	Representational State Transfer
SDK	Software Development Kit
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
WKB	Well-Known Binary

A. ACRONYMS

JSON JavaScript Object Notation

FR Functional Requirement

NFR Non-Functional Requirement

JWT JSON Web Token

KMS Key Management Service

DML Data Manipulation Language

DDL Data Definition Language

Contents of enclosed USB

readme.txt	the file with USB contents description
src ...	the directory containing source codes of the implemented modules
├─ manta-connector-bigquery-aggregation	
│ └─ manta-connector-bigquery-dictionary	
│ └─ manta-connector-bigquery-dictionary-extractor	
│ └─ manta-connector-bigquery-dictionary-mapping	
│ └─ manta-connector-bigquery-model	
│ └─ manta-connector-bigquery-resolver	
│ └─ manta-connector-bigquery-testutils	
└─ manta-dataflow-generator-bigquery-aggregation	
└─ manta-dataflow-generator-bigquery	
text	the thesis text directory
├─ thesis.pdf	the thesis text in PDF format
└─ thesis	the directory of L ^A T _E X source codes of the thesis