FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Metadata extraction, parsing, and dataflow detection in Snowflake sql dialect
Student:	Bc. Marek Tornóci
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 summer semester 2020/21

Instructions

Study the syntax and semantics of the Snowflake SQL dialect and the metadata of the Snowflake database engine.
Learn about the Manta project, how it represents database engine metadata, how it analyzes and represents similar SQL dialects, and how it represents data flows.
Suggest a way to represent Snowflake database engine metadata and a way to analyze and represent Snowflake source code for later analysis of data flows.
Suggest a way to detect data flows between Snowflake data structures by analyzing source codes of this SQL dialect.
Implement a prototype tool that extracts metadata from the Snowflake database engine and which extracts data flows from a set of files in the Snowflake SQL dialect to the Manta system.

References

Will be provided by the supervisor.

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

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

Prague February 11, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Metadata extraction, parsing, and dataflow detection in Snowflake SQL dialect

Bc. Marek Tornóci

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

July 30, 2020

Acknowledgements

First of all, I would like to thank the supervisor of this work, Ing. Jan Trávníček, Ph.D. for his help, time, and valuable advice during the work. I would also like to thank all the members of Manta, namely Mgr. Jiří Toušek for his help with understanding their software. My appreciation also goes towards my family and friends for their support during my studies.

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 July 30, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Tornóci, Marek. *Metadata extraction, parsing, and dataflow detection in Snowflake SQL dialect.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Práca sa zaoberá analýzou dátových tokov SQL dialektu Snowflake a možnosťami ich reprezentácie. Práca najprv skúma spôsob akým je potrebné analyzovať zdrojové SQL skripty, ich reprezentáciu a vizualizáciu dátových tokov pomocou systému Manta. Práca sa ďalej zaoberá skúmaním databázových objektov, ich metadát potrebných k extrakcií, spôsobom akým je možné tieto objekty extrahovať a analyzuje samotný SQL dialekt Snowflake. Na základe tejto analýzy vznikne návrh prototypu riešenia a jeho implementácia pokrytá testami, ktoré overujú jeho funkčnosť.

Klíčová slova Databáza Snowflake, dialekt SQL jazyka Snowflake, analýza dátových tokov, Manta, parsovanie, extrakcia

Abstract

The thesis deals with the analysis of data flows in the Snowflake SQL dialect and their possible representations. The work first examines how to analyze source codes, their representations, and the visualization of its data flows via the Manta system. The work continues to describe Snowflake database objects, their metadata needed to extract, how the metadata can be extracted, and analyzes the Snowflake SQL dialect. Based on this analysis, the design of a prototype solution and its implementation is created. The implemented prototype is covered by tests verifying its functionality.

Keywords Snowflake database, Snowflake SQL dialect, data flow analysis, Manta, parsing, extraction

Contents

List of Figures

List of Tables

Introduction

Nowadays, data are essential, and there is still more and more data which big organizations generate, store, and use in their favor, which in many times ensure their existence itself. These data need to be appropriately managed by the organizations.

To manage their data, companies often use various software tools for data storage and data analysis, such as databases, BI, and ETL tools. The tools that the organizations use are interconnected, create a significant infrastructure with many complex use cases. Because of this complex interconnection among the tools, there is a problem of understanding and keeping a good overview of their data. Companies need to know what is happening with data and how is data moved across systems over time, which we also call a data lineage. Based on knowing this information, it is possible to do complex data migrations or optimizations when needed.

Understanding of data can be achieved with the appropriate software solution that is able to analyze many technologies and provide a clear visualization of the data lineage. The main advantage over written documentation is that it is much more accurate, effective, and easier than reading documentation written by many people, containing many mistakes that arose over time.

One such software tool is Manta. Manta can analyze various technologies from which it can produce and visualize the data lineage and help companies understand their data. This thesis aims to extend Manta by adding a new module that can analyze another trending technology called Snowflake, a cloud relational database.

Goal of the thesis

This thesis aims to implement a prototype module for Manta, which can extract metadata from a Snowflake database, then use the extracted metadata during static analysis of SQL scripts in the Snowflake dialect and produce a data flow graph representing the data lineage. The prototype must be able to analyze the following Snowflake statements:

- SELECT
- INSERT
- UPDATE
- DELETE
- MERGE

In the first chapter, theoretical concepts are explained needed to comprehend to be able to produce a data flow graph from Snowflake SQL scripts.

The second chapter analyzes the Snowflake database, its metadata, and its SQL dialect.

The third chapter is about communication and design of implemented modules that extract, analyze, and produce data flow graphs.

In the fourth chapter, the modules from the design chapter are explained from an implementation point of view.

The fifth chapter describes specific testing methods of each implemented module.

The last chapter shows and explains the output samples.

Chapter 1

Theoretical background

This chapter explains the theoretical concepts necessary to understand how to produce a data flow graph from a script containing an arbitrary SQL code written in the Snowflake dialect. At the end of the chapter, the Manta tool is introduced as well.

1.1 Static code analysis

According to [1], a static code analysis is an analysis of a program without executing the program itself. Nowadays, many people use it without even realizing it from an integrated development environment, also known as IDE, where it is used to detect errors that would prevent the program from being executed. In many cases, it is even smart enough to suggest some code improvements or optimizations which may be done to improve the overall performance or readability of the code.

Another known usage is in the compilers' world, whereby a static code analysis, an intermediate representation of the input program, is created, which is easy to work with, and many code optimizations and algorithms can be easily implemented there.

Both cases are very similar to the static code analysis described in this thesis. It is needed to create an intermediate representation of any Snowflake SQL script from which it is relatively easy to detect data flows and produce a data flow graph. The following three known analysis need to be performed in order to build a meaningful intermediate representation.

1. Lexical analysis
2. Syntactic analysis
3. Semantic analysis

The analyses are performed in a pipeline fashion. The first performed analysis is lexical analysis, which receives a sequence of characters and produces an input for syntactic analysis, which then produces input for semantic analysis. The output of the semantic analysis is a meaningful intermediate representation for building a data flow graph.

This process is illustrated in the following figure.

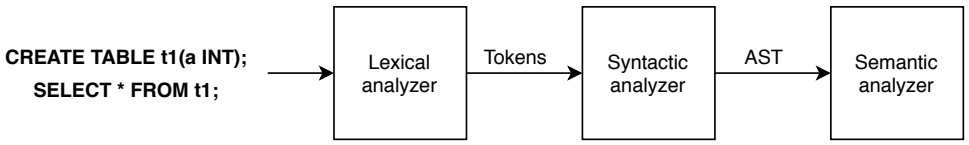


Figure 1.1: Flow diagram of static code analysis

1.1.1 Lexical analysis

The first phase of input processing is lexical analysis, which scans the input character stream and recognizes all possible tokens, which are substrings of consecutive characters that belong together logically.[2]

Tokens are recognized by rules that contain the regular expressions which we define to recognize patterns in the input.

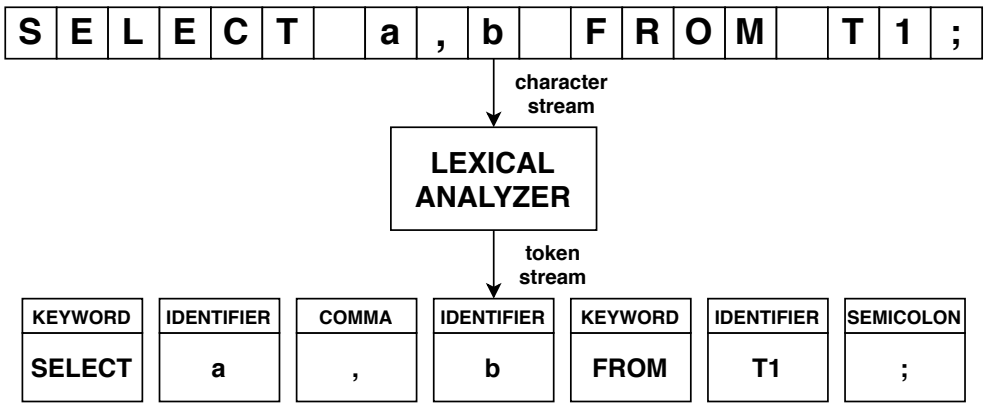


Figure 1.2: Lexical analysis

The figure 1.2 illustrates the recognition of tokens from the input character stream, which consists of a simple `SELECT` statement. Whitespaces can be part of token sequences, but they do not contain any syntactic information. For this reason, we usually drop them.

1.1.2 Syntactic analysis

"*The second phase of the compiler is a syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.*"[3]

A typical intermediate representation is called an abstract syntax tree, also known as AST. The abstract syntax tree does not represent every piece of information in the input program, but rather its logical and structural units. It has the essential structure of the parse tree that represents the whole input program, but many times eliminates a lot of internal nodes.[4]

The following figure shows an example of the AST that represents a simple `SELECT` statement.

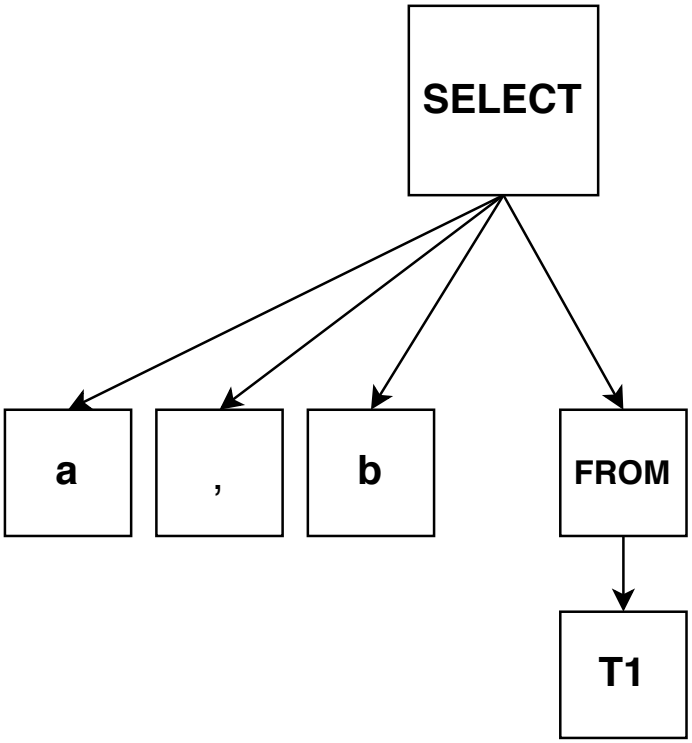


Figure 1.3: AST of the basic SELECT statement

In the figure 1.3, we can observe that some parts, such as whitespaces or a semicolon, are omitted because, in this concrete case, they are not important for the logical structure of the statement. Even though we might omit the less important parts in the input code, the AST often does not have the logical structure that is easy enough to work with for further processing.

In such a case, we want to make the AST more abstract and expressive. We can also model our AST to a certain degree by adding so-called imaginary

nodes. The imaginary nodes represent pseudo-operations, making the tree both, more abstract for our better understanding, and much more suitable for further post-processing.

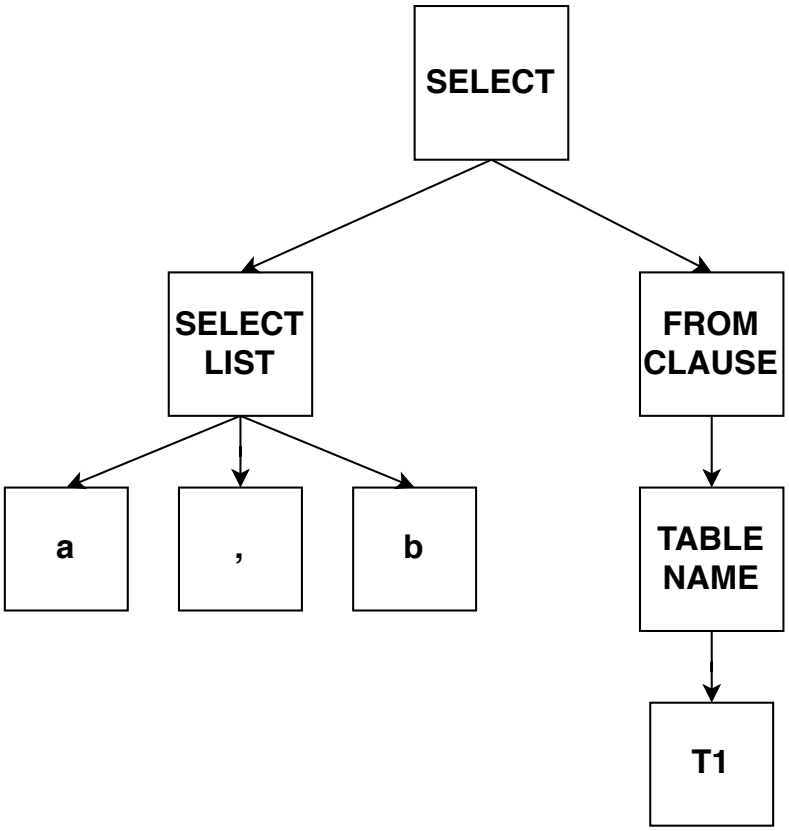


Figure 1.4: AST example of a basic `SELECT` statement with added imaginary nodes.

This technique of creating virtual nodes is also used in this thesis because using nodes that consist only from real tokens would make the tree hard to process during semantic analysis.

1.1.3 Semantic analysis

The last phase is a semantic analysis that discovers the meaning of a program. The semantic analysis recognizes when multiple occurrences of the same identifier are meant to refer to the same program entity and ensures that the uses are consistent. The analyzer typically builds and maintains a data structure called a symbol table, which maps each identifier with the known information about it.[5]

In this thesis, a semantic analysis is also called resolving, and the symbol table for keeping program entities is called a data dictionary. The data dic-

tionary is a special internal structure that keeps all unique program entities to which the program identifiers are recognized during AST processing with the addition of hierarchical relations among them the same way the database does. In other words, the dictionary structure represents the hierarchy of objects inside the database (for example, tables are inside schemas, schemas are inside databases, etc.).

The result of this phase is the resolved AST with all its identifiers referring to unique entities representing the declarations.

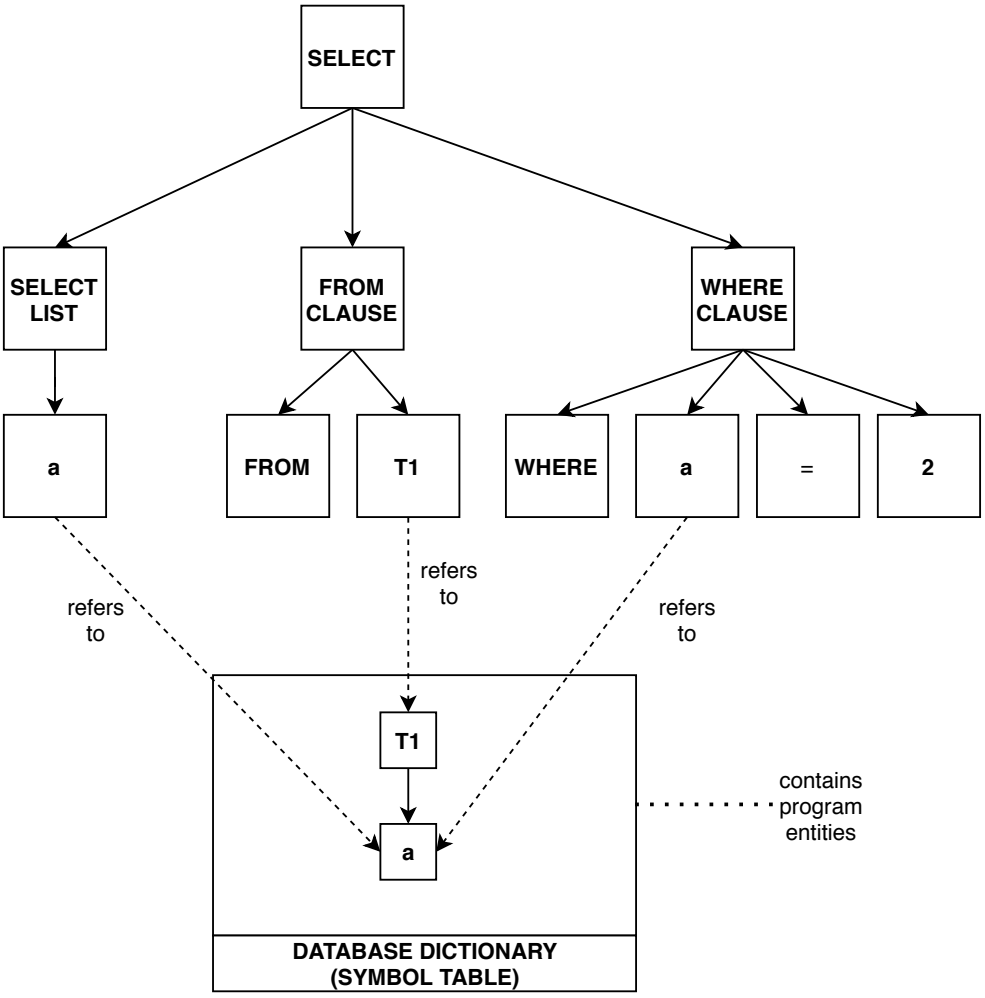


Figure 1.5: Example of recognized identifiers after semantic analysis

In the figure 1.5, we can see that each identifier in the AST of a `SELECT` statement refers to its actual unique declaration. Table identifier `t1` refers to a table entity `t1`, and both identifiers `a` refer to the same column entity `a`. We can also see a parent-child relation inside the dictionary between entities.

1.2 Metadata Extraction

Extraction is the process of obtaining metadata from software systems such as databases. The metadata is extracted to a database dictionary before processing the script. They contain information describing all database objects created inside a database. The reason for the metadata extraction is to produce the most accurate data flow graph. Even though we may accurately deduce many things during the semantic analysis, sometimes it is not possible. Suppose we have a simple `SELECT` statement that reads data from a table without knowing a DDL definition of the table.

```
1 SELECT * from t1;
```

In this case, we would not know how to resolve an `ASTERISK(*)` during the semantic analysis, if we did not have the metadata about the table `t1` containing information about its columns.

Another example would be functions and views where we need to extract definitions of them.

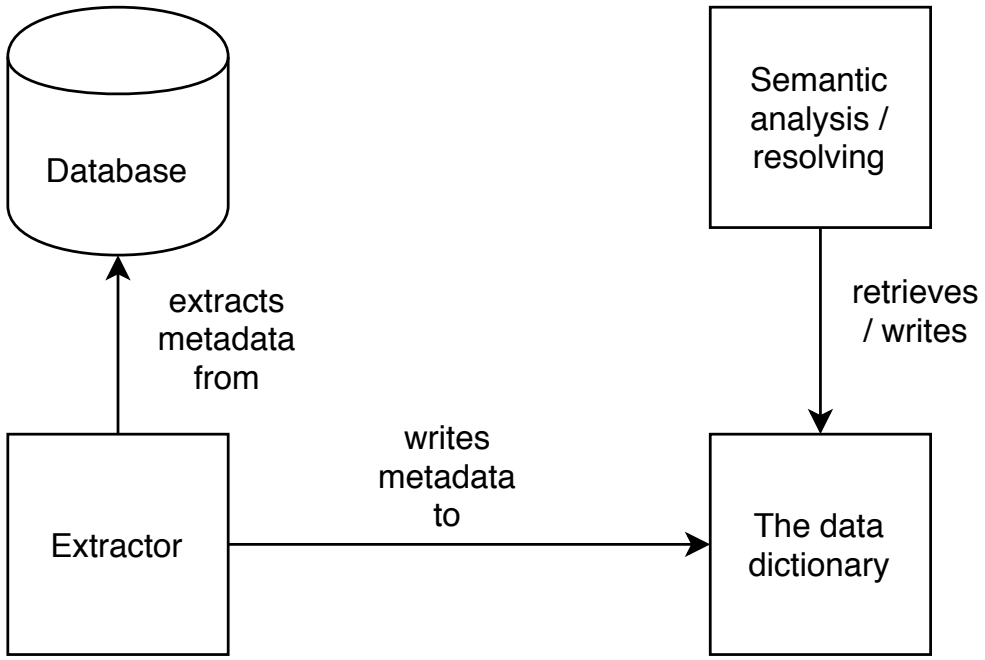


Figure 1.6: Metadata extraction diagram

1.3 Dataflow graph

Data lineage describes the data origin, what transformations they went through, and its movements over time. It helps many big organizations to better understand the movement of their data. They need to know where their data come from as well as when and where the data separate or merge with other data.[6]

Although there are several ways of representing data lineage, a graph representation is considered the best for its clarity. The graph representation is called a data flow graph. It is an oriented graph that consists of nodes and oriented edges. The nodes represent the actual data. Two nodes can be connected by the oriented edges representing data flows between them. There exist two types of data flow edges:

- Direct flows
- Filter flows

Direct data flows indicate that the source nodes directly participate in the data origin of target nodes. The example of this is a `SELECT` statement because the data from the source table from where the statement reads data directly appear in the result of the `SELECT` operation.

Filter data flows affect the content of the target node without directly contributing to it. The example of filter flows is a `WHERE` clause of the `SELECT` statement which only restricts the value set of the `SELECT` operation.

1.4 Manta Flow

Manta is a software solution for complex visualization of data lineage in the Business Intelligence environment. For Manta, it is essential to support analyzing a more extensive range of trending technologies among the organizations so they can cover and see the whole data lineage of their infrastructure. Currently, the following technologies are supported: [7]

- StreamSets
- IBM DataStage
- Informatica Power Center
- Microsoft SQL Server Integration Services
- Oracle Data Integrator
- IBM DataStage
- Sqoop
- Pig
- Talend
- Cobol
- Java
- Teradata database
- Sysbase

- Hive
- Neteeza
- Microsoft SQL Server database
- Microsoft Azure SQL database
- IBM DB2
- Microsoft Azure SQL Data Warehouse
- PostgreSQL
- RedShift
- Greenplun
- Oracle
- Microsoft SSAS, SSRS, SSIS
- Microsoft APS
- Microsoft Excel
- Sas
- Cognos
- Sap Business Objects
- Tableu
- PowerBI
- Oracle Business Intelligence Enterprise Edition

The following figure 1.7 shows a dataflow graph visualized in Manta.

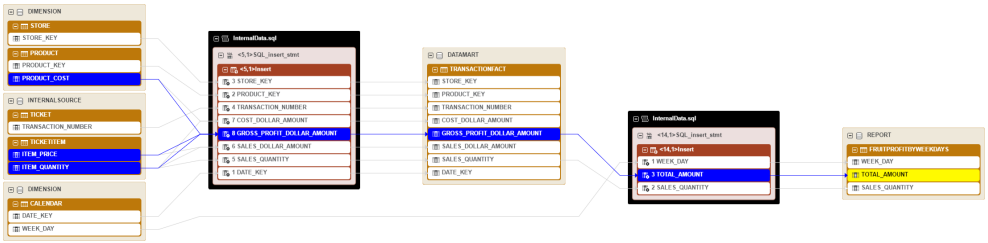


Figure 1.7: Visualization of a data flow graph with Manta [8]

Chapter 2

Analysis

This chapter describes a Snowflake technology, describes what kind of metadata about Snowflake's objects we need to extract, how exactly are the metadata extracted, and in the end describes the Snowflake SQL dialect.

2.1 Snowflake

Snowflake is an analytic data-warehouse platform. It provides a data warehouse technology that is faster, more flexible, and easier to use than most of the current data warehouse solutions. Snowflake data warehouse uses a new SQL database engine with a unique architecture designed for the cloud. It is provided as a Software-as-a-Service (SaaS) and runs completely on cloud offering users the following big advantages: [9]

- No hardware (virtual or physical) and software for users to install, manage or configure.
- All ongoing maintenance management is handled by Snowflake.

All components run in public cloud infrastructure that uses virtual computing instances, which users can scale according to their needs in computational power. Snowflake is a multi-cloud solution that supports multiple cloud services such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform. It is available globally and supports multiple regions across all mentioned cloud platforms. The regions determine where our data are stored geographically. For connecting to a Snowflake instance, we need to obtain Snowflake's root account representing the database instance. This account is used for connecting to the instance through multiple interfaces provided by Snowflake.

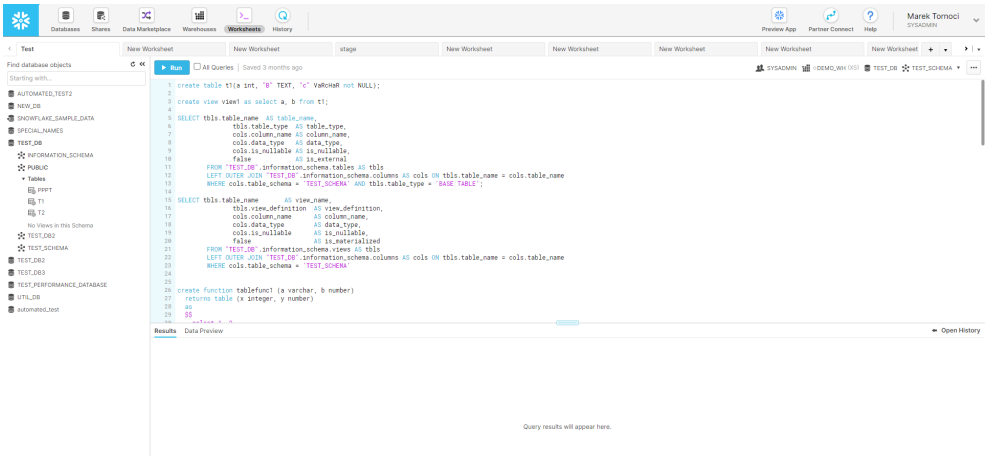


Figure 2.1: Snowflake user web interface

In the figure 2.1 we can see the web user interface that users normally use to manage their Snowflake instance.

2.1.1 Architecture

"*Snowflake's architecture is a hybrid of traditional shared-disk database architectures and shared-nothing database architectures. Similar to shared-disk architectures, Snowflake uses a central data repository for persisted data that is accessible from all compute nodes in the data warehouse. But similar to shared-nothing architectures, Snowflake processes queries using MPP.*"[9]

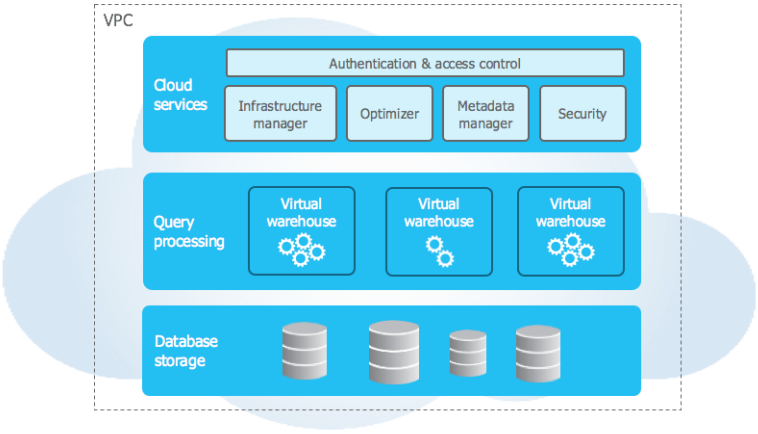
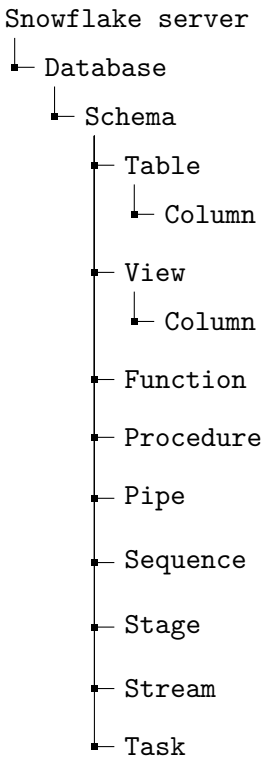


Figure 2.2: Figure showing overview of Snowflake architecture [9]

In the figure 2.2, we can see architecture layers where each layer is responsible for different tasks. The purpose of a cloud service layer is to manage all of Snowflake's activities, such as authentication, query parsing, and optimizations. Snowflake performs query processing in a processing layer by using virtual warehouses. Each warehouse is a virtual MPP cluster composed of compute nodes allocated by Snowflake from the cloud provider. A database storage layer stores loaded data in a unique, optimized, and compressed columnar format.[9]

2.1.2 Snowflake objects and its structure

Snowflake is similar to all known SQL databases containing many objects such as schemas, tables, views, and many more. In the following diagram, we can see a structural hierarchy of Snowflake's objects.



At the top of the hierarchy exists a Snowflake server corresponding to a unique Snowflake account. Inside the Snowflake server, we can create an unlimited amount of databases. The database is a logical grouping of schemas where Snowflake maintains all of its data. Each schema belongs to a single database. Schemas are logical groupings containing all of Snowflake's database objects.

2.1.3 Metadata access

We can access metadata of Snowflake's database objects through a Snowflake schema called `INFORMATION_SCHEMA`. The schema resides inside each database and contains set of system-defined views providing metadata about Snowflake's database objects. It is based on the SQL-92 ANSI Information Schema but with the addition of views that are specific for Snowflake. [10]

`INFORMATION_SCHEMA` contains the following list of useful views needed during the extraction of database objects:

- `TABLES` view
- `COLUMNS` view
- `EXTERNAL_TABLES` view
- `FUNCTIONS` view
- `PROCEDURES` view
- `SCHEMATA` view
- `SEQUENCES` view
- `STAGES` view
- `SCHEMATA` view

2.2 Metadata extraction of Snowflake's objects

Section 1.2 explains the meaning behind the need of metadata extraction in general. This chapter, on the other hand, aims to explain what kind of objects we need to extract from Snowflake and the possible ways of extracting them.

2.2.1 Possible ways of metadata extraction

There exist two ways of extracting metadata from databases in general. The first way is to extract metadata information from defined views or catalogs provided by a database engine. However, some databases do not have any views or functions providing these metadata. In such a case, there is a second way of obtaining objects metadata by extracting all DDL scripts of needed database objects. After the extraction, the DDL scripts are parsed and analyzed where during semantic analysis, all declarations are created in the database dictionary. This way of extracting metadata is much more complicated and time-consuming. As described in 2.1.3, Snowflake supports the concept of `INFORMATION_SCHEMA`, so there is no reason for us to extract all DDL scripts.

2.2.2 Description of Snowflake database objects

This section describes all Snowflake objects needed for extraction to obtain their metadata.

2.2.2.1 Databases

Snowflake creates `INFORMATION_SCHEMA` implicitly inside each database, so it would not make sense for this schema to contain a view describing all databases inside a Snowflake server instance globally. For this reason, Snowflake provides a SQL statement called `SHOW` that can list all databases inside the Snowflake server instance.

Each database in Snowflake also contains an implicit schema called `PUBLIC`. This schema is created along with the database. `PUBLIC` is a default name for the schema set by a session internal property that can be changed.

2.2.2.2 Schemas

To obtain all schemas from a database, we can use a view called `SCHEMATA` residing in `INFORMATION_SCHEMA`. The view contains a column called `SCHEMA_NAME` containing names of all schemas. The only thing of which we have to be careful about is not to extract `INFORMATION_SCHEMA` because we do not need to extract this schema.

2.2.2.3 Tables

Tables in Snowflake are of 2 types:

- Tables
- External tables

Tables are regular tables as we know them from other SQL dialects. External tables are a bit different because each of them contains a URI specifying an external stage. In Snowflake, whenever we query external tables, they read data from a set of one or more files from the specified external stage and output it in a single `VARIANT` column. The metadata extraction of both external and regular tables is slightly different.

Regular tables are extracted from the `TABLES` view containing three relevant columns - `TABLE_NAME`, `SCHEMA_NAME`, and `TABLE_TYPE`. The column `TABLE_NAME` is used for a join with the `COLUMNS` view to obtain its columns. `TABLES` also contains all user-created materialized views, so the value of the column `TABLE_TYPE` is used to distinguish between tables and materialized views. The last column `SCHEMA_NAME`, is used to obtain tables only from a specified schema.

External tables are extracted from the `EXTERNAL_TABLES` view containing only external tables. This view also contains three essential columns - `TABLE_NAME`, `SCHEMA_NAME`, and `LOCATION`. The first two columns have the same purpose as with regular tables. The third column contains information about the mentioned external stage URI that we need to obtain.

2.2.2.4 Stages

Stage is a Snowflake specific object for storing files. There exists four types of stages in Snowflake:

- External stage
- Internal named Stage
- Table stage
- User stage

The stored files inside Snowflake stages are possible to query and process. The difference between external stages and the others is that in external stages, files are stored outside Snowflake in the cloud. External stages contains a URI identifying a location of the stored files. We only need to obtain metadata about the external and the internal named stages. To obtain its metadata we use `STAGES` containing all the needed information inside the following columns:

- `STAGE_NAME` - contains stage name
- `STAGE_URL` - contains a URI of an external stage
- `STAGE_TYPE` - describes a stage type

2.2.2.5 Functions

Snowflake supports classic functions similar to functions we know from other databases. Functions in Snowflake can have zero or multiple input parameters and return either single scalar value or a table. Based on the return type Snowflake distinguish between 2 types of functions.

- User defined function (UDF)
- User defined table function (UDTF)

The function body can contain either SQL or JavaScript code. Snowflake supports function overloading by data types. Multiple functions in the same schema can have the same name as long as they differ in number or data type of their parameters. To extract a function, we use the `Functions` view. The view contains the following four important columns:

- `FUNCTION_NAME` - contains a function name
- `ARGUMENT_SIGNATURE` - contains an argument signature of a function
- `DATA_TYPE` - contains a return type of a function. If the function is UDTF then the column contains table return type signature in the form of `TABLE (ARG_NAME_1 ARG_TYPE_1, ..., ARG_NAME_N ARG_TYPE_N)`
- `VOLATILITY` - contains an information about function's volatility

The problem with signatures stored inside `ARGUMENT_SIGNATURE` and `DATA_TYPE` is that their argument names are not enclosed by double-quotes. This can make argument signatures nondeterministic in special cases if their names contain special characters. Suppose we create a function with a single column called `"ARG1 NUMBER, ARG2 " NUMBER` (perfectly valid according to 2.3.1). The following source code 2.1 shows an example of such a function.

```
1 create or replace function f1("ARG1 NUMBER, ARG2 " NUMBER)
2  returns NUMBER
3  as
4  $$
5      1
6  $$;
```

Source Code 2.1: Example of a function with argument name containing special characters

An argument signature of a function `f1` stored inside the `ARGUMENT_SIGNATURE` has the value `(ARG1 NUMBER, ARG2 NUMBER)`. Because the argument names are not double-quoted, we cannot decide how many arguments the function actually contains.

2.2.2.6 Stored procedures

Stored procedures in Snowflake contain Java Script code executing SQL statements. Stored procedures can be executed many times by calling them with a statement named `CALL`. The `PROCEDURE` view is used for extracting its metadata.

2.2.2.7 Views

Extraction of views depends on their type. In Snowflake, there are two type of views:

- Views
- Materialized views

Regular views are extracted from `VIEW` containing important columns such as `VIEW_NAME`, `VIEW_DEFINITION`. The first column is used for a join with the `COLUMNS` to obtain its columns. The second column contains view definitions needed to extract as well.

Materialized views, on the other hand, are extracted from the `TABLES` view, where we need a way to distinguish between them and tables. As mentioned earlier, the column `TABLE_TYPE` exists for this purpose.

2.3 Snowflake's SQL dialect

Snowflake dialect supports standard SQL similar to PostgreSQL dialect supporting a subset of ANSI SQL: 1999 and SQL: 2003 with the addition of Snowflake specific clauses and constructs. This section aims to explain only the subset of Snowflake SQL dialect, mainly constructs implemented in the prototype or Snowflake's unique constructs and concepts. The section does not explain all known SQL constructs in detail because it is reckoned that the reader is familiar with the basics of the standard SQL.

2.3.1 Snowflake object identifier

This section aims to explain rules for naming Snowflake objects. In Snowflake, identifiers can be either quoted or unquoted. Unquoted object identifiers must fulfill the following rules:[11]

- Starts with a letter [A-Z, a-z] or underscore (_).
- Must contain only letters, underscores, decimal digits [0-9], and dollar signs ($).
- Identifiers are case-insensitive.

If we put double quotes around identifier (for example, "My identifier"), the following 2 rules apply:[11]

- The identifier is case-sensitive.
- The identifier can contain and start with any ASCII character from the blank character (32) to the tilde (126), excluding the double quote (") character.

2.3.2 Snowflake data types

Snowflake supports only built-in data types. The following table shows all built-in data types with added notes.

Category	Type	Notes
Numeric Data Types	NUMBER	Default precision and scale are (38,0).
	DECIMAL	Synonymous with NUMBER.
	NUMERIC	Synonymous with NUMBER.
	INT, INTEGER, BIGINT, SMALLINT	Synonymous with NUMBER except precision and scale cannot be specified.
	FLOAT, FLOAT4, FLOAT8	
	DOUBLE, DOUBLE PRECISION	Synonymous with FLOAT.
	REAL	Synonymous with FLOAT.
String & Binary Data Types	VARCHAR	Default (and maximum) is 16,777,216 bytes.
	CHAR, CHARACTER	Synonymous with VARCHAR except default length is VARCHAR(1).
	STRING	Synonymous with VARCHAR.
	TEXT	Synonymous with VARCHAR.
	BINARY	
	VARBINARY	Synonymous with BINARY.
Logical Data Types	BOOLEAN	
Date & Time Data Types	DATE	
	DATETIME	Alias for TIMESTAMP_NTZ
	TIME	
	TIMESTAMP	Alias for one of the TIMESTAMP variations (TIMESTAMP_NTZ by default).
	TIMESTAMP_LTZ	TIMESTAMP with local time zone.
	TIMESTAMP_NTZ	TIMESTAMP with no time zone.
	TIMESTAMP_TZ	TIMESTAMP with time zone.
Semi-structured Data Types	VARIANT	
	OBJECT	
	ARRAY	

Table 2.1: Table showing supported data types in Snowflake [12]

In the table 2.1, we can see all of Snowflake's supported data types. The most interesting ones are semi-structured data types - VARIANT, OBJECT, and ARRAY.

`VARIANT` is a generic data type that can store any value of any other data type in Snowflake. A value of any Snowflake data type can be explicitly or implicitly converted to the `VARIANT` data type.

`OBJECT` is a data type representing a collection of key-value pairs. Non-empty strings are used as keys, and the values are of the `VARIANT` data type.

`ARRAY` is a data type representing sparse arrays of arbitrary size. Indexes are a non-negative integers, and values are of the VARIANT data type.

2.3.3 Non reserved and reserved words

Snowflake, like other programming languages, uses a set of keywords. Some of them are reserved by Snowflake, which means that they cannot be used as identifiers. Snowflake reserves almost all ANSI keywords (with few exceptions) as well as some additional keywords.

Non-reserved keywords, on the other hand, can be used as identifiers. Snowflake contains some interesting non-reserved words that, in some situations, behave as reserved words. One of the interesting non-reserved words is `FULL`, which cannot be used as a table identifier or alias inside the `FROM` clause. Another interesting non-reserved word is `WHEN`, which cannot be used as a column reference in a scalar expression.

In [13], Snowflake describes and categorizes all reserved keywords with the addition of keywords that behave as reserved in special situations.

2.3.4 Resolution of unqualified identifiers

This section aims to explain how Snowflake resolves unqualified object names. Unqualified identifiers in Snowflake are resolved in two different ways depending whether they appear in DML, DDL statements or in `SELECT` queries.[14]

2.3.4.1 Current database, current schema

By connecting to a Snowflake server, we establish a session. The session contains many session settings. Among the many settings it contains, there are two important for object name resolution - a current database and current schema. The current schema always belongs to the current database. These properties must be set in order to use unqualified or partially-qualified identifiers in queries.

2.3.4.2 DDL and DML

All unqualified identifiers present in DDL or DML statements are always resolved within the current schema.[14]

2.3.4.3 Queries

Unqualified names in queries are resolved through a search path. The search path contains a list of fully- or partially-qualified valid schema names. The search path is a session parameter that can be changed by the `ALTER SESSION` statement. The search path contains two pseudo-variables called `$current` and `$public`. The first variable represents the current schema. The second variable specifies the public schema (see 2.2.2.1) of the current database. The value of the search path is reinterpreted every time it is used. Therefore changing the current schema changes the meaning of `$current` and changing the default database changes the meaning of both `$current` and `$public`.[14]

2.3.4.4 Function and view definitions

The search path is not used for resolving unqualified identifiers inside view definitions and function body definitions. All unqualified identifiers in a view or function definition are resolved in the view's or function's schema only (in the schema where they reside).[14]

2.3.5 SELECT statement

A `SELECT` statement is used to query tables and retrieve a set of rows. Supported clauses and behavior of the statement is very similar to `SELECT` statements of other SQL dialects, but there are some differences. The following list contains all supported `SELECT` clauses:

- WITH
- TOP
- FROM
- AT|BEFORE
- CONNECT BY
- JOIN
- PIVOT|UNPIVOT
- VALUES
- SAMPLE
- WHERE
- GROUP BY
- HAVING
- ORDER BY
- LIMIT
- QUALIFY

2.3.5.1 AT|BEFORE clause

Snowflake supports the concept called Time Travel, meaning we can access historical data (changed or deleted) at any point within a defined period. This clause is used for Time Travel to query historical data for the specified object. It is specified in the `FROM` clause immediately after the table name. [15]

In the code example 2.2, we are querying historical data from a `customers` table using the exact date by a specified timestamp.

```
SELECT * FROM customers
 AT(TIMESTAMP => 'Mon, 22 July 2020 13:37:00 -0700'::TIMESTAMP);
```

Source Code 2.2: Example of querying a historical data in Snowflake

2.3.5.2 WITH clause (CTE)

`WITH` is an optional clause that precedes body of a `SELECT` statement and defines one or more CTEs that can be referenced later in the `FROM` clause of the statement. Snowflake supports both recursive, and non-recursive CTEs. The clause is very similar to the `WITH` clause of other SQL dialects. However, there is difference at what all possible places we can define the clause.

In Snowflake, each `SELECT` unit can have its own `WITH` clause. This, for example, allows using multiple `WITH` clauses with the use of set operators.

```
(WITH
alias_t1 AS (SELECT * FROM t1)
SELECT * FROM alias_t1)

UNION

(WITH
alias_t2 AS (SELECT * FROM t2)
SELECT * FROM alias_t2)
```

Source Code 2.3: Example of `WITH` clause with the set operation

In the source code example 2.3, we can see that each `SELECT` unit has its own defined `WITH` clause. The only condition is that each `SELECT` unit must be enclosed by parenthesis. The scope of such `WITH` clause is restricted only for its enclosed `SELECT` unit. This can be done recursively. Allowing each SELECT unit to have its own `WITH` clause means that we are not restricted from using it in a subquery as well.

```
WITH alias_t1 AS (SELECT * from t1)
SELECT * FROM (
    WITH alias_t2 as (SELECT * FROM t2)
    SELECT * FROM alias_t2
)
```

Source Code 2.4: Example of `WITH` clause in a subquery

The example 2.4 shows the usage of using the WITH clause in a subquery. The strange behaviour occurs when we give CTEs the same name. In such a case the subquery CTE reference always refers to the outer most defined CTE.

2.3.5.3 FROM clause

This section describes some specific behaviors inside the `FROM` clause that can differ from other SQL dialects like PostgreSQL or Oracle.

The first specific behavior is about table joins. In Snowflake, it is possible to join two tables without specifying conditional clauses `ON` or `USING`. The following source code 2.5 is perfectly valid.

```
1 SELECT * FROM t1 INNER JOIN t2;
```

Source Code 2.5: Example of inner join without the conditional `ON` clause

The result of such a query where we omit the conditional clauses is `CROSS JOIN`. If we omit the conditional clauses we can still specify the `WHERE` clause with some condition to get the same result.

Snowflake does not allow aliasing a group that consists of joined tables. The following source code 2.6 shows a query that does not compile because it tries to alias a group that contains a join between tables `t1` and `t2`.

```
1  SELECT * FROM
2  (t1 INNER JOIN t2 ON t1.a = t2.b) AS r INNER JOIN t3;
```

Source Code 2.6: Example of a join group alias

2.3.6 INSERT statement

Snowflake supports two types of insert statements.

1. Insert statement
2. Multi-table insert statement

The first mentioned statement is a basic insert statement that is very similar to all other SQL dialects. It updates a specified table by inserting one or multiple rows into the table. We can insert values to the table by specifying value rows inside the `VALUES` clause or using the `SELECT` statement.

The second multi-table insert statement is rather special and not all SQL dialects support such a statement. It updates multiple tables by inserting one or more rows with column values from a query into the tables. Snowflake supports two types of multi-table insert statements:

- **Unconditional multi-table insert statement** - Each row from a specified query executes each `INTO` clause inside the `INSERT` statement.
- **Conditional multi-table insert statement** - Specifies the condition that must evaluate to true in order to execute specified `INTO` clauses.

```
1 INSERT ALL
2    WHERE a > 100 THEN
3      INTO t1
4    WHEN n1 > 10 THEN
5      INTO t1
6      INTO t2
7    ELSE
8      INTO t2
9 SELECT a FROM src;
```

Source Code 2.7: Example of conditional multi-table insert statement

In the source code 2.7, we can see the example of the multi-table conditional `INSERT` statement. In order to execute `INSERT INTO` statements, the condition must evaluate to true otherwise an optional `ELSE` branch is executed. Instead of `ALL` keyword we can also use `FIRST` keyword that executes only the first conditional branch that evaluates to true and ignores all the following ones. If we want to truncate the target tables before inserting values, we can specify a keyword called `OVERWRITE` right after keyword `INSERT`.

2.3.7 MERGE statement

The statement is used to `INSERT`, `UPDATE` or `DELETE` in one table based on subquery or values matched from another table. It consists of the following clauses:

- **MERGE INTO <target>** - used to specify the target table

- **USING <source>** - used to specify the table or subquery to join with the target table
- **ON <expression>** - used to specify the expression on which the source and target table are joined
- **WHEN MATCHED [AND <conditional_expression>]** - contains either single `UPDATE` or `DELETE` statement that is executed when both the merge condition and optional conditional predicate evaluate to true
- **WHEN NOT MATCHED [AND <conditional_expression>]** - contains a single `INSERT` statement that is executed when both, the merge condition and the optional conditional predicate evaluate to true

```
1 merge into target
2   using src on target.id = src.id
3   when matched and src.v = 10 then delete
4   when matched then update set target.v = src.v;
5   when not matched then insert (id, v) values (src.id, src.v);
```

Source Code 2.8: Example of MERGE statement in Snowflake

The code example 2.8 shows basic usage of the `MERGE` statement covering all possible constructs.

2.3.8 UPDATE statement

This statement is used to update specified rows in the target table with new values. The `UPDATE` statement is very similar to the `UPDATE` statement of other known SQL dialects. The statement consists of the following clauses:

- **UPDATE <target>** - used to specify the target table
- **SET <column_list>** - used to specify new values and columns to update
- **FROM <sources>** - used to specify the source tables
- **WHERE <expression>** - conditional expression specifying which rows to update

`FROM` and `WHERE` clauses are optional.

```
1 UPDATE t1
2 SET t1.a = t1.a + t2.a, t1.b = t2.b, t1.c = 'NEW TEXT'
3 FROM t2
4 WHERE t1.a = t2.a AND t1.b < 10;
```

Source Code 2.9: Example of `UPDATE` statement in Snowflake

In the example, 2.9 we can see the statement updating the target table `t1` with values from table `t2` when the specified condition in the `WHERE` clause is evaluated to true.

2.3.9 DELETE statement

The statement is used to remove data from a table. The statement consists of the following logical constructs:

- **DELETE FROM <table_name>** - used to specify the table from which we want to remove rows
- **USING <additional_tables>** - used to specify new values and columns to update
- **WHERE <conditional_expression>** - used to specify the source tables

`USING` and `WHERE` clauses are optional and can be omitted. The following source code 2.10 shows an example of deleting rows from a table `t1` when the condition is evaluated to true.

```
1 DELETE FROM t1 USING t2 WHERE t1.a = t2.a
```

Source Code 2.10: Example of `DELETE` statement in Snowflake

2.3.10 Querying stages

One of the Snowflake's unique constructs is querying staged files. To query staged files, we have to specify a stage location inside the `FROM` clause of the `SELECT` statement. Stage location reference always starts with `@` and can have one of the following forms depending on the referenced stage type:

- **@[namespace.]external_stage_object_name[/path]**
- **@[namespace.]internal_named_stage_object_name[/path]**
- **@[namespace.]%table_stage[/path]**
- **@[namespace.]~[/path]**

Namespace and path are optional. If the path is not specified in the location URI or ends with a directory, all the files inside the specified stage or directory are queried.

```
1 SELECT $1, $4 FROM @my_int_stage1/a/b/c;
```

Source Code 2.11: Stage Query example

In the code example 2.11, we are selecting the first and fourth columns from the internal named stage called `my_int_stage1`. To use special characters, we can quote the location by single quotes.

2.3.10.1 Limitations

Querying staged files has its own limitations. One of such limitations is that in the `SELECT LIST` we can only reference file columns by positions. By default file columns are separated by commas, but this behaviour can be changed by specifying own file format.

Another quite big limitation is that in the `FROM` clause of any stage query, we can only specify one single stage location and nothing else, meaning we cannot use table joins or specify more tables in the stage query.

2.3.11 Querying semi-structured data

Snowflake supports SQL queries for accessing elements of semi-structured data using special operators or functions. The following list shows supported semi-structured data in Snowflake: [16]

- JSON
- Avro
- ORC
- Parquet

The semi-structured data must be stored in a single `VARIANT` column. To access its elements, we have to specify colon (`:`) behind the `VARIANT` column name.

```
+-----------------------------------+
| PERSON (VARIANT)                  |
|-----------------------------------/
| {                                 |
|    "name" : Marek Tornóci         |
|    "address" : {                  |
|        "Street" : Chaloupeckého 13 |
|    }                              |
| },                                |
| {                                 |
|    "name" : Tomáš Drietomský      |
|    "address" : {                  |
|        "Street" : Chorvátska 5    |
|    }                              |
| }                                 |
+-----------------------------------+

SELECT person:"address"."Street" FROM persons;

+---------------------+
| PERSON:ADDRESS.CITY  |
|---------------------/
| "Chaloupeckého 13"   |
| "Chorvátska 5"       |
+---------------------+
```

Source Code 2.12: Example of accessing JSON elements

The example 2.12 shows accessing the element `Street` from the JSON structure stored inside the `VARIANT` column named `PERSON`. There exist two ways of accessing elements of inside the data:

- Dot notation
- Bracket notation

The dot notation is used in the example 2.12, where element names in the reference are separated by dots.

Another way of referencing elements is to use bracket notation, where we put enclosed element names by single quotes to brackets. The following example 2.13 shows how to access JSON element called `Street` by using the bracket notation.

```
1 SELECT person['address']['Street'] FROM persons;
```

Source Code 2.13: Example of accessing JSON elements by the bracket notation

2.4 Functional Requirements

Functional requirements are descriptions of functionalities the software must offer. This section describes the functional requirements for the implemented prototype.

2.4.1 Extracting metadata from Snowflake database

The prototype must know how to extract metadata representing Snowflake's objects from a given database instance and fill up the database dictionary with them. Another required functionality of extractor should include extracting all required DDL statements of objects to the file system.

2.4.2 Parsing Snowflake SQL scripts

The prototype must be able to read a given file or a string containing SQL scripts written in the Snowflake SQL dialect and recognize its structure and parse the script.

2.4.3 Build AST

The prototype must build an AST during the parsing stage.

2.4.4 Semantic analysis

After the parser builds the abstract syntax tree, it is needed to process the tree and resolve all its name references to the unique declaration entities that reside inside the data dictionary. During the semantic analysis, the prototype must also fill up the dictionary with newly recognized declarations.

2.4.5 Dataflow graph

The prototype is required to process a resolved AST and build a data flow graph containing nodes representing declarations and operations connected by direct or filter edges.

2.5 Non Functional requirements

This section covers the non-functional requirements the implemented prototype must meet. Non-functional requirements do not describe the system functionalities but rather its general characteristics.

2.5.1 Use Manta classes

The prototype should use or extend implemented Manta classes resulting in significantly simpler code. Java classes representing AST nodes should extend the abstract class called `MantaAstNode`, containing many implemented methods for more straightforward navigation inside the tree.

2.5.2 Imaginary nodes

An AST is supposed to contain imaginary nodes (described in 1.1.2) to make the tree much more suitable for post-processing.

2.5.3 Execution time of the prototype

The prototype must process the scripts and build the data flow graph in a reasonable time. The term reasonable time means a time that is not significant for the user waiting while working with the prototype, usually in seconds.

2.5.4 Maintability and Extendability

The implemented prototype must meet the quality of a good software code covered by a reasonable amount of tests resulting in an easily maintainable and extendable software solution.

Chapter 3

Design

This chapter aims to explain what technologies are used for implementing the functional prototype as well as what kind of modules the prototype consists of.

3.1 Technologies

The prototype is implemented in Java with the help of the Spring framework to simplify the development. For parsing the Snowflake SQL dialect and creating an AST, we use ANTLR. MyBatis, along with the Snowflake JDBC driver, is used for metadata extraction from a Snowflake database server. The prototype uses Maven for managing dependencies among modules. The last significant framework concerns testing where JUnit is used.

3.1.1 Java

Java is one of the most popular programming languages in the world. It is an object-oriented language intended to let developers write once and run anywhere code. According to [17], Java is used everywhere, from laptops to data centers, game consoles to scientific supercomputers, and cell phones to the Internet. All modules in Manta are implemented in Java as well.

3.1.2 Spring

Spring is an open-source framework that provides support for developing Java applications. Spring helps developers create high performing applications using POJOs. The framework is considered to be a secure, low-cost, and flexible. Spring helps to improve coding efficiency and reduces overall application development time and complexity. The Spring ecosystem consists of about 20 modules. [18]

The prototype uses mainly the Spring Core module for creating many configurations and using dependency injections for system decoupling.

3.1.3 ANTLR

ANTLR is LL(*) parser generator we can use to implement language interpreters, compilers, and other translators. ANTLR is written in Java and is used to translate grammars specifying a language to executable Java code that recognizes the language. [19]

The latest ANTLR version is 4. The prototype uses ANTLR version 3 for generating both Lexer and Parser to build AST for further processing and generating a dataflow graph. The main reason for using an older version is that it supports creating a custom AST by defining rewrite rules in the parsing grammar. This functionality is not supported in ANTLR 4 as the tool can only produce a parse tree.

3.1.4 Maven

Maven is a tool used for building and managing Java-based projects. [20]. Maven provides a solution for managing dependencies to other modules, libraries, and frameworks. The prototype uses Maven, especially for managing dependencies to other external libraries or other Manta modules. Each module using Maven contains a configuration file named `pom.xml` in the root directory with defined dependencies and configurations unique for the module.

3.1.5 MyBatis

MyBatis is a persistence framework with support for SQL code, stored procedures, and advanced mappings that eliminates almost all of the manual JDBC code and manual settings of parameters. [21]

The implemented prototype uses MyBatis for extracting metadata from a given Snowflake server instance. It consists of configuration files in XML containing SQL queries for metadata extraction of Snowflake objects. MyBatis maps retrieved result sets from SQL queries to custom model classes implemented in Java, making the code very clean and easy to maintain and extend.

3.1.6 JUnit

JUnit is an open-source framework used for creating automated unit tests for Java applications. It allows us to make Java test classes and methods by marking them with custom annotations. These Java test classes and methods contain logic for testing our application. Each unit test should test only the smallest piece of code that is logically independent. They are also considered

as regression tests because we run them each time the application is extended to check if the previous functionality is not broken.

3.2 Modules

The prototype is divided into the following six modules based on the analysis of other implemented applications for a static code analysis of similar SQL dialects in Manta:

- **manta-connector-snowflake-dictionary** - this module is the implementation of the database dictionary, where we create all object declarations. It also contains a so-called policy. The policy is a set of rules for creating a hierarchy of objects in the dictionary.
- **manta-connector-snowflake-dictionary-extractor** - the extractor module is used to extract metadata and DDL definitions of Snowflake objects from a given Snowflake server instance.
- **manta-connector-snowflake-model** - contains model interfaces used by other modules. The module ensures compatibility between modules.
- **manta-connector-snowflake-resolver** - is responsible for lexical, syntactic, and semantic analysis. It parses, builds an AST, and resolves input scripts. The module contains all classes that are used to build the AST.
- **manta-connector-snowflake-testutils** - contains common test classes that are used or extended by classes from other modules.
- **manta-dataflow-generator-snowflake** - the module builds a dataflow graph from a resolved AST.

Each module represents a separate unit that accomplishes its responsibility. The following subsections describe the main individual modules.

3.2.1 Dependencies

The diagram 3.1 shows dependencies among the implemented modules. Modules `manta-connector-snowflake-dictionary`, `manta-connector-snowflake-resolver`, and `manta-dataflow-generator-snowflake` depend on `manta-connector-snowflake-model` providing a common AST interface for them. The module `manta-connector-snowflake-extractor` depends on the module `manta-connector-snowflake-dictionary` representing a data dictionary. `manta-connector-snowflake-testutils` is not shown in the diagram because its only purpose is to provide general testing classes for other modules. The implemented modules also depend on Manta's external modules that are

not shown in the diagram. The most important of them are described in the following subsections 3.2.2, 3.2.3, and 3.2.4.

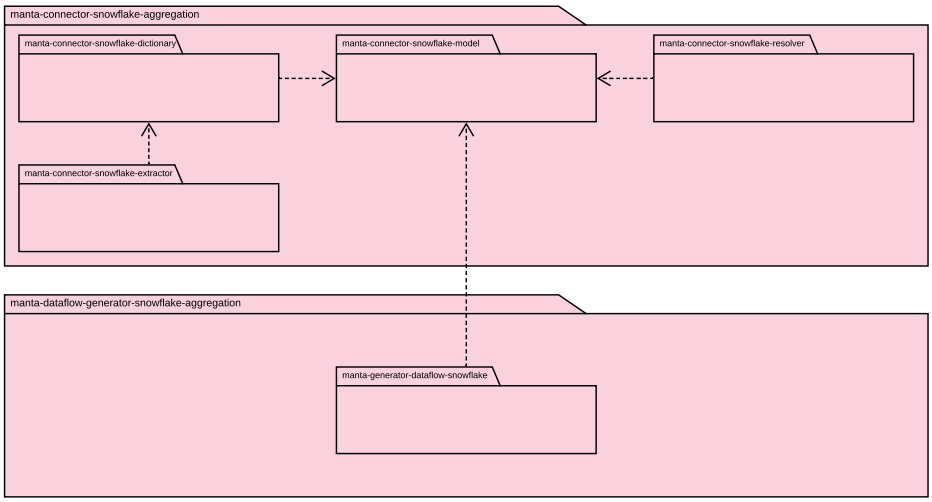


Figure 3.1: Diagram showing dependencies among implemented modules

3.2.2 Extractor

The following diagram 3.2 describes essential classes and their relationships inside the `manta-connector-snowflake-dictionary-extractor` module. `SnowflakeExtractor` contains a method called `extract()` that begins the whole extraction process. During the process, the class uses other classes shown in the diagram. `SnowflakeDao` is used to obtain metadata from a Snowflake server and map them to custom model classes such as `Table`, `View`, `Function` etc. All model classes extend an abstract class named `AbstractSnowflakeObject` containing common properties such as name. A class `SnowflakeDdlScriptGenerator` is used for generating DDL scripts from extracted metadata. To store these generated DDL scripts in the filesystem, `SnowflakeExtractor` uses a class called `SnowflakeDdlWriter`. `SnowflakeDdlWriter` uses `AliasManager` for normalizing directory or file names to avoid collisions when two strings have the same normalized string or when they contain invalid characters. The last essential thing is to store extracted metadata to a database dictionary. For this purpose, there is a class called `SnowflakeDictionaryWriter`. `SnowflakeDictionaryWriter` uses an external Manta module containing model classes for objects that we create from the extracted metadata and store in the dictionary. There exists one more dependency that is not shown in the diagram to the module representing the dictionary `manta-connector-snowflake-dictionary`.

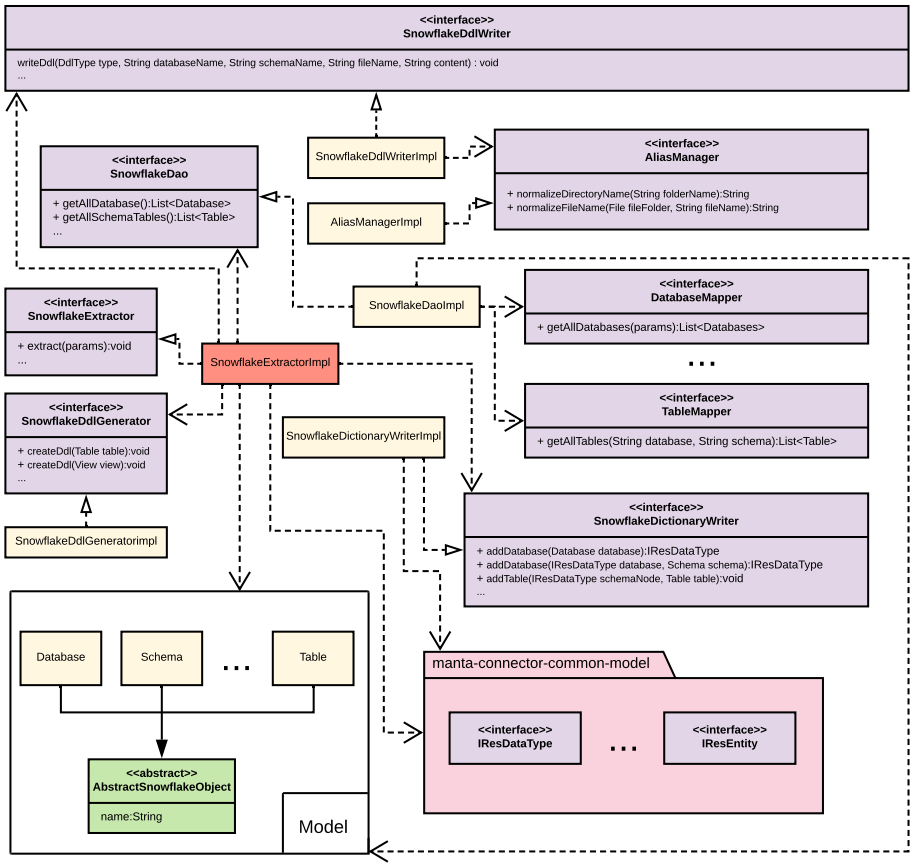


Figure 3.2: Diagram of essential classes and their relationships inside the extractor module.

3.2.3 Parser

In the following diagram 3.3, we can see essential classes and relationships inside `manta-connector-snowflake-resolver`. The module contains an important class called `ParserServiceImpl`. The class parses a string or file containing SQL scripts, builds an AST, and resolves the AST. Classes `SnowflakeMain`, `SnowflakeMain_NonReservedKW`, and `SnowflakeLexer` are generated from lexer and parser grammars. The generated classes extend external Manta classes `MantaAbstractParser` and `MantaAbstractLexer`, providing additional functionality. Lexer and parser are generated from grammar files called `SnowflakeLexer.g` and `SnowflakeMain.g`. They are located inside a package called `antlr3/parser`. Classes with a prefix of `Ast` represent AST nodes and contain logic executed during the resolving. They all extend a common class `SnowflakeAstNode` and implement model interfaces from the model module

`manta-connector-snowflake-model` for compatibility between modules.

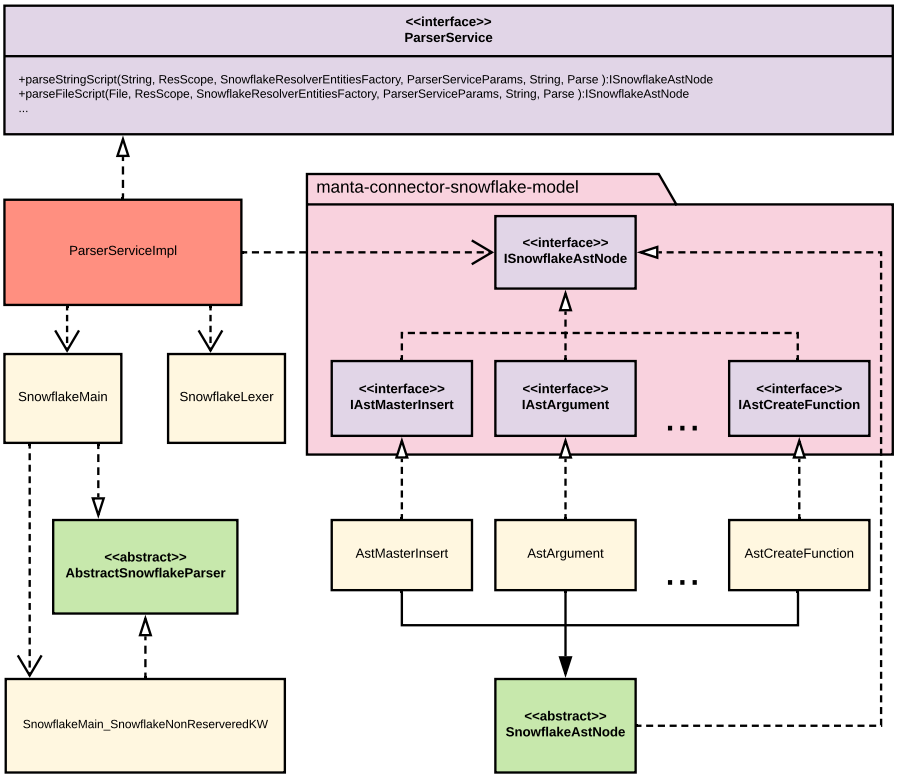


Figure 3.3: Diagram of essential classes and their relationships inside the resolver module.

3.2.4 Dataflow generator

The following diagram 3.4 shows essential classes and relationships inside the `manta-generator-dataflow-snowflake` module. The module depends on two other important modules. The first is `manta-connector-snowflake-model` containing an interface for the Visitor design pattern named `ISnowflakeAstVisitor`. A class `SnowflakeAstVisitorAdaptor` provides a default implementation of the visitor interface. `FlowVisitor` extends `SnowflakeAstVisitorAdaptor` and implements the visitor logic for creating a dataflow graph. `ISnowflakeAstNode` contains a method called `accept` accepting the visitor interface and ensures that each AST node must implement the method. FlowVisitor uses a helper class for building the dataflow graph called `SnowflakeGraphHelper`. The class extends an external AbstractGraphHelper containing im-

plemented methods. `SnowflakeGraphHelper` uses classes from the `manta-dataflow-model` module for creating the dataflow graph.

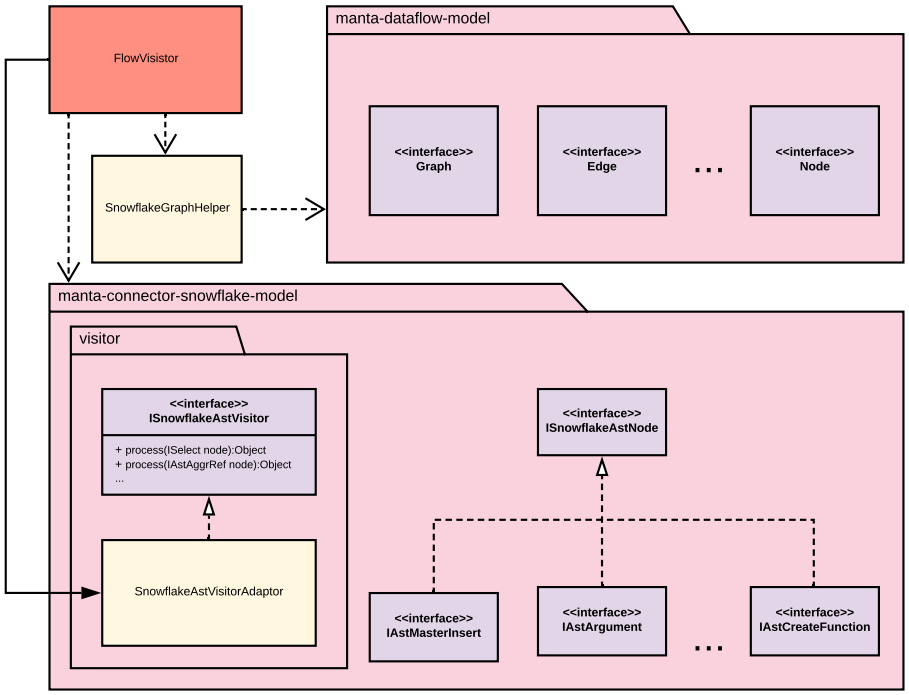


Figure 3.4: Diagram of essential classes and their relationships inside the dataflow module.

3.2.5 Execution flow

Each module represents a separate logical unit accomplishing a task. These tasks must be executed in the right order in order to produce a dataflow graph. The following diagram shows a simplified execution order of all implemented modules. The first executed module is `manta-connector-snowflake-extractor` that extracts metadata into a data dictionary together with DDL scripts. The next module is `manta-connector-snowflake-resolver`, which takes the data dictionary, extracted DDL scripts and input SQL scripts, and creates a resolved AST tree for `manta-dataflow-generator-snowflake`. The dataflow module then processes the resolved AST and builds the dataflow graph.

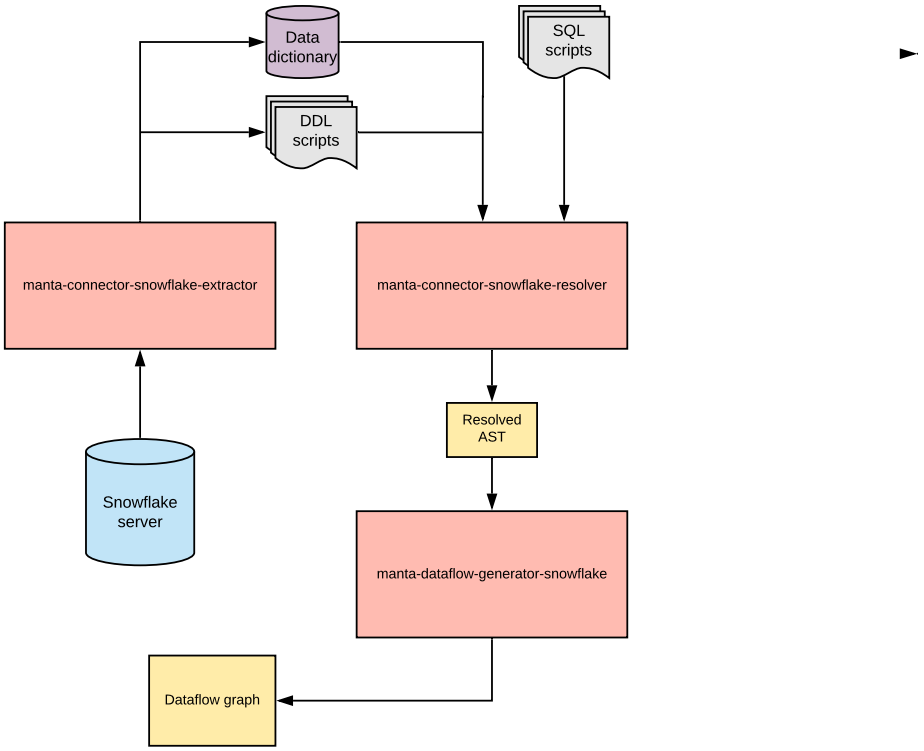


Figure 3.5: Diagram showing an execution flow of modules

Chapter 4

Implementation

This chapter describes the implementation aspects of the implemented prototype in more detail. It describes interesting classes and constructs. The chapter also aims to provide a big picture of resolving for a reader to better grasp the concept.

4.1 Extractor

This subsection describes implementation constructs of the `manta-connector-snowflake-extractor` module.

4.1.1 SnowflakeExtractor and SnowflakeExtractorImpl

`SnowflakeExtractorImpl` is the implementation of the `SnowflakeExtractor` interface. The class contains implemented a following set of important methods:

- **`setDictionary / getDictionary`** - setter and getter method for a data dictionary where all extracted metadata are stored.
- **setDdlOutputDirectory / getDdlOutputDirectory** - setter and getter method to set up a directory where extracted DDL scripts are stored.
- **setOutputDdlTypes** - sets types of objects for which their DDL are extracted.
- **setExtractedDdlTypes** - sets types of objects to extracted.
- **setIncludedDbsSchemas / setExcludedDbsSchemas** - sets a filter for extracted databases and schemas. The method uses Manta's class named `DatabaseSchemaFilter` for this purpose. Databases and

schemas present in the exclude filter are not extracted unless they also appear in the include filter.

- **extract()** - starts the whole extraction process. In the beginning, the method first checks whether an output directory is correctly set. After the check, the method extracts all databases. For each database that is not excluded, the method extracts all schemas. For each extracted schema that is not excluded, the method extracts all objects inside the schema. Each object is only extracted if its type is present inside the `ExtractedDdlTypes` set. For each object type exists a method named `extract*` (for example, `extractTables()`) that extracts all objects of a given type from the schema. The `extract*` method uses `SnowflakeDao` to extract objects. Each extracted object is saved in the dictionary by `SnowflakeDictionaryWriter`. The `extract*` method also generates and stores (in the filesystem defined by the output directory) DDL definition for each extracted object that is present inside the `outputDdlTypes` set. In the end, a method `persist()` is called to save all the changes in the dictionary to the disk.

4.1.2 SnowflakeDao and SnowflakeDaoImpl

`SnowflakeDaoImpl` implements an interface SnowflakeDao that extracts metadata from a provided Snowflake server instance using the MyBatis framework. To extract the metadata, it uses a set of custom mappers called `*Mapper` where `ASTERISK (*)` stands for an object type name (for example, `TableMapper`). Each mapper is an interface containing a set of methods to extract its specific object type (for example, `TableMapper` contains methods for extracting tables). MyBatis implements the interfaces based on corresponding configuration files called `*Mapper.xml` (for example, `TableMapper.xml`) located in the `extractor.mappers` directory.

4.1.3 SnowflakeDdlGenerator

This interface is implemented by a class called `SnowflakeDdlGeneratorImpl` that is used to generate needed DDL scripts for some Snowflake objects. Generated DDL scripts do not need to be complete definitions. They usually contain only useful information; for example, table definitions need to contain columns and their data types but do not need to contain many additional properties such as data retention or collation. The class contains methods called `createDdl` that differ in type of the input argument (for example `createDdl(Table t)`, `createDdl(Function f)`).

4.1.4 SnowflakeDdlWriter

`SnowflakeDdlWriterImpl` implements an interface `SnowflakeDdlWriter` for creating a directory structure and writing generated DDL scripts to the disk.

4.1.5 AliasManager

`AliasManagerImpl` is a utility class that solves problems with invalid file names. There might exist two DDL files with the same name or files that might contain special characters that are not supported on many file systems. The class creates unique file names by replacing unsupported characters with the character `_` and adding a numerical index to the name. The unique file names are stored in internal caches.

4.1.6 SnowflakeDictionaryWriter

A class `SnowflakeDictionaryWriterImpl` implements this interface. The purpose of the class is to add extracted metadata to the database dictionary using Manta's external structures. The class contains methods named `add*` (for example, `addTable()`) used to add a specific object type to the dictionary. There are no dependencies among database objects in Snowflake, so the objects are added in no arbitrary order to the dictionary.

4.1.7 ParsingUtils

This is a util class containing many methods used for parsing. The class contains the following two main methods:

- **parseSignature** - method is used to parse an argument or a return signature of a function. The `parseSignature` function returns a list of columns representing the parsed arguments from the signature.
- **parseJdbcUrl** - parses a Snowflake JDBC connection string and creates `ServerInfo` that contains information such as an account, region, and cloud type from the JDBC connection string.

4.1.8 Model classes

The extractor module contains a set of custom model classes representing Snowflake database objects that are used to hold extracted metadata. MyBatis creates instances of them with mapped metadata during the execution of methods defined in `*Mapper` files.

4.2 Parser

This section describes the main classes and files of the `manta-connector-snowflake-resolver` module.

4.2.1 Lexing grammar files

`SnowflakeLexer.g` is a grammar file, which Antlr uses to generate a lexer class named `SnowflakeLexer` that is used for lexical analysis. The file contains lexer rules that mostly contain regular expressions for recognizing character sequences. `SnowflakeLexer` creates a token stream that consists of tokens produced from recognized character sequences by the defined rules. The lexer rules describe constructs such as reserved and non-reserved keywords, regular identifiers, delimited identifiers, comments, many operators, etc. All rules that define non-reserved words are prefixed with `KW_`. The following code 4.1 reflects some of the stated rules defined in `SnowflakeLexer.g`.

```
1
2  // Reserved words
3  USING : 'USING';
4  AND   : 'SELECT';
5  FROM  : 'FROM';
6
7  // Non reserved words
8  KW_DATABASE : 'DATABASE';
9  KW_SCHEMA   : 'SCHEMA';
10 KW_MERGE    : 'MERGE';
11
12 fragment DIGIT         :  '0'..'9';
13 fragment UNDERSCORE    :  '_';
14 fragment DOUBLE_QUOTES :  '"';
15
16 REGULAR_ID
17  :  (UNDERSCORE | LETTER)
18     (UNDERSCORE | LETTER | DIGIT | DOLLAR_SIGN)*
19  ;
20
21 DELIMITED_ID
22  :
23     DOUBLE_QUOTES
24     (~(DOUBLE_QUOTES) | DOUBLE_QUOTES DOUBLE_QUOTES)*
25     DOUBLE_QUOTES
26  ;
```

Listing 4.1: Lexer rules from SnowflakeLexer.g

Fragment rules are special lexer rules that are only used by other rules. The lexer does not create tokens from them.

4.2.1.1 Generating custom Java functions

Sometimes it is not possible to write lexing rules that consist only from other fragment rules to recognize more complex character sequences. In such a case, we can create our own custom Java functions that we can use in lexing rules. The following code 4.2.1.1 shows a defined Java function that is used inside a lexing rule.

```
1  @members {
2
3      public void consumeDollarQuotedString()
4                  throws MismatchedTokenException {
5          char next1 = (char) input.LA(1);
6          while(next1 != '\$' || input.LA(2) != '\$') {
7              if ( next1 == -1 ) {
8                  throw new MismatchedTokenException ( '\$', input );
9              }
10             match(next1);
11             next1 = (char) input.LA(1);
12         }
13     }
14 }
15
16 DOLLAR_QUOTED_STRING_LIT
17 :   '$' '$' {
18         consumeDollarQuotedString();
19     }
20     '$' '$'
21 ;
```

Source Code 4.1: Lexer java code

The function recognizes body definitions of Snowflake user-defined functions (UDF). Each function definition is enclosed by double dollars ($$) and might contain a single dollar inside. The function recognizes characters from an input stream and stops only when it sees two consecutive dollar characters in look ahead.

4.2.2 Parsing grammar files

The parsing grammar is split into two files named `SnowflakeMain.g` and `SnowflakeNonReservedKW.g`. Antlr uses them to generate parser classes called

`SnowflakeMain.java` and `SnowflakeMain_SnowflakeNonReserveredKW`. `SnowflakeMain.g` contains parsing rules that describe the Snowflake SQL dialect. `SnowflakeNonReserved.g` grammar file contains grammar rules describing Snowflake identifiers, reserved, and non-reserved keywords. Parser rules consist of other defined alternative parsing rules and lexing rules defined in a lexer grammar. Lexing rules are used to match tokens from a token stream. Because only a valid SQL script can be used as an input into the parser, many parsing rules describe a superset of the Snowflake dialect for their simplification.

4.2.2.1 Grammar structure

A `snowflake_script` rule is the first entrance rule used by `ParserServiceImpl` to start parsing a Snowflake SQL script. The rule consists of rules describing the structure of all implemented Snowflake statements. The listed rules at this level are Snowflake statements that can appear independently, such as `select_statement` or `insert_statement`. The following code 4.2 is only a very simplified illustration of parsing rules for a reader to get a better idea about the basic grammar structure.

```
1
2  snowflake_script
3      :  bl = statement_list eof = EOF
4         -> ^(AST_SCRIPT $bl?)
5      ;
6
7  statement_list
8      :  common_table_expression_statement
9      |  insert_statement
10     |  delete_statement
11     |  update_statement
12     |  merge_statement
13     |  use_statement
14
15     |  create_table_statement
16     |  create_view_statement
17     |  create_database_statement
18     |  create_schema_statement
19     |  create_stage_statement
20     |  create_file_format
21     |  create_function_or_procedure_statement
22     ;
```

Listing 4.2: Parsing rules from SnowflakeMain.g

4.2.2.2 Identifiers

A parsing rule describing Snowflake identifiers consists of alternative paths matching regular unquoted identifiers, quoted identifiers, and non-reserved words because they are not reserved by the Snowflake dialect and therefore, can be used as identifiers. The following example 4.3 shows a rule for matching any valid Snowflake identifier.

```
1
2  identifier
3      :
4          REGULAR_ID
5      |   DELIMITED_ID
6      |   non_reserved_words
7      ;
8
9  non_reserved_words
10     :
11         KW_DATABASE
12     |   KW_SCHEMA
13     |   KW_ACCOUNT
14     ...
15     ;
```

Listing 4.3: Parsing rules from SnowflakeMain.g

4.2.2.3 Rewrite rules

`SnowflakeMain.g` contains so-called rewrite rules that are often defined inside the paring rules. Each alternative path inside a parsing rule can have defined a rewrite rule. They are used to create imaginary nodes, allowing us to model AST the way we want. This concept was briefly explained in 1.1.2. The rewrite rules start with `->` character. In the example 4.2, we can see a rewrite rule specified in the rule `snowflake_script`.

4.2.3 ParserService and ParserServiceImpl

`ParserService` is a standard parser interface in the Manta project. The interface is implemented by a class called `ParserServiceImpl`. The class implements `parseStringScript` and `parseFileScript` methods that parse a passed string or file containing a Snowflake SQL script. One of the essential parameters for parsing a file or string is `ParserServiceParams`. The class is used to pass additional parameters such as default database name, current database name, current schema name, and search path.

The following diagram 4.1 shows a simplified sequential execution of method calls inside `ParserServiceImpl`. First, one of the mentioned methods that

accepts the SQL script is called. Then an instance of the lexer class is created that accepts the input SQL script and creates a token stream. After this step, the method creates an instance of the parser class `SnowflakeMain` that accepts the created token stream and calls the entrance parsing method `parse_script()` (in the diagram, it is simplified by just calling a `parse()` method with the passed token stream for easier understanding). The method creates AST and returns the root of the tree. The root of the tree is an object of type `SnowflakeAstNode`. The last step is to call a method `resolve()` from the root that starts semantic analysis (also called resolving). The `parseStringScript` or `parseFileScript` method returns resolved AST, where nodes contain references to object declarations stored in a data dictionary.

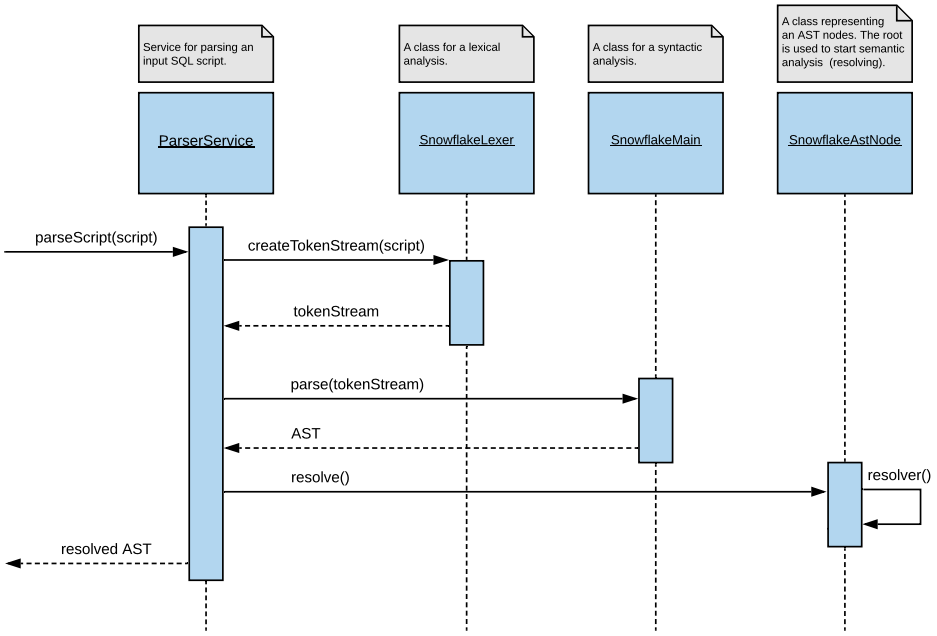


Figure 4.1: Sequential diagram of the `ParserServiceImpl` class

4.2.4 SnowflakeAstNode

The result of parsing a Snowflake SQL script is an AST. SnowflakeAstNode is the ancestor of each node in the AST. It implements an interface named `ISnowflakeAstNode` that prescribes the following set of methods:

- **`getParent()`** - returns a parent node
- **`getChildren()`** - returns a list of children nodes of the current node

- `accept()` - accepts a visitor class that implements `ISnowflakeVisitor` interface

4.2.4.1 Resolving

`SnowflakeAstNode` contains a method called `resolve()`. Some nodes extending this class have overloaded the `resolve()` method containing implemented logic to find declarations of them and their child nodes in a data dictionary. The overloaded `resolve()` method usually only delegates to a function `resolveInternal()` where the logic is usually implemented. The classes with overloaded `resolve()` are mainly nodes representing identifiers or Snowflake statements. The resolving starts when the `ParserServiceImpl` class calls the `resolve()` method on the root of AST. The default implementation of the method calls `resolve()` method on children nodes. This process is repeated recursively. At the end of the resolving, we have got the resolved AST where all identifier nodes contain references to their declarations. The following code example 4.2.4.1 shows the default implementation of `resolve()`.

```
1 public IResEntity resolve() {
2     //default implementation, do nothing, just delegate to childs
3     for (SnowflakeAstNode item : getChildren()) {
4         item.resolve();
5     }
6
7     return null;
8 }
```

Source Code 4.2: Default implementation of the `resolve()` method

4.2.5 Ast*

If an AST node has a special meaning, then it is created using any of the `SnowflakeAstNode`'s descendants. The descendants are special classes prefixed with `Ast` used as AST nodes. They usually contain resolving logic or various auxiliary methods used during the resolving. An example of such a class is `AstMasterInsert` representing the `INSERT` statement. It has the overloaded `resolve()` method that assigns declarations from a data dictionary to all descendant nodes representing identifiers. The classes are located in the `resolve/ast/impl` package.

4.2.6 SnowflakeContextState

The class extends `ParserContextState`. It represents a context that contains references on useful objects which are needed during parsing or resolving. For example, during the resolving, it is used to resolve unqualified identifiers because it contains a search path (2.3.4 explains what the search path is) with schema declarations where the unqualified identifiers are sequentially searched.

4.2.7 ResScope

The class is used to represent a scope as we know from programming languages, where it refers to the visibility of variables. Scopes are created during parsing. The purpose of scopes is to store and search for important objects during resolving. Global scope is always created first. The global scope contains other scopes like `SELECT` and `FUNCTION` scopes. Each function and `SELECT` statement contains its own scope. The `FUNCTION` scope is used to store and search input parameters. The `SELECT` scope is, for example, used to store and search named subqueries defined inside the `WITH` clause.

4.3 Dataflow generator

This section aims to describe the essential classes of the `manta-dataflow-generator-snowflake` module used to build a data flow graph.

4.3.1 FlowVisitor

FlowVisitor is the implementation of the Visitor design pattern. It processes a resolved AST and creates a dataflow graph. The class implements a `process()` method for each node type. The method is used to process every AST node and optionally builds nodes and data flows among them in the output graph. For creating nodes and flows in the output graph, `FlowVisitor` uses a helper class called `SnowflakeGraphHelper`.

4.3.2 SnowflakeGraphHelper

SnowflakeGraphHelper is a helper class for building a dataflow graph. It extends a class called `AbstractGraphHelper` from Manta's external libraries. `SnowflakeGraphHelper` uses a `buildNode` method for creating nodes. The method maps AST nodes to their corresponding nodes of the output dataflow graph. `SnowflakeGraphHelper` uses common methods `addNode` from `AbstractGraphHelper`.

Chapter 5

Testing

Implemented modules are covered by tests. JUnit is used for creating unit tests. The following sections describe the tests of the individual modules.

5.1 manta-connector-snowflake-resolver

The module uses a particular class named `AnnotatedFilesResolverTest`. The class tests SQL scripts based on special annotations. These annotations in the test files describe an expected result after resolving a code. SQL tests are located in a directory called `SimpleTests`. Each test file is named according to a specific functionality it tests.

```
1  CREATE TABLE t1 (
2    a /*=t1a*/ INT,
3    b /*=t1b*/ INT,
4    c /*=t1c*/ INT
5  );
6
7  CREATE TABLE t2 (
8    a /*=t2a*/ INT,
9    b /*=t2b*/ INT
10 );
11
12 SELECT * FROM t1 INNER JOIN t2 ON t1.a/*=t1a*/ = t2.a/*=t2a*/;
13 /* A|B|C|A|B */
14 SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.a/*=t1a*/ = t2.a/*=t2a*/;
```

Source Code 5.1: Example of annotated test file

The source code example 5.1 shows part of the annotated test file called `join.sql`. It tests if the script is correctly parsed and if all identifiers contain references to the correct declarations.

5.2 manta-connector-snowflake-dataflow

The module is tested by using `AstFilesFlowTest`. The tests verify the correctness of generated dataflow graphs based on comparing them with the corresponding files representing correctly generated graphs. Each test file is named according to the functionality it tests. Their corresponding expected result files have the same name with a suffix `_expected`. All the mentioned files are located in the `flow` directory.

5.3 manta-connector-snowflake-extractor

The module contains a set of classes used for testing independent functionalities of the extractor module. The extractor contains the following set of test classes:

- **`SnowflakeTestResource`** - this is a resource class that contains the same structure as Snowflake server used for testing. Other test classes use the resource class, for example, to compare extracted results from the Snowflake server.
- **`SnowflakeDictionaryWriterTest`** - the class is used to test the `SnowflakeDictionaryWriter` class. It contains a set of methods testing if extracted metadata are correctly stored inside the data dictionary.
- **`JdbcUrlParseTest`** - the class tests if a Snowflake JDBC URL is correctly parsed and all the information is correctly retrieved.
- **`SnowflakeDDLWriterTest`** - tests if generated DDL scripts are correctly saved to the disk.
- **`SnowflakeDDLScriptGeneratorTest`** - tests if `SnowflakeDdlScriptGenerator` generates valid DDL scripts.
- **`SnowflakeDaoTest`** - tests if metadata are correctly extracted from the Snowflake test server.

Chapter 6

Output Samples

The chapter shows simple SQL scripts written in the Snowflake SQL dialect with the resulting dataflow graphs. The examples focus on a few statements implemented in the prototype. Only the most important nodes and edges are captured in individual graphs. The chapter also shows how the dataflow graphs look in the Manta tool.

6.1 Description of graph picture

The following figures 6.1, 6.3 show generated dataflow graphs that consist of nodes and edges. Each node contains a name, type, and a name of its parent node. Name of the parent node is present inside parenthesis `()`. The type of the node is present inside brackets `[]`. For example, `A[Column](T1)` refers to a node called `A` of type `Column` with the parent node `T1`.

6.2 SELECT statement

In the following code example 6.1, we can see a code of a simple `SELECT` statement. First, we create a table `t1`, and then the table is used as the source for the `SELECT` statement. The statement also contains a conditional `WHERE` clause to filter output rows according to the value of column `a`.

```
1 CREATE TABLE t1(a INT, b INT);
2
3 SELECT * FROM t1 WHERE a > 10;
```

Source Code 6.1: Example of a `SELECT` statement

The following figure 6.1 shows the resulting dataflow graph for the source code 6.1. The graph consists of the following important nodes:

- `T1` - represents the source table `t1`. In the graph we can also see its columns `A`, `B`
- `<3,1>ResultSet` - represents a single `SELECT` unit. If a `SELECT` statement contains set operations there exist more `SELECT` units, thus more ResultSets. The resultset contains resultset columns `A` and `B`.
- `<3,1>MasterResultSet` - represents the whole `SELECT` query and contains result set columns `A` and `B`
- `<3,18>WHERE` - represents a conditional node.

Direct edges lead from columns `A`, `B` of the source table `T1` to the resultset columns `A`, `B`. From the resultset columns, edges proceed to the columns `A` and `B` of `<3,1>MasterResultSet`. Filter flows connects the conditional node `<3,18>Where` with the resultset columns of `<3,1> ResultSet`.

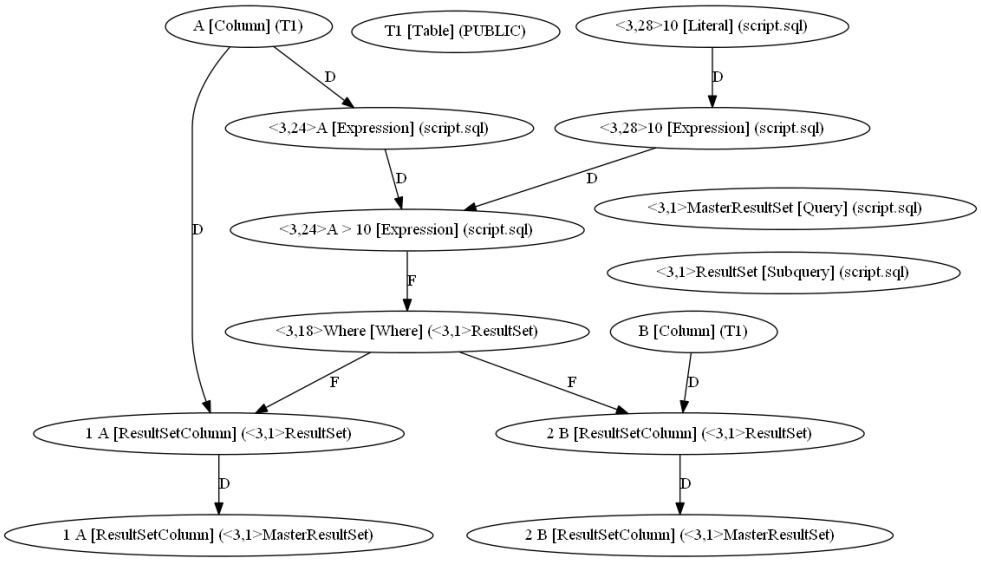


Figure 6.1: Dataflow graph of the `SELECT` statement from 6.1

In figure 6.2, we see the same dataflow graph visualized in the Manta tool. The graph is simplified and does not contain many nodes that are not important for clients, such as for example, expression nodes. Manta tool shows only the most important nodes, and edges for clear visualization of data flows.

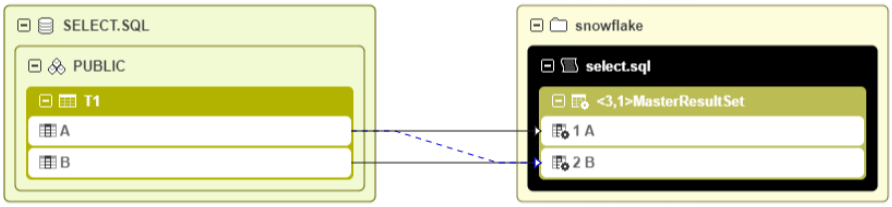


Figure 6.2: Dataflow graph of the `SELECT` statement from 6.1 visualized in the Manta tools

6.3 INSERT statement

The following source code shows an example of a simple `INSERT statement`. First, we create two tables `t1` and `t2`. Table `t1` is used as the target table, and `t2` is used as the source table. The `INSERT` statement writes data to `t1` from a `SELECT` query, which reads data from `t2`.

```
1 CREATE TABLE t1(a INT, b INT);
2
3 CREATE TABLE t2(a INT, b INT);
4
5 INSERT INTO t1(a, b) SELECT a, b FROM t2;
```

Source Code 6.2: Example of an `INSERT` statement

Figure 6.3 shows the resulting dataflow graph for the source code 6.2 of the `INSERT` statement. In the figure, we can see the following important nodes:

- `T1` - represents the target table `t1`. In the graph we can also see its columns `A`, `B`
- `T2` - represents the source table `t2`. In the graph we can also see its columns `A`, `B`
- `<5,22>ResultSet` - represents a `SELECT` unit
- `<5,22>MasterResultSet` - represents the whole `SELECT` query
- `<5,1>INSERT` - node representing an operation, in this case the insert operation and its column flows `A` and `B`

Direct edges lead from columns `A`, `B` of the source table `T2` to the columns `A`, `B` of the target table `T2` through resultset columns and columns of the `<5,1>INSERT` operation node.

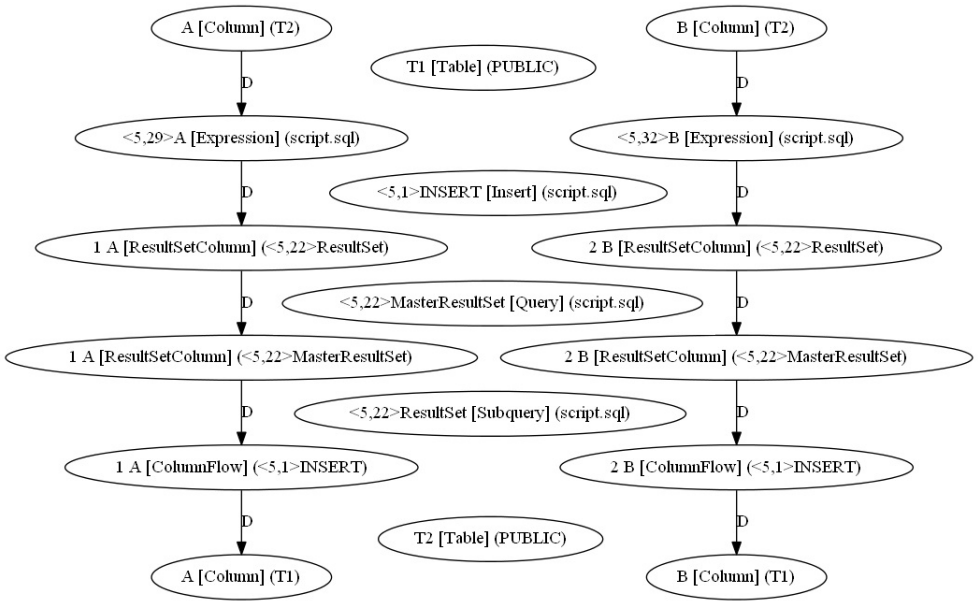


Figure 6.3: Dataflow graph of the `INSERT` statement from 6.2

The following figure 6.4 shows the same source code visualized in Manta. We can see direct flows going from columns of `T1` to columns of `T2` only through the `INSERT` operation node. The resultset nodes are filtered by Manta because they are not important to see in the resulting graph.

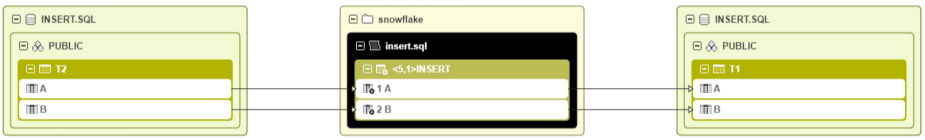


Figure 6.4: Dataflow graph of the `INSERT` statement from 6.2 visualized in the Manta tool

Conclusion

The goal of the thesis was to analyze syntax and semantics of the Snowflake SQL dialect, analyze the metadata of the Snowflake database engine, and learn about the Manta project, about how it analyzes and represents similar SQL dialects, and how it represents data flows. Among other goals was to analyze and design how to represent Snowflake source code for later analysis of data flows. The last goal was to implement the prototype that extracts metadata from the Snowflake database engine, analyzing Snowflake SQL scripts and producing a valid data flow graph.

All goals in the thesis were accomplished. A method of analysis and representation of source codes in the Snowflake dialect was found, as well as a method of data flows detection. The prototype is implemented, and the main functionalities are tested. The prototype can analyze more statements than it was originally required in the thesis. The prototype was also successfully integrated with Manta, and the alfa version is about to be released in a month to the production environment.

The prototype still cannot analyze the whole Snowflake SQL dialect. Thus, the prototype will be gradually extended in the future to allow the analysis of new statements.

Bibliography

[1] Wikipedia contributors. *Static program analysis* [online]. 28 June 2020, at 15:58 (UTC). [cit. 2020-07-04]. Available from: `https://en.wikipedia.org/wiki/Static_program_analysis`.

[2] Hopcroft, J., Motwani, R., Ullman, J. *Introduction to Automata Theory, Languages, and Computation*. 3rd Edition. Addison Wesley, Pearson, 2006. 535. ISBN: 0-321-45536-3.

[3] Aho, A.V., Lam, M.S., Sethi, R. and Ullman, J.D. *Compilers: Principles, techniques, and tools*. Second edition. Boston:Pearson education, 2007. 993. ISBN 0-321-48681-1.

[4] Cooper, K. and Torczon, L. *Engineering a compiler*. Second edition. 2011. 787. ISBN: 978-0-12-088478-0.

[5] Michael L. Scott. *Programming Language Pragmatics*. Third edition. 2011. 787. ISBN 13: 978-0-12-374514-9.

[6] Olivia Wassén. *What is Data Lineage?* [online]. June 7th, 2019. [cit. 2020-07-05]. Available from: `https://www.nodegraph.se/what-is-data-lineage`.

[7] *Tech Summary – MANTA* [online]. MANTA. Apr 2019. [cit. 2020-07-06]. Available from: `https://getmanta.com/scanners-and-integrations/tech-summary/`.

[8] Jan Andrš. *How To Inspect Raw Data Lineage With Manta Flow* [online]. MANTA. June 1st, 2016. [cit. 2020-07-06]. Available from: `https://getmanta.com/blog/how-to-inspect-raw-data-lineage-with-manta-flow/`.

[9] Snowflake. *Key Concepts & Architecture* [online]. [cit. 2020-07-10]. Available from: https://docs.snowflake.com/en/user-guide/intro-key-concepts.html.

[10] Snowflake. *Information Schema* [online]. [cit. 2020-07-12]. Available from: https://docs.snowflake.com/en/sql-reference/info-schema.html.

[11] Snowflake. *Identifier Requirements* [online]. [cit. 2020-07-12]. Available from: https://docs.snowflake.com/en/sql-reference/identifiers-syntax.html.

[12] Snowflake. *Summary of Data Types* [online]. [cit. 2020-07-14]. Available from: https://docs.snowflake.com/en/sql-reference/intro-summary-data-types.html.

[13] Snowflake. *Reserved & Limited Keywords* [online]. [cit. 2020-07-15]. Available from: https://docs.snowflake.com/en/sql-reference/reserved-keywords.html.

[14] Snowflake. *Object Name Resolution* [online]. [cit. 2020-07-15]. Available from: https://docs.snowflake.com/en/sql-reference/name-resolution.html.

[15] Snowflake. *Understanding & Using Time Travel* [online]. [cit. 2020-07-17]. Available from: https://docs.snowflake.com/en/user-guide/data-time-travel.html.

[16] Snowflake. *Querying Semi-structured Data* [online]. [cit. 2020-07-19]. Available from: https://docs.snowflake.com/en/user-guide/querying-semistructured.html.

[17] *What is Java technology and why do I need it?* [online]. [cit. 2020-07-27]. Available from: https://java.com/en/download/faq/whatis_java.xml.

[18] Margaret Rouse. *Spring Framework* [online]. August, 2019. [cit. 2020-07-27]. Available from: https://searchapparchitecture.techtarget.com/definition/Spring-Framework.

[19] Terence Parr. *The Definitive ANTLR Reference*. 2007, ISBN: 0-9787392-5-6.

[20] The Apache Software Foundation. *What is Maven?* [online]. April 4th, 2018. [cit. 2020-07-27]. Available from: https://maven.apache.org/what-is-maven.html.

[21] MyBatis.org. *Introduction* [online]. 05 June 2020. [cit. 2020-07-27]. Available from: https://mybatis.org/mybatis-3/.

Appendix A

Acronyms

BI Business Intelligence

ETL Extract, Load, Transform

AST Abstract Syntax Tree

SQL Structured Query Language

IDE Integrated Development Environment

SaaS Software as a Service

MPP Massively Parallel Processing

ANSI American National Standards Institute

DDL Data Definition Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

ANTLR ANother Tool for Language Recognition

JDBC Java Database Connectivity

UDF User Defined Function

UDTF User Defined Table Function

ASCII American Standard Code for Information Interchange

CTE Common Table Expression

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with USB contents description
- `src` ... the directory containing source codes of the implemented modules
  - `manta-connector-snowflake-aggregation`
    - `manta-connector-snowflake-dictionary`
    - `manta-connector-snowflake-extractor`
    - `manta-connector-snowflake-model`
    - `manta-connector-snowflake-resolver`
    - `manta-connector-snowflake-testutils`
  - `manta-dataflow-generator-snowflake-aggregation`
    - `manta-dataflow-generator-snowflake`
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format