FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Data flow analysis of scripts in Databricks SQL dialect
Student:	Bc. Lucie Procházková
Supervisor:	Ing. Jan Trávníček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

Study the Databricks dialect of SQL language – its syntax and semantics.
Familiarize yourself with the Manta project, its metadata and the way it represents data flow and with how the Databricks objects are retrieved from the Databricks database in the Manta project; if needed, propose and implement additions.
Analyze whether it is possible to retrieve information about data flow between data structures in a Databricks database from scripts in Databricks SQL language using static analysis of these scripts.
Propose an approach to parsing and representing Databricks SQL scripts; design a suitable datastructure for a follow-up dataflow analysis. Describe its statements relevant to the later data flow analysis.
Continue with a design of analysis of Databricks SQL scripts capable of retrieval of the data flow between data structures in Databricks SQL database from Databricks SQL scripts and an approach to represent this data flow in the Manta project.
Create a prototype implementation of a tool that can retrieve data flow between data structures in the Databricks SQL database from Databricks SQL scripts and can store this data flow in the Manta project.
Propose and implement testing of your prototype.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 29 December 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Data flow analysis of scripts in Databricks SQL dialect

Bc. Lucie Procházková

Department of Theoretical Computer Science

Supervisor: Ing. Jan Trávníček Ph.D.

May 4, 2023

Acknowledgements

I want to thank my supervisor, Ing. Jan Trávníček, Ph.D., for his guidance and support throughout my research. Thanks for all comments and highly detailed reviews of all the pull requests.

I also want to thank Ladislav Louka for his continuous support and always welcomed advice. Thank you, I would not be able to finish this on time without you.

I have to mention here also all my friends, who supported me not only through writing this thesis but through all my studies. I can't mention them all here, so at least Matěj Havránek, Tomáš Kořistka, Linda Beková, Antonín Kříž, Adam Procháska, Michal Dvořák and Xuan Thang Nguyen.

Lastly, I would like to acknowledge Manta, for giving me the opportunity to conduct this research. Special thanks go to my amazing parsers team.

Declaration

Citation of this thesis

Procházková, Lucie. *Data flow analysis of scripts in Databricks SQL dialect.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstract

This thesis investigates the Databricks SQL dialect, proposing an automatic script analysis method and a prototype scanner unit for Manta, a data lineage tool. Data lineage is essential for data integrity and governance.

The research outcomes include a comprehensive data flow analysis in Databricks SQL, the prototype scanner unit design and implementation, and thorough testing. Our contributions enhance Manta's ability to work with Databricks systems, providing a valuable analytic tool for organizations relying on Databricks SQL for data processing.

Klíčová slova Databricks, SQL, data lineage, data flow analysis, Manta tools, ANTLR, parsing

Abstrakt

Tato diplomová práce zkoumá dialekt Databricks SQL a navrhuje metodu automatické analýzy skriptů a prototyp skeneru pro nástroj Manta, který se zabývá data lineage. Data lineage je nezbytná pro integritu a správu dat.

Výsledky výzkumu zahrnují komplexní analýzu datového toku v Databricks SQL, návrh a implementaci prototypu skeneru a důkladné testování. Naše práce rozšiřuje schopnosti nástroje Manta v práci se systémy Databricks a poskytují cenný analytický nástroj pro organizace, které spoléhají na Databricks SQL pro zpracování dat.

Keywords Databricks, SQL, data lineage, analýza datových toků, Manta tools, ANTLR, parsing

Contents

List of Figures

List of code snippets

Introduction

Data lineage is a crucial concept in modern data management. As data infrastructure grows in complexity, keeping track of how data is sourced, transformed, and used across the organization is challenging. This complexity poses significant challenges for data consumers, engineers, and governors.

Data lineage provides a practical solution for addressing these challenges by offering a comprehensive understanding of how data flows through an organization. It allows stakeholders to trace errors back to their root causes, perform impact analysis before making changes, and understand how sensitive data is used throughout the organization to ensure compliance with regulatory requirements.

Without data lineage, it can be difficult for data consumers to understand the origins of data and how it has been transformed, leading to issues of trust in reports and insights. Data engineers also face challenges when trying to troubleshoot issues related to data transformations without a reliable way to track them. Data governance teams may also struggle to ensure compliance with regulations when they cannot easily understand how sensitive data is used across the organization.

Knowing how the data flows through the system enables better decision-making by providing a clear and comprehensive view of the data flow within an organization. Using lineage graph visualizations, stakeholders can understand how data is sourced, transformed, and used, enabling better change management and preventing unexpected downtime or errors. [1, 2]

Databricks is a unified data analytics platform designed to accelerate innovation by unifying data engineering, data science, machine learning, and analytics. It is built upon the open-source Apache Spark framework, providing a scalable and collaborative environment for processing large volumes of data. Databricks has gained popularity due to its ability to simplify complex data processing tasks, foster collaboration among diverse teams, and support advanced analytics and machine learning use cases. Its cloud-based architecture and integration with major cloud providers further contribute to its widespread adoption in various industries.

Manta is a project that focuses on managing and analysing data lineage, offering organisations a comprehensive view of the flow of their data across different systems and applications. The Manta project analyses metadata about the structure of the data. It provides metadata about data flow and its representations, making it a valuable tool for organisations to understand the life cycle of their data and make informed decisions about it.

This thesis aims to study the Databricks dialect of SQL language and its syntax and semantics. The implementation part of this thesis will focus on extending the Manta tool to retrieve information about data flow in Databricks SQL scripts through static analysis of these scripts. The goal of this thesis is to implement a prototype of a scanner unit of the Manta product, which will be used to analyze SQL code in the Databricks system.

The thesis is organized into six chapters, each focusing on a different aspect of the research and implementation. The first chapter will provide a theoretical background, focusing on parsing, static analysis of formal languages, automata and grammar theory, and related topics. This chapter will help the reader understand the concepts and techniques used in the thesis.

In the second chapter, the Databricks system will be introduced, as well as the use of SQL in it. The chapter also introduces the philosophy of the system and covers the data objects in the Databricks ecosystem.

The third chapter will analyze the Databricks SQL dialect, focusing on its syntax, semantics, and features. This chapter will provide a general overview of the language and its capabilities.

The fourth chapter will cover the design of the prototype. We will discuss the technologies used and describe the overall architecture of the created prototype.

The fifth chapter will focus on implementing the prototype for analyzing Databricks SQL, which will retrieve data flow information from the scripts and store this information in the Manta. We will also explain the implementation of our tool and its key components and describe the grammar used for generating the parser in this chapter.

The sixth and final chapter will conclude the thesis with a discussion of the testing of our prototype. This chapter will describe our approach to testing and various types of tests created for the prototype.

To summarize, this thesis seeks to enhance the understanding of the SQL language dialect used in the Databricks ecosystem, its syntax, and semantics, as well as extend the Manta tool with a new module for Databricks SQL analysis. By studying the Databricks dialect of SQL, we aim to extend the capabilities of the Manta tool and enable it to retrieve information about data flow in Databricks SQL scripts through static analysis.

Chapter 1

Preliminaries

This chapter provides a comprehensive overview of the theoretical concepts used in the static analysis of Databricks scripts. This chapter aims to provide a solid foundation for the rest of the thesis and help the readers understand the concepts and techniques used in our work. The chapter will cover several important topics in detail, including basic grammar and automata theory, lexical analysis, syntax analysis and parsing, abstract syntax trees and their resolution, the ANTLR library, and data flow graphs.

1.1 Automata and grammar theory

The first section of this chapter focuses on basic grammar and automata theory. We will introduce key concepts such as formal languages, grammars, and automata and discuss their relevance to the analysis of scripts. The definitions and theory are based on [3].

First, we present the fundamental concepts of strings, languages, and their notations in the context of formal language theory.

Definition 1 (Alphabet)**.** *An alphabet Σ is a finite, non-empty set of symbols.*

Definition 2 (String)**.** *Let Σ be an alphabet. A string w is a finite sequence of symbols from Σ. The length of a string is the number of symbols it contains and is denoted by $|w|$. The empty string is denoted by ε and has a length of 0.*

Definition 3 (Concatenation)**.** *Let $x = x_1x_2 \ldots x_m$ and $y = y_1y_2 \ldots y_n$ be two strings over an alphabet Σ. The concatenation of x and y, denoted by xy, is the string $z = x_1x_2 \ldots x_my_1y_2 \ldots y_n$, where z has length $m + n$.*

Definition 4 (Power)**.** *Σ is an alphabet. Let $\Sigma^0 = \varepsilon$. Then $\Sigma^k = \Sigma^{k-1}.\Sigma$.*

See that $\Sigma = \Sigma^1$.

Definition 5 (Kleene star)**.** *A Kleene star of an alphabet Σ^* is defined as $\bigcup_{i=0}^{\infty} \Sigma^i$.*

We denote $\bigcup_{i=1}^{\infty} \Sigma^i$ by Σ^+ and call this a Kleene plus of an alphabet.

The Kleene star of an alphabet Σ^* is a set of all strings over this alphabet. Kleene plus Σ^+ is a set of all non-empty strings over the Σ alphabet.

We now formally define a language.

Definition 6 (Language)**.** *Given an alphabet Σ, a language L is a (possibly infinite) set of strings over Σ. Formally, $L \subseteq \Sigma^*$.*

Since languages are sets, we may perform standard set operations such as union, intersection or set minus.

In computer science, multiple formalisms are used to define languages. The most commonly used ones are grammars, automata, and regular expressions. Grammar is a mathematical model for generating a language and is defined as follows.

Definition 7 (Grammar)**.** *Grammar is a quadruple (Σ, N, R, S), where:*

- *Σ is the set of terminal symbols, also known as the alphabet, representing the final set of elements in the language, such as letters, symbols, or tokens.*

- *N is the set of non-terminal symbols, which are symbols that are intended to be rewritten by a rule in the grammar. It is important to note that Σ and N are disjoint sets.*
- *R is the set of grammar rules describing how non-terminal symbols can be rewritten.*
- *S is the starting symbol*

Usage of the rewriting rule is called derivation and is denoted by $\Rightarrow$. $\beta A \gamma \Rightarrow \beta \alpha \gamma$ only if there is a rule $A \rightarrow \alpha \in R$. We denote by $\Rightarrow^*$ any number of chained derivations.

Definition 8 (Sentential form)**.** *A string s over $N \cup \Sigma$ is a sentential form in $G = (\Sigma, N, R, S)$, if $S \Rightarrow^* s$. If $s \in \Sigma^*$, s is called a sentence or a word in the language L defined by grammar G.*

When we apply derivation on a sentential form, we may rewrite any non-terminal symbol. However, we define left-most (resp. right-most) derivation as always rewriting the left-most (resp. right-most) non-terminal symbol. Assuming rules are numbered, we can write down numbers of rules used by left-most derivation; we call this left parse. The right parse is then defined analogously.

1.1.1 Grammar Types

There are different types of grammars based on the type of rules used, and two of the most important types for us are regular and context-free grammars.

Regular grammars have rules that are in the form of $A \rightarrow bB$, $A \rightarrow b$, or $S \rightarrow \varepsilon$, where $A, B \in N$, $b \in \Sigma$. S is a starting symbol and is not on the right side of any rule, and ε is an empty string. Regular grammars are often used in lexical analysis to generate tokens of a source code in a programming language.

On the other hand, context-free grammars have rules in the form of $A \rightarrow \alpha$, where $A \in N$, and α is any sequence of terminals and non-terminals. Every regular grammar is context-free, as regular grammars are a subset of context-free grammars. This thesis will use context-free grammar to define the grammar of the Databricks SQL dialect.

Context-sensitive grammars are characterized by their rules, which are in the form of $\alpha N \beta \rightarrow \alpha \gamma \beta$, where α and β are any strings, N is a non-terminal symbol, and γ is a non-empty string of symbols. Context grammar can include a rule in the form of $S \rightarrow \varepsilon$, as long as S is not on the right side of any other rule.

Unrestricted or unlimited grammars, also known as Turing-complete grammars, have the most general form of rules – $\alpha \rightarrow \beta$, where $\alpha \in (\Sigma \cup N)^+$ and $\beta \in (\Sigma \cup N)^*$. These grammars are equivalent in power to Turing machines and can describe any computable language.

In conclusion, while unrestricted grammars have the greatest expressive power, they are not practical for most language processing tasks, as the parsing algorithms for unrestricted grammars are not feasible for real-world applications. At the same time, context-sensitive grammars are more powerful than context-free grammars but also more complex to parse. For this reason, context-free grammars are often the preferred choice for describing programming languages and other formal languages in computer science.

Let us define a parse tree.

Definition 9 (Parse tree)**.** *Let $G = (\Sigma, N, R, S)$ be a grammar. The tree is a parse tree for G if it meets the following*

1. *Every inner node is labelled by $A \in N$.*

2. *Every leaf is labelled by either $a \in \Sigma$ or ε. If the leaf is labelled by ε, then it must be the only child of its parent.*

3. *If there is a node labeled A with children, in sequence, $X_1, X_2, X_3, \ldots, X_k$, then $A \to X_1X_2X_3 \ldots X_k \in R$*

For grammars, we will define ambiguity, as this property is quite important for parsing because it introduces non-determinism of the parsing task.

Definition 10. *Grammar is ambiguous if there exists a string for which there is more than one possible parse tree in this grammar.*

This can occur when the grammar has productions that can be applied in multiple ways, leading to multiple possible derivations for a single input string. For example, consider the grammar $G = (\{a, +, *\}, \{S\}, R, S)$:

$$R = \{S \to S + S \mid S * S \mid a\}$$

This grammar generates arithmetic expressions consisting of additions and multiplications of the symbol a. The ambiguity arises for example when we consider the expression $a{+}a *a$. This expression can be parsed in two different ways; see the Figure 1.1

In the first parse tree, we first used the addition rule and then on the second S the multiplication rule is used, while in the second parse tree, the multiplication is used as first, then on the first S the addition rule is used, leading to a different parse tree. Thus, the grammar G is ambiguous.

Ambiguity in grammar can lead to difficulties in designing and implementing parsers, as it requires disambiguation techniques to be used in order for the parser to be deterministic. These techniques include modifying the grammar itself to eliminate ambiguity or using disambiguation rules in the parsing process. Additionally, ambiguity can cause confusion and unexpected behaviour in programming languages or other applications that rely on unambiguous grammars for correct parsing and interpretation.

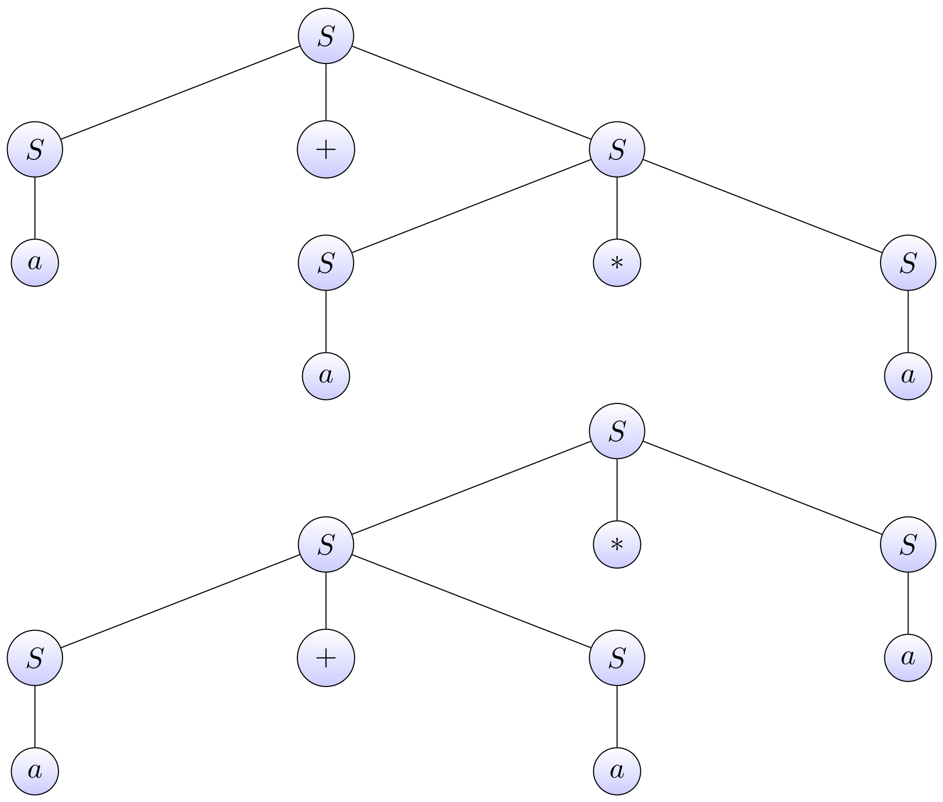


Figure 1.1: Two parse trees for the expression $a + a * a$ in the grammar G

1.1.2 Automata

Automata are abstract mathematical models used in computer science to accept languages and recognize strings in those languages. Several types of automata exist, including finite automata, push-down automata, and Turing machines. We will discuss the first two models' basic definitions and characteristics, including deterministic and non-deterministic versions, and their applications in language processing tasks such as parsing.

The simplest automaton is a finite automaton, which can be defined as follows.

Definition 11 (Finite automaton)**.** *Finite automaton is defined as a tuple* $M = (Q, \Sigma, \delta, q_0, F)$*, where:*

- Q *is a finite set of states*

- *Σ is a finite set of input symbols, also known as the alphabet*
- *$\delta : Q \times \Sigma \to 2^Q$ is the transition function*
- *$q_0 \in Q$ is the initial state*
- *$F \subseteq Q$ is the set of final states, also known as accepting states*

Informally, a non-deterministic finite automaton is one in which there may be multiple states q' such that $\delta(q, a) = q'$ for a state q and input symbol a.

A finite automaton is called deterministic if, for each state $q \in Q$ and each input symbol $a \in \Sigma$, there is at most one state $q' \in Q$ such that $\delta(q, a) = q'$. I.e., the definition of the delta function is $Q \times \Sigma \to Q$.

A string $w \in \Sigma^*$ of length n is said to be recognized by the automaton if there is a sequence of states $q_0, q_1, \ldots, q_n$ such that q_0 is the initial state, $q_n \in F$ is a final state and $\delta(q_{i-1}, w_i) = q_i$ for each $1 \leq i \leq n$.

Push-down automata (PDA) are similar to finite automata, but they have an additional push-down store that allows them to recognize context-free languages. A push-down automaton can be defined formally as follows.

Definition 12 (Push-down automaton). *Push-down automaton is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$, where:*

- *Q is a finite set of states*
- *Σ is a finite set of input symbols, also known as the alphabet*
- *Γ is a finite set of push-down store symbols*
- *δ is a finite subset of $Q \times (\Sigma \cup \varepsilon) \times \Gamma * \to Q \times \Gamma^*$ is the transition function*
- *$q_0 \in Q$ is the initial state*
- *$Z \in \Gamma$ is the initial push-down store symbol*
- *$F \subseteq Q$ is the set of final states, also known as accepting states*

The PDA can accept a string in two ways after reading the entire input string: either by reaching a final state or by having an empty push-down store. The accepting method is declared when declaring the specific PDA by either specifying final states, then the automaton accepts by reaching a final state, or by making the set of final states empty and thus accepting by emptying the push-down store.

A configuration of a PDA consists of a state, a push-down store content, and the unread part of the input string.

An accepting configuration of a push-down automaton is a configuration that satisfies the conditions for the PDA to accept a string given the selected method described above.

An accepting configuration with a final state occurs when the PDA reaches a designated final state, often denoted as q_f, after processing the entire input string. The final state must be specified as part of the definition of the PDA and is often used to represent that the input string has been successfully processed. An accepting configuration with an empty push-down store occurs when the PDA reaches the end of the input string, and the push-down store is empty.

Formally, let M be a push-down automaton and let $w \in \Sigma^*$ be a string. We say that M accepts the string w if there is a sequence of configurations $c_0, c_1, \ldots, c_n$ such that $c_0 = (q_0, \varepsilon, w)$, c_n is valid accepting configuration and $(c_0, \ldots, c_n)$ is a valid transition chain of M, meaning, that every configuration is a result of a transition function used on the configuration before.

Push-down automata can be either deterministic or non-deterministic, just like finite automata. Informally, the difference is that in a non-deterministic push-down automaton, there can be multiple possibilities for the next configuration based on a single input symbol (or epsilon) and push-down store content.

1.2 Language processing

This section provides a brief overview of the language analysis process, which consists of several stages: lexical analysis, syntax analysis, semantic analysis, and further processing.

The lexical analysis breaks down the input text into tokens or meaningful units, such as words, numbers, and punctuation marks. The main goal of this stage is to identify and categorise tokens based on their types and properties, preparing them for the subsequent stages of analysis. The details of the lexical analysis will be elaborated upon section 1.3.

Syntax analysis is concerned with creating a hierarchical structure that represents grammatical relationships between tokens. This stage's goal can be to ensure that the input text conforms to the rules of the language's grammar, which is not an issue in our case, as we expect only valid input scripts. Syntax analysis can, however, also creates a structure that can be used later in the language analysis. This structure, a parse tree or a syntax tree, facilitates the examination of the input text's structure and organisation. Syntax analysis will be covered by section 1.4.

The semantic analysis focuses on understanding the meaning of the input text by examining the relationships and dependencies between its parts. The goal of this stage can be to ensure that the input text is both logically and semantically correct, considering context, among other aspects. In our case, we will resolve the syntax tree, trying to determine which reference is referencing what object.

The last step, further processing, is, in our case, a data flow analysis. We create a data flow graph from the resolved syntax tree. We delve into this process in section 1.7.

```
fileName = inputName + ".sql"
```

Code snippet 1: Example of assingment

1.3 Lexical analysis

This section covers lexical analysis, which involves the process of breaking down the input into its constituent parts, known as tokens. We will discuss the role of lexical analysis in language processing and explain how it is used to tokenise the input.

The purpose of the first step, lexical analysis is to identify the lexical symbols, encode them, hide redundant symbols (comments, whitespaces etc.) and determine token value [4].

The lexical analyser reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyser produces as output a token of the form `<token-name, attribute-value>` *that it passes on to the subsequent phase, syntax analysis.* [5]

For example, statement in Code snippet 1 would translate to following tokens `<id, "fileName"> <=> <id, "inputName"> <+> <stringLiteral, ".sql">`. We see that not every token has to have a value; however, usually, it can be useful to refer to tokens like `<=>` or `<+>` by some name. Then, for example, we can distinguish between '=' as equals and '=' as assign and interchange these two symbols in the lexer more easily. With this, the tokens for the same statement can be: `<id, "fileName"> <assign,'='> <id, "inputName"> <plus,'+'> <stringLiteral, ".sql">`

Identifier tokens are of particular interest to us. The attribute value of the token is the actual name, character sequence identifying an entity in the program. When needed, i.e. in compilers, the identifiers can be stored in the symbol table directly during lexical analysis. Attribute values of different

```sql
SELECT * FROM schema1.as;
SELECT from AS col FROM t1;
```

Code snippet 2: SELECT example

identifier tokens are then usually pointers to the symbol table [5]. In this thesis, we use a different approach. The tokens refer to the original input sequence of characters and thus know exactly the original sequences of characters they represent. What they are referring to is then solved during the semantic analysis.

During lexical analysis, keywords are also identified. Keywords are a finite subset of all possible identifiers. In some languages, keywords cannot be used as identifiers.

There are two ways to implement this differentiation. First, we can distinguish different keywords by different final states of an automaton used for lexical analysis. The second option is to check in the keywords table whether this identifier is a keyword after the identifier is processed [4]. The second approach generally produces smaller automata; however, it is slower as it adds the overhead of checking the table of keywords after processing every identifier or keyword.

However, in SQL, we distinguish reserved keywords, which can not be used as an identifier and non-reserved keywords, which may be used as an identifier. Consider examples in Code snippet 2. There we can see `as` and `from` used as column names, but they are also SQL keywords. The example was highlighted using `minted` package [6], and we see that it used the wrong colour for `from` and `as` when they were used as column names – this is a perfect example of this issue - without a context it is sometimes impossible to determine whether a sequence is used as an identifier or a keyword. This raises a problem in the lexical analysis phase, as the lexical analyser is not able to work with context; therefore, we postpone this determination to the syntactical analysis.

Theoretically, the lexical analyser can be represented as a final automaton that recognizes lexical symbols – tokens. This approach is used, for example, in [4] or in [5]. However, in practice, this approach may have issues, such as comments being recursive and requiring push-down automata instead of final automata. Also, in this thesis, we retain comments and whitespaces in the token sequence for later restoration of the script.

1.4 Syntax analysis and parsing

The third section focuses on syntax analysis and parsing. This step involves analysing the input based on its syntax and constructing a parse tree to represent the structure of the input. We will discuss the different types of parsers, including top-down and bottom-up parsers, and the pros and cons of each approach.

Syntax analysis, also known as parsing, is the process of analysing the syntactic structure of a sentence (in our case, a script) to understand its meaning. Parsing aims to create a parse tree, a hierarchical representation of the sentence, showing the relationships between its components.

There are three main approaches to parsing: universal, top-down and bottom-up. Universal parsing methods include the Cocke-Younger-Kasami algorithm and Earley's algorithm. They can parse any grammar; however, they are usually too inefficient to use in production analysers. Top-down parsing starts with the highest level of the parse tree – starting symbol S – and works its way down to the leaves – tokens, while bottom-up parsing starts with the leaves and works its way up to the root. The parser scans the input from left to right, one symbol at a time regardless of the approach used [5].

A bottom-up parser starts by reading the input symbols from left to right and then tries to reduce them to non-terminals of the grammar until it reaches the start symbol of the grammar. In order to do so, it utilizes a push-down store to keep track of the symbols that have been read so far and already reduced-to non-terminals.

A PDA can be used to implement a bottom-up parser. The bottom parser PDA for a grammar $G = (\Sigma, N, R, S)$ is a tuple $(Q = \{q, q_f\}, \Sigma, \Gamma = (\Sigma \cup N \cup \{Z\}), \delta, q_0 = q, Z, F = \{q_f\})$ and accepts by reaching a final state q_f. The transition function δ is defined as follows:

1. $\delta(q, a, e) = \{(q, a)\}$ for all $a \in \Sigma$. These transitions cause input symbols to be shifted on top of the push-down store; we call them a shift.

2. If $A \to \alpha$ is in R, then $\delta(q, \varepsilon, \alpha)$ contains (q, A). These transitions say that if the top of the push-down store contains a sequence of symbols that can be reduced to a non-terminal A by using a production $A \to \alpha$, then the PDA can make a reduction step and replace these symbols with the non-terminal A. These transitions are called reductions.

3. $\delta(q, \varepsilon, ZS) = (q_f, \varepsilon)$. This transition accepts the input when the entire input has been parsed, and the push-down store contains only the start symbol S. This transition is called acceptance.

By defining the transition function δ, the PDA can recognize the language generated by a context-free grammar $G(\Sigma, N, P, S)$ using a bottom-up parsing strategy, producing the right parse. The PDA shifts input symbols onto the push-down store, reduces the push-down store symbols to non-terminals, and finally accepts the input by reducing the push-down store to the start symbol S [1] [7].

On the other hand, a top-down parser is a type of parser that constructs a parse tree from the root to the leaves. It starts by trying to expand the grammar's start symbol into a sequence of terminals and non-terminals, then recursively expands each non-terminal until the entire input string is derived.

[1]The top of the push-down store is maintained on the right (as opposed to the general left).

If the top of the push-down store is the terminal symbol, it tries to match it against the first unprocessed symbol of the input.

A PDA can also be used to implement a top-down parser. Top down parser PDA for a grammar $G = (\Sigma, N, R, S)$ is a tuple $(Q = \{q\}, \Sigma, \Gamma = (\Sigma \cup N), \delta, q_0 = q, Z = S, F = \{\})$ and accepts by an empty push-down store. The transition function δ for a top-down parser is defined as follows:

1. $\delta(q, e, A) = (q, \alpha)$ for all $A \to \alpha \in P$. These transitions push the right-hand side of the production rule $A \to \alpha$ onto the push-down store when the top of the push-down store contains the non-terminal A.

2. $\delta(q, a, a) = (q, e)$ for all $a \in \Sigma$. These transitions consume input symbols from the input string and the push-down store when the top of the push-down store contains the terminal symbol a. a transition like this is called the match.

The top-down parser will produce the left parse of a given input if we output the respective rule on each expansion [7].

Both these approaches are, as defined, non-deterministic. More is needed for their usage in the compilers; therefore, deterministic variants are explored.

There are LR parsers or Generalized LR grammars for bottom-up parsing, which parse in linear to cubic time. This approach is not ideal for parsing because it accepts ambiguous grammar [8] – which can be seen as an advantage. In practise, it can be problematic, as the programmer can not rely on one particular parse tree. Thus accepting unambiguous grammars only leads to more sustainable code.

$LL(k)$ analysis is a deterministic type of top-down parsing that uses a push-down store automaton to construct a parse tree for a given input string. In $LL(k)$ analysis, the expansion rule is selected based on the content of a look-ahead window of to-be-parsed symbols of length k. $LL(k)$ is more powerful than $LL(k-1)$. The rules to use when expanding are given by a precomputed parse table. Construction of the $LL(k)$ analyser involves building this table.

$LL(1)$ analysis is a specific case of $LL(k)$ analysis where $k = 1$. It is the least powerful among all the $LL(k)$ analysis variants but has a straightforward implementation by recursive descent. Two functions are used to construct the $LL(1)$ parse table: `FIRST` and `FOLLOW`.

For $LL(1)$ analysis, the `FIRST` function is defined for any string $\alpha \in (N \cup T)^*$. It is a set of terminal symbols and possibly ε that are the first symbols in sentences generable from α by rules from the grammar. Formally,

$$\texttt{FIRST}(\alpha) = \{a : \alpha \Rightarrow^* a\beta; a \ \in T; \alpha, \beta \in (N \cup T)\} \cup \{\varepsilon : \alpha \Rightarrow^* \varepsilon; \alpha \in N^*\}.$$

The `FOLLOW` function is defined for each non-terminal symbol A. `FOLLOW`(A) is a set of terminal symbols, and possibly ε, that follow A in sentential forms generated by the grammar. Formally,

$$\texttt{FOLLOW}(A) = \{a : S \Rightarrow^* \alpha A\beta; a \ \in \texttt{FIRST}(\beta)\}.$$

In the situation, when $\varepsilon \in$ `FIRST`(β), for instance, in the case of S, where $S \Rightarrow^* \alpha$, where α is the entire input sequence, $\varepsilon \in$ `FOLLOW`(S).

$LL(k)$ parsers require grammars to meet specific criteria, including being left-recursion free, unambiguous, and having no `FIRST -- FIRST, FIRST -- FOLLOW` conflicts. For $LL(1)$, we mean by those conflicts, that no non-terminal can have the same symbol in two or more `FIRST` sets of its rules and, if there is an ε in the `FIRST` of a rule, the `FOLLOW` can't contain any of the symbols in the `FIRST` sets of the nonterminal's rules. For $LL(k)$, `FIRST` and `FOLLOW` sets contain k-grams, `FIRST -- FIRST, FIRST -- FOLLOW` conflicts are then defined analogously.

Failure to meet these criteria will result in conflicts and ambiguities in the parse table, making the parser unable to parse certain input strings deterministically.

1.5 ANTLR library and LL(*) analysis

In this thesis, the LL(*) parsing method and ANTLR library are used. ANTLR is a popular library for implementing parsers. This section explains how it can be used to implement a parser for a specific language. We also discuss the key concepts and techniques used in ANTLR and provide a brief overview of its capabilities.

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator widely used for constructing parsers, interpreters, compilers, and other language processing tools. ANTLR can generate parsers that can handle a wide range of input languages.

The ANTLR parser generator uses a domain-specific language to describe the grammar of the input, which is then used to generate a parser that can recognize words from the language generated by the grammar. The parser generator is based on the LL(*) parsing algorithm, a $LL(k)$ parsing variant that uses a lookahead of arbitrary length to parse more complex languages.

ANTLR supports a wide range of target languages, including Java, Python, C#, and JavaScript, which makes it a popular choice for developing language processing tools in various programming environments. It is used, for example, as a part of the Google app engine (Python), IBM Tivoli Identity Manager, BEA/Oracle WebLogic, Yahoo! Query Language, Apple Keynote, Oracle SQL Developer IDE or NetBeans IDE [8].

In the example in Code snippet 3, we define a grammar for a simple language that includes if statements and simplified boolean expressions[2]. We define a rule for a program consisting of zero or more statements and a rule for a statement, which can be an if statement with an optional else clause or a print statement. We also define a rule for a simplified expression, which can be either true, false, or an identifier (ID). Using ANTLR to generate a parser for this grammar would result in a parser that can recognize and parse input

[2]The grammar in Code snippet 3 is ambiguous, as this is a simplified example

```
grammar SimpleLanguage;

program: statement*;
statement
    :   'if' expression 'then' statement ('else' statement)?
    |   'print' '(' expression ')' ';'
    ;
expression: 'true' | 'false' | ID;
ID: [a-zA-Z]+;
```

Code snippet 3: An example of a simple parser grammar

written in this language, allowing us to perform language processing tasks such as data flow analysis.

1.5.1 LL(*) Parsing

LL(*) parsing is a top-down parsing algorithm introduced in [8]. In this subsection, we explain the base idea of the algorithm. It uses a predictive parsing table to determine the following production rule to apply during parsing based on the current non-terminal symbol and the lookahead symbol(s). In LL(*) parsing, the predictive parsing table is constructed using deterministic finite automata (DFAs) generated for each grammar non-terminal symbol and for places where a similar decision is made (bracketed alternation, etc.).

During parsing of the grammar, the LL(*) algorithm starts with an initial non-terminal symbol. Then it uses the DFA for that symbol to determine the following production rule to apply based on the current lookahead symbol(s). The use of DFAs in ANTLR's implementation of LL(*) parsing allows for fast and efficient parsing of input languages. The deterministic nature of the automata ensures that the algorithm can proceed quickly through the input stream without having to backtrack excessively.

To construct these DFAs, LL(*) employs augmented transition networks (ATNs), a form of a directed graph representing context-free grammars. An

ATN consists of states connected by transitions, where each state represents a position within a production rule, and transitions correspond to grammar symbols (either terminal or non-terminal). ATNs are "augmented" because they include epsilon transitions, which are transitions that can be taken without consuming any input symbols. Epsilon transitions represent non-deterministic choices within the grammar, such as when multiple production rules share the same prefix.

To create a DFA for each non-terminal symbol in the grammar, the LL(*) algorithm first constructs an ATN representation of the grammar. Then, the algorithm converts the ATN to a DFA through ATN simulation. The ATN simulation involves traversing the ATN based on the lookahead symbols and exploring all possible texts to predict the correct production rule to apply during parsing. During this process, the algorithm resolves any non-determinism that arises due to epsilon transitions and merges equivalent states to construct a deterministic automaton. This process may fail on recursive rules, as they introduce a loop to the automaton. To handle this issue algorithm limits the number of recursive invocations to any particular rule. Another strategy to optimise the algorithm is a heuristic termination of DFA construction upon discovering recursion in multiple alternatives. In such case, ANTLR falls back on LL(1) lookahead for a non-terminal with backtracking or other predicates.

1.6 Abstract syntax tree

The sixth section explains what an abstract syntax tree is and how it is used to represent the structure of the input. We also discuss the process of resolving abstract syntax trees during the analysis of scripts.

The abstract syntax tree (AST) is a data structure that represents the syntactic structure of the source code in a tree-like form. It is an essential component of many program analysis techniques. The abstract syntax tree captures the hierarchy and relationships between the various elements of

```sql
SELECT id, name
FROM students
WHERE age > 30;
```

Code snippet 4: Example of a simple `SELECT` statement

a program, such as statements, their clauses, expressions, function calls or their parameters.

The AST does not have an exact definition, as there are multiple possibilities for how the AST for the same input can look. The AST design is decided by a programmer, and it varies based on the purpose of the AST.

For example, consider the SQL statement in Code snippet 4. The corresponding AST can be represented both as Figure 1.2 and Figure 1.3. In the first AST, the root node represents the `SELECT` statement, with three child nodes representing the selected columns and the table from which they are selected. The `WHERE` clause is represented by a child node with an operator and a value. We can see that the second AST is much more detailed, using generic nodes like `AST-STATEMENT`.

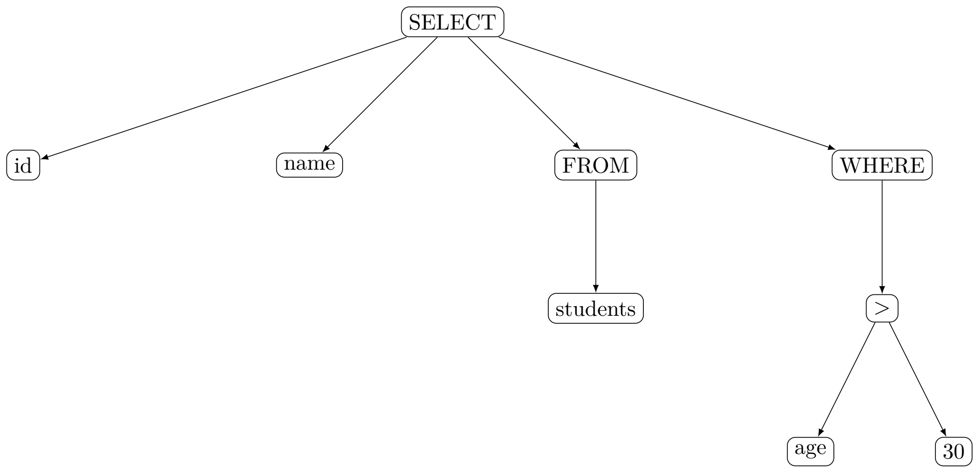


Figure 1.2: AST tree for the SQL statement: `SELECT id, name FROM students WHERE age > 30;`

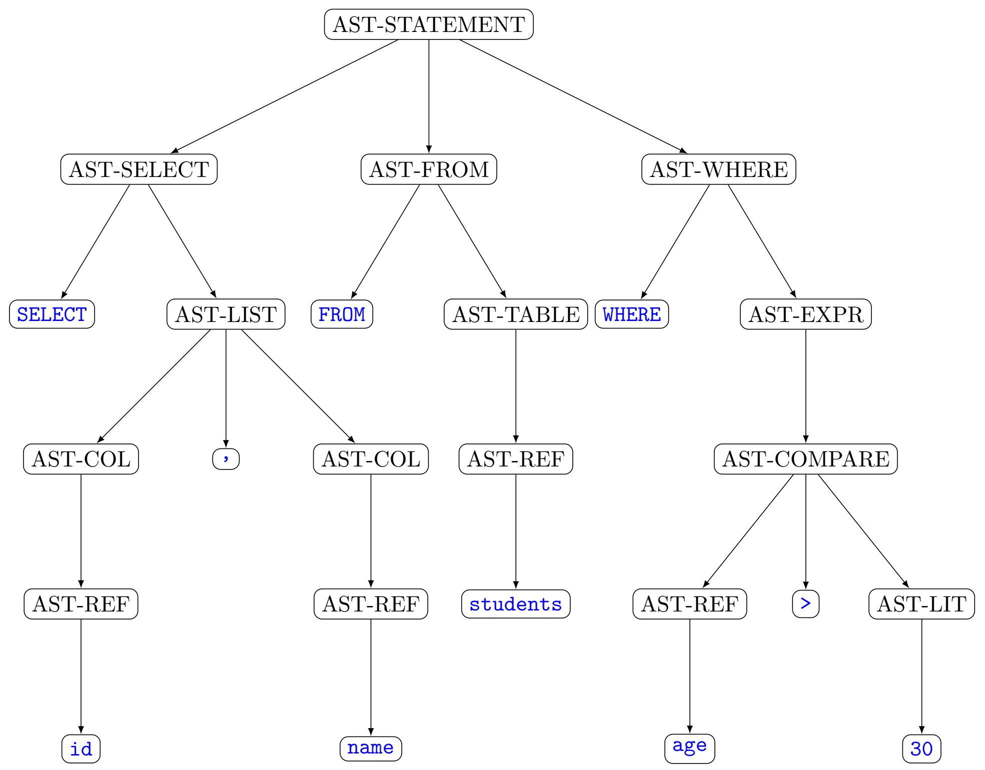


Figure 1.3: Detailed AST tree for the SQL statement: `SELECT id, name FROM students WHERE age > 30;`

The AST is usually generated by a parser that reads the source code and produces a tree structure representing the program's syntax. Each node in the tree represents a different syntactic construct, such as a variable declaration, function call, or loop statement. The nodes are connected by edges that represent the syntactic relationships between them.

The AST is often used as the basis for program analysis techniques because it provides a high-level view of the program's structure. From the AST we can extract information about the program, such as all the variable declarations, function calls, and control flow statements, as they may use the same node type. The AST can be used to perform code optimizations, such as loop unrolling and constant folding. The AST typically omits unnecessary information from the input.

1.7 Data flow analysis

The final section covers the data flow analysis and the data flow graph, which represent the flow of data within a system. We explain what data flow graphs are and how they can be used to analyse the flow of data in a system.

A data flow graph is not defined precisely, as its design may vary with different tools. Because in this thesis, we are extending the Manta tool, we describe how the data flow graph is designed in Manta.

A data flow graph is an oriented graph that consists of nodes and oriented edges. The nodes represent the actual data, and two nodes can be connected by the oriented edges representing data flows between them. We define two types of data flow edges: direct flows and filter flows.

Direct data flows indicate that the source nodes directly participate in the data origin of target nodes. For example, let's consider table `t2` with columns `a,b` and statement in Code snippet 5. We want to have direct flow from column `a` of table `t1` to column `a` of table `t2`, similarly with `b` column.

Filter data flows, on the other hand, impact the content of the target node without directly contributing to it. As an example, see again Code snippet 5. Here the `c` column affects which rows are selected, so it has an impact on the result of the `SELECT` statement an thus the values inserted in table `t2`.

For the example statement in Code snippet 5, the detailed data flow graph of this statement can be seen in Figure 1.4. However, for the customer, a graph in Figure 1.4 is too detailed. For the analysis purpose, we need information about all the result sets, and how they interact, how the expressions are constructed, in the final data flow graph, it may not be as important. Therefore we may contract some of the edges and delete unimportant nodes. We will call this filtered graph; see an example for the example statement in Code snippet 5 in the Figure 1.5.

```
INSERT INTO t2
SELECT a, b FROM t1 WHERE c > 100;
```

Code snippet 5: An example of INSERT statement

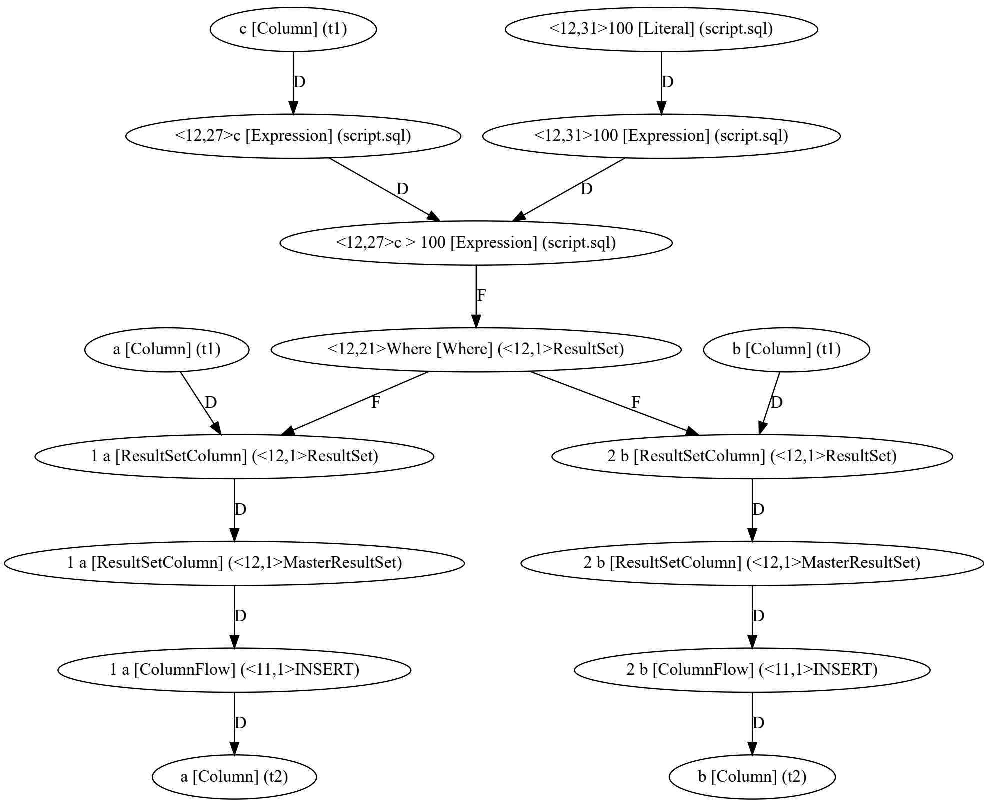

c [Column] (t1)
<12,31>100 [Literal] (script.sql)
D
D
<12,27>c [Expression] (script.sql)
<12,31>100 [Expression] (script.sql)
D
D
<12,27>c > 100 [Expression] (script.sql)
F
a [Column] (t1)
<12,21>Where [Where] (<12,1>ResultSet)
b [Column] (t1)
D
F
F
D
1 a [ResultSetColumn] (<12,1>ResultSet)
2 b [ResultSetColumn] (<12,1>ResultSet)
D
D
1 a [ResultSetColumn] (<12,1>MasterResultSet)
2 b [ResultSetColumn] (<12,1>MasterResultSet)
D
D
1 a [ColumnFlow] (<11,1>INSERT)
2 b [ColumnFlow] (<11,1>INSERT)
D
D
a [Column] (t2)
b [Column] (t2)

Figure 1.4: Detailed data flow graph

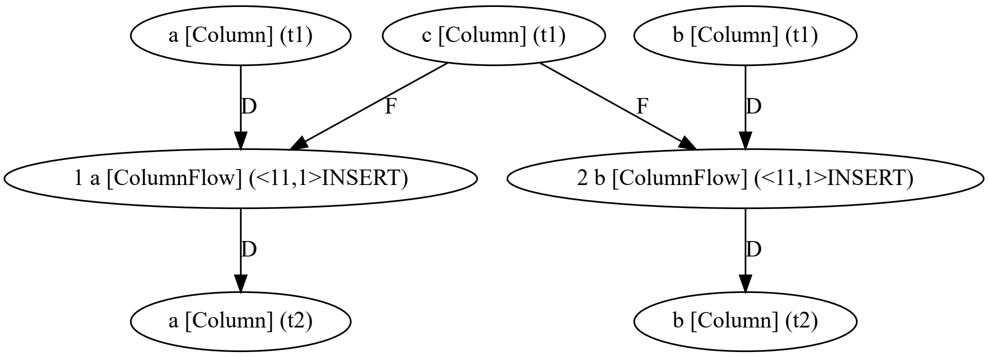


Figure 1.5: Filtered data flow graph

Chapter 2

Databricks System analysis

This chapter aims to analyse and introduce the Databricks system. Databricks runtimes consist of core components that run on Databricks clusters. In addition to Apache Spark, Databricks Runtime also includes features and updates that substantially improve big data analytics' usability, performance, and security.

2.1 Data Lake vs. Data Lakehouse vs. Data Warehouse

The philosophy of Databricks is built on the idea of data lakes. *A data lake is a central location that holds a large amount of data in its native, raw format. Compared to a hierarchical data warehouse, which stores data in files or folders, a data lake uses a flat architecture and object storage to store the data. Object storage stores data with metadata tags and a unique identifier, which makes it easier to locate and retrieve data across regions and improves performance. By leveraging inexpensive object storage and open formats, data lakes enable many applications to take advantage of the data.* [9]

One of the most important advantages of data lakes is that they are an open format, so one can avoid lock-in to a proprietary system. Also, Data lakes are highly durable and low-cost. Data lakes store large amounts of structured, semi-structured, and unstructured data. They can contain everything from relational data to JSON documents to PDFs to audio files.

Databricks implements a data lakehouse concept. *Data Lakehouse is an architecture that enables efficient and secure Artificial Intelligence (AI) and Business Intelligence (BI) directly on vast amounts of data stored in Data Lakes. It has specific capabilities to efficiently enable both AI and BI on all the enterprise's data at a massive scale. Namely, it has the SQL and performance capabilities (indexing, caching, MPP processing) to make BI work fast on data lakes. It also has direct file access and direct native support for Python, data science, and AI frameworks without ever forcing it through a SQL-based data warehouse. The key technologies used to implement Data Lakehouses are open source, such as Delta Lake, Hudi, and Iceberg.* [10]

2.2 Delta Lake

Delta Lake is an open-source project that enables building a Lakehouse architecture on top of data lakes. *Delta Lake is the optimized storage layer that provides the foundation for storing data and tables in the Databricks Lakehouse Platform. Delta Lake is open-source software that extends Parquet data files with a file-based transaction log for ACID transactions and scalable metadata handling. All tables on Databricks are Delta tables by default. Whether you're using Apache Spark DataFrames or SQL, you get all the benefits of Delta Lake just by saving your data to the lakehouse with default settings.* [11]

Delta Lake validates data on write so the user can set requirements every write must meet. Metadata used to reference the table is added to the metastore in the declared schema or database. Data and table metadata are saved to a directory in cloud object storage. There is an option to create Delta tables by directly interacting with directory paths using Spark APIs. Some new features that build upon Delta Lake store additional metadata in the table directory. Still, all Delta tables have a directory containing table data in the Parquet file format and a sub-directory `_delta_log` that contains metadata about table versions in JSON and Parquet format.

2.3 Databricks SQL and Databricks Runtime

This subsection describes the difference between Spark SQL and Databricks SQL. Databricks was built on top of Apache Spark [12].

While Databricks SQL is an optimized computing environment, Spark SQL is a Spark module for structured data processing. of Apache Spark APIs that enable data processing through SQL queries and DataFrame API [13]. Spark SQL has replaced the Spark RDD API in Spark 2.x and supports SQL queries and the DataFrame API for programming languages such as Python, Scala, R, and Java.

SQL queries executed in Databricks Runtime closely resemble the open-source Apache Spark functionality, with some built-extension protocols for Delta Lake and proprietary Databricks features [14].

There are three computing options for the Databricks system, Databricks SQL, Databricks Runtime and Delta Live Tables. All computing options provide slightly different semantics and syntax.

Delta Live Tables pipelines are multiple pipelined notebooks with some special SQL syntax [15]. These pipelines offer to use Delta Live Views and Tables that only exist during the run time of the Delta Live Table pipeline. There is an option to make the results a permanent table by publishing it by setting the target property in the configuration. This action can only publish tables; views are not published to the metastore.

SQL executed with Delta Live Tables inherits syntax and semantics from the Databricks Runtime but adds some exclusive keywords and functions for Delta Live Tables.

Databricks SQL run on SQL warehouses generally abides by ANSI standards. However, some features supported by Databricks SQL do not work when run in Databricks notebooks against Databricks Runtime compute, including HiveQL syntax, variable declaration and reference, and DButils widgets.

Databricks SQL offers several features, including a schema browser and a SQL editor with the possibility of visualizations. Databricks SQL provides some configuration parameters; however, according to the documentation, they should not impact lineage, only the system's performance; therefore, they are not further examined [16].

2.4 SQL in Databricks

Databricks SQL is a system for SQL on the Databricks platform. It uses Spark SQL as the base for its SQL dialect.

SQL as a language can be found in three places: in SQL cells inside a notebook and in queries, and in Delta Live pipelines. However, last two options use not Databricks SQL environment, but Databricks Runtime. This brings slight differences in supported syntax.

2.4.1 Notebooks

Notebooks in Databricks run against Databricks Runtime and support multiple programming languages. Although each notebook has a default language, cells within the notebook can use a different language. The language of a cell can be set by starting the cell with `%<language_name>` or by setting it manually in the user interface. Spark context is a shared resource across languages, and it is possible to reference Spark context variables from SQL. However, there is no current method to set a variable from SQL. There is a feature to have the last result set available in other languages under the `_sqldf` variable

[17]. It has been discovered that these variables can contain code, so it is necessary to analyze the code with variables replaced by their values before performing any analysis. Notebooks in Databricks are created in workspaces and can be shared.

2.4.2 Queries

Databricks SQL Queries run against a certain metastore in the Databricks SQL environment. They are SQL scripts that come with their integrated development environment (IDE) on the Databricks platform. Users can parameterize and schedule these queries using the platform's features. Extraction of information can be done through the API endpoint, which contains information about parameters, schedules, and the query itself; however, there is currently no way to extract the target of the query - what is a default catalog and schema used in the query.

2.5 Data objects

The Databricks Lakehouse organizes data stored with Delta Lake in cloud object storage with relations like databases, tables, and views. The information about the data objects used in Databricks in this section was based on [18] and testing on the Databricks instance. The Databricks Lakehouse architecture combines data stored with the Delta Lake protocol in cloud object storage with metadata registered to a metastore.

The Databricks Lakehouse comprises five main objects: a Catalog, which serves as a grouping of databases; a Database or Schema, which groups objects in a Catalog and contains tables, views, and functions; a table, which is a collection of rows and columns stored as data files in object storage; a view, which is a saved query typically against one or more tables or data sources; and a function, which is saved logic that returns a scalar value or set of rows.

The table metadata is currently persisted in a central metastore accessible by all clusters in every Databricks deployment.

2.5.1 Metastore

A metastore is the top-level container of objects in the Unity Catalog. The Unity catalog is a data governance technology, a metastore and its user management. The metastore stores data assets (tables and views) and the permissions that govern access to them. Databricks account admins can create metastores and assign them to Databricks workspaces to control which workloads use each metastore. The metastore contains all metadata that defines data objects in the lakehouse. Databricks provides several metastore options.

The first option is a Databricks custom solution, a Unity Catalog metastore, where customers can create a metastore to store and share metadata across multiple Databricks workspaces. Unity catalog is managed at the account level. The second option is the default one - Hive metastore. Databricks stores all the metadata for the built-in Hive metastore as a managed service. An instance of the metastore deploys to each cluster and securely accesses metadata from a central repository for each customer workspace. The last option is to run Databricks against a custom external metastore.

For a workspace to use Unity Catalog, it must have a Unity Catalog metastore attached.

Regardless of the metastore used, Databricks stores all data associated with tables in object storage configured by the customer in their cloud account.

2.5.2 Catalog

A catalog is the highest abstraction in the Databricks Lakehouse relational model. They may be seen as a database in other dialects such as PostgreSQL [19]; however, `DATABASE` keyword is a synonym to a `SCHEMA` not to `CATALOG` in the Databricks SQL. Every schema is associated with a catalog. Catalogs exist as objects within a metastore and are used to organize customers' data assets. Users can see all catalogs on which they have been assigned the `USAGE` data permission. Before the introduction of Unity Catalog, Databricks used a two-tier namespace. Catalogs are the third tier in the Unity Catalog namespacing model: `catalog_name.database_name.table_name`. The built-in Hive metastore only supports a single catalog, `hive_metastore`

2.6 Extraction

Manta is currently extracting metadata from the Databricks system through Hive metastore and a Unity catalog. Metadata about data objects, such as tables, schemas and catalogs, needed for the SQL code analysis are stored in the Manta project using a dictionary. There is an interface that can be used for interacting with the dictionary.

The scripts, both notebooks and queries, are being extracted as well; therefore, the analysis did not discover any need for extending the extractor.

Chapter 3

Databricks SQL Dialect

This chapter aims to describe the essential concepts of Databricks SQL dialect. We focus on concepts which can directly influence the flow of the data through the system. This section covers data types, expressions, functions identifiers and statements in the Databricks SQL dialect, as well as the needed preprocessing of the script. The examples and syntax specifications in this chapter were mostly taken from Databricks documentation [20].

3.1 Datatypes

This section discusses the various data types supported in Databricks SQL. These data types are grouped into several classes, including integral numeric types, exact numeric types, binary floating point types, date-time types, simple types and others. Integral numeric types represent whole numbers, including `TINYINT`, `SMALLINT`, `INT`, and `BIGINT`. Exact numeric types represent base-10 numbers and include only one type: `DECIMAL`. Binary floating point types use exponents and a binary representation to cover a large range of numbers, and Databricks SQL supports two types: `FLOAT` and `DOUBLE`. Date-time types represent date and time components and include two types: `DATE` and `TIMESTAMP`. Simple types hold singleton values, and Databricks SQL supports two types: `BINARY` and `BOOLEAN`.

In addition to STRING, which represents textual data, there are two other string data types in Databricks SQL: CHAR and VARCHAR. These data types store fixed-length and variable-length character strings, respectively.

Structs are data types used to group related data and are represented using the STRUCT data type. Maps are data types used to represent key-value pairs and are represented using the MAP data type. For instance, consider the example in Code snippet 6 of creating a struct containing two fields: first_name and last_name, using the STRUCT data type. The result of the statement will be enclosed in curly braces {} and will contain two fields: first_name and last_name. The values of these fields are 'John' and 'Doe', respectively.

```sql
SELECT STRUCT (
    'John' AS first_name,
    'Doe' AS last_name
    ) AS person
```

Code snippet 6: Struct construction example

```sql
SELECT MAP(1, 'one', 2, 'two', 3, 'three') AS number_to_word
```

Code snippet 7: Map construction example

Similarly, a map can be created calling the MAP constructor, see Code snippet 7. In this example, the resulting map is enclosed in curly braces {} and maps the numbers 1, 2, and 3 to the words 'one', 'two', and 'three', respectively.

Maps can also be nested inside other data types, such as structs, as shown in the example in Code snippet 8. The struct in this statement contains three fields in this example: first_name, last_name, and phone_numbers. The value of the phone_numbers field is a map that maps the keys home and work to the phone numbers '555-1234' and '555-5678', respectively.

Lastly, Databricks SQL supports several other types, including `INTERVAL`, which represents time intervals on a scale of seconds, years, and months. These types are compatible with HiveQL, which Manta already supports.

```
SELECT STRUCT (
    'John' AS first_name,
    'Doe' AS last_name,
    MAP('home', '555-1234', 'work', '555-5678') AS
↪   phone_numbers
    ) AS person
```

Code snippet 8: Nested map example

3.2 Expressions

Expressions are used to specify computations on data in SQL queries. The syntax for expressions in Databricks SQL is defined in Code snippet 9. Here, a `literal` refers to a fixed value, while `colum_reference` and `field_reference` refer to columns and fields in the table or a struct or a map. When the expression is used in the function `parameter_reference`, it refers to a parameter passed to this particular function. The `CAST` expression converts one data type to another. The `CASE` expression is used for conditional logic.

Operators such as `+`, `-`, `*`, `/`, and `%` can be used with expressions. It is also possible to use logical operators such as `AND`, `OR`, and `NOT`.

The priority levels of operators are as follows [21]:

1. `:`, `::`, `[ ]`
2. `-` (unary), `+` (unary),
3. `*`, `/`, `%`, `div`
4. `+`, `-`, `||`
5. `&`

6. ^
7. |
8. =, ==, <=>, <>, !=, <, <=, >, >=
9. not, exists
10. between, in, rlike, regexp, ilike, like, is [not] [NULL, true, false], is [not] distinct from
11. and
12. or

Operators with a higher precedence level (lower number) are evaluated before operators with a lower precedence level (higher number).

In Databricks SQL, associativity is generally left-to-right for arithmetic, comparison, and logical operators. This means that when multiple operators

```
expression:
{ literal
| column_reference
| field_reference
| parameter_reference
| CAST expression
| CASE expression
| expr operator expr
| operator expr
| expr [ expr ]
| function_invocation
| ( expr )
| ( expr, expr [, ... ] )
| scalar_subquery
}

scalar_subquery: ( query )
```

Code snippet 9: Expression syntax

with the same precedence level appear in an expression, they are evaluated from left to right.

In addition to operators, functions can be invoked on data. Functions are discussed in the following section.

Finally, a scalar subquery is a subquery that returns a single value. It can be used in place of a literal or expression in an expression.

3.3 Functions

In this section, we discuss the built-in, lambda, and user-defined functions in Databricks.

3.3.1 User-defined Functions

Databricks allows users to define their own functions, either as scalar or aggregate functions. Scalar functions return a single value, while aggregate functions return a result set. To create a user-defined function, the `CREATE FUNCTION` statement is used.

The general syntax of the `CREATE FUNCTION` statement is described in Code snippet 10. For example, the code in Code snippet 11 creates a simple scalar function that returns the string “Hello World!”. The function can then be used in a query; see Code snippet 12.

This returns the result set with one column `hello()` containing `Hello World!` string.

More complex user-defined functions can be created using SQL or other languages such as Java or Scala and imported as JAR. In this way, Hive UDFs can also be used. The user-defined functions must be registered to the Spark context in order to use them in SQL. This happens in the technology we use to define the function (Java, Scala etc.), not in SQL. Function parameters are referenced by unqualified names and are mapped by position.

```
CREATE [OR REPLACE] [TEMPORARY] FUNCTION [IF NOT EXISTS]
    function_name ( [ function_parameter [, ...] ] )
    RETURNS { data_type | TABLE ( column_spec [, ...]) 
    [ characteristic [...] ]
    RETURN { expression | query }

function_parameter :
    parameter_name data_type [DEFAULT default_expression]
↪   [COMMENT parameter_comment]

column_spec :
    column_name data_type [COMMENT column_comment]

characteristic :
  { LANGUAGE SQL |
    [NOT] DETERMINISTIC |
    COMMENT function_comment |
    [CONTAINS SQL | READS SQL DATA] |
    SQL SECURITY DEFINER }
```

Code snippet 10: UDF syntax

```
CREATE TEMPORARY FUNCTION hello()
    RETURNS STRING
    RETURN 'Hello World!';
```

Code snippet 11: UDF example

```
SELECT hello();
```

Code snippet 12: UDF usage example

```
CREATE FUNCTION roll_dice(num_dice  INT DEFAULT 1 COMMENT
↪  'number of dice to roll (Default: 1)',
                            num_sides INT DEFAULT 6 COMMENT
↪  'number of sides per die (Default: 6)')
    RETURNS INT
    NOT DETERMINISTIC
    CONTAINS SQL
    COMMENT 'Roll a~number of n-sided dice'
    RETURN aggregate(sequence(1, roll_dice.num_dice, 1),
                     0,
                     (acc, x) -> (rand() *
↪  roll_dice.num_sides)::int,
                     acc -> acc + roll_dice.num_dice);
```

Code snippet 13: A more complex UDF example

See the function in Code snippet 13; this is non-deterministic and returns an integer. It can be called in the ways seen in Code snippet 14.

```
-- Roll a single 6-sided dice
SELECT roll_dice();

-- Roll three 6-sided dice
SELECT roll_dice(3);

-- Roll three 10-sided dice
SELECT roll_dice(3, 10);
```

Code snippet 14: A usage of more complicated UDF

There can be user-defined functions with SQL query as their body; see Code snippet 15. However, it is impossible to run multiple statements with one UDF call - there is always only one expression or one query in the body of the UDF.

Examples were taken from the documentation [22].

```
CREATE FUNCTION weekdays(start DATE, end DATE)
    RETURNS TABLE(day_of_week STRING, day DATE)
    RETURN SELECT extract(DAYOFWEEK_ISO FROM day), day
             FROM (SELECT sequence(weekdays.start,
↪   weekdays.end)) AS T(days)
                  LATERAL VIEW explode(days) AS day
             WHERE extract(DAYOFWEEK_ISO FROM day) BETWEEN 1
↪   AND 5;
```

Code snippet 15: UDF returning a SELECT result set

3.3.2 Built-in Functions

Databricks provides many built-in functions that can be used in SQL expressions. These functions can be called in the same way as user-defined functions. a comprehensive list of Databricks built-in functions can be found in the documentation [23].

3.3.3 Lambda Functions

```
lambda_func:
{ param -> expr
| (param1 [, ...] ) -> expr
}
```

Code snippet 16: Lambda function syntax

```
SELECT transform(array('hello world', 'goodbye world'),
    s -> split(s, ' ')[0]) AS first_word
```

Code snippet 17: Lambda function example

Lambda functions are anonymous functions that can be used as built-in function arguments. The syntax for lambdas is defined in Code snippet 16, where expr refers to an expression. For example, the query in Code snippet 17 uses a lambda function to extract the first word from a string. This returns the result set containing one column with the name first_word and one row with an array ["hello","goodbye"].

Lambda function expressions may contain function calls, including UDFs and built-in functions. Lambda functions, in general, may introduce some data flow. Analysis of this flow must be based on the function that is calling the lambda function, as these dictate how the lambda is used.

3.4 Identifiers

Databricks employs a three-level naming notation [24]. The first level is a catalog; the second is a schema, and the lowest is table, view or function. The USE statement can define the catalog or schema. This statement defines the default catalog or schema for the whole script or notebook until USE statement is used again.

There are standard identifiers, which can begin with an underscore, a digit, or a letter. Valid identifiers include, for example _df, 123, _test, __, 1_, and _1.

To enable the use of special characters or reserved keywords in identifiers, delimited identifiers need to be used. These are enclosed in the backticks `` and can contain any character from the character set, even a dot, which normally separates name segments. You may refer to the delimited identifier without backticks as long as it does not contain special characters. Valid delimited identifiers include `a.b` or `a``B` or `a*@#$%&̂*\|:<>?/b`. However, delimited identifiers are not allowed in table names.

3.5 Statements

In Databricks SQL, statements are the primary means of interacting with and managing data and database objects. They can be broadly categorized into two types relevant to data lineage and the prototype implementation:

Data Definition Language (DDL) statements and Data Manipulation Language (DML) statements. This section provides an in-depth discussion of both DDL and DML statements, their specific use cases in Databricks SQL, and how they impact the data flow analysis of scripts. Note that Auxiliary, Delta Lake, and Security statements, which serve purposes such as optimization, metadata exploration, and resource management, are not discussed in this thesis, as they are not impactful to the data lineage analysis in the Databricks environment.

3.5.1 DML statements

DML statements enable data manipulation, including insertion, updating, deletion, and retrieval of data within these structures. The statements important for data lineage are especially `INSERT` statements, which may contain `SELECT` statements; there are also `LOAD DATA`, `MERGE IN` or `UPDATE` statements.

3.5.1.1 SELECT statements

`SELECT` syntax is illustrated by Code snippet 18. The `SELECT` statements are a crucial concept of SQL language. Here we analyse their clauses and their impact on the data flow within the statement.

The `SELECT` clause or a `SELECT` list is the main component of a `SELECT` statement, and it specifies the columns and literals to be retrieved in the result of this particular `SELECT` query. These columns can also be called a result set. In the example in Code snippet 19, the result set will contain three columns `a`, `b` and `c`.

The `FROM` clause specifies the tables, views or subqueries from which the data is retrieved. In the same example from Code snippet 19, direct data flow occurs between the source table `t1` and the selected columns (`a`, `b`, `c`). The flow of the statement is illustrated by Figure 3.1.

```
select_statement:
{
[ with_clause ]
  { query } [ set_operator select]*
  [ ORDER BY clause
    | { [ DISTRIBUTE BY clause ] [ SORT BY clause ] }
    | CLUSTER BY clause ]
  [ WINDOW clause ]
  [ LIMIT clause  ]
  [ OFFSET clause ]
}

query :
{ select |
  VALUES clause |
  ( query ) |
  TABLE [ table_name | view_name ]}


  select :
  SELECT [ hints ] [ ALL | DISTINCT ] { named_expression |
↪ star_clause } [, ...]
      FROM table_reference [, ...]
      [ LATERAL VIEW clause ]
      [ WHERE clause ]
      [ GROUP BY clause ]
      [ HAVING clause]
      [ QUALIFY clause ]

named_expression
   expression [ column_alias ]

star_clause
   [ { table_name | view_name } . ] * [ except_clause ]

except_clause
   EXCEPT ( { column_name | field_name } [, ...] )
```

Code snippet 18: Syntax of a select statement

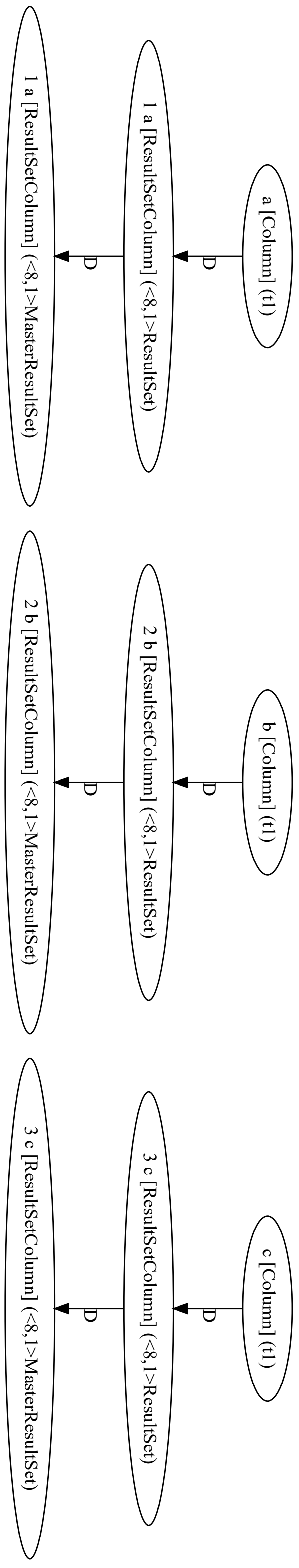

a [Column] (t1)
D
1 a [ResultSetColumn] (<8,1>ResultSet)
D
1 a [ResultSetColumn] (<8,1>MasterResultSet)
b [Column] (t1)
D
2 b [ResultSetColumn] (<8,1>ResultSet)
D
2 b [ResultSetColumn] (<8,1>MasterResultSet)
c [Column] (t1)
D
3 c [ResultSetColumn] (<8,1>ResultSet)
D
3 c [ResultSetColumn] (<8,1>MasterResultSet)

Figure 3.1: Data flow of the statement from Code snippet 19

```
SELECT a, b, c
    FROM t1;
```

Code snippet 19: An example of simple `SELECT` statement

The `WHERE` clause filters the rows returned in the result set based on one or more conditions. In the example in Code snippet 20, filter data flow occurs as the `c` column filters the rows but is not part of the selected columns. The data flow is illustrated by Figure B.1, a less detailed version by Figure 3.2.

```
SELECT a, b
    FROM t3
    WHERE c = 'Hello';
```

Code snippet 20: An example of `SELECT` statement with `WHERE` clause

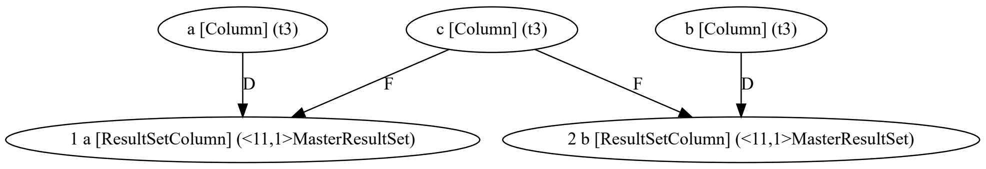

a [Column] (t3)
c [Column] (t3)
b [Column] (t3)
D
F
F
D
1 a [ResultSetColumn] (<11,1>MasterResultSet)
2 b [ResultSetColumn] (<11,1>MasterResultSet)

Figure 3.2: Data flow of the statement from Code snippet 20

The `GROUP BY` clause groups the rows based on one or more columns, and the `HAVING` clause filters the grouped data based on a condition. In the example in Code snippet 21, filter data flow occurs in the `GROUP BY` clause, as illustrated in Figure 3.3, in more detail in Figure B.2.

```
SELECT d, COUNT(*) as c
    FROM t4
    GROUP BY d;
```

Code snippet 21: An example of `SELECT` statement with `GROUP BY` clause

During analysis, it was discovered that the syntax `FROM table SELECT columns` is also supported, despite the fact it is not documented in the official

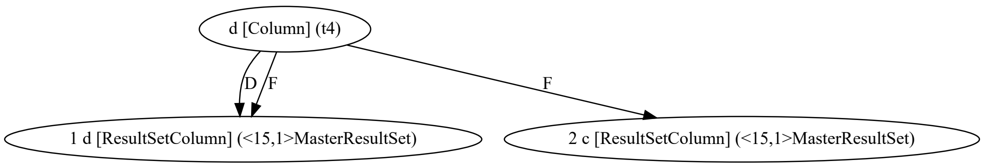

d [Column] (t4)
D
F
F
1 d [ResultSetColumn] (<15,1>MasterResultSet)
2 c [ResultSetColumn] (<15,1>MasterResultSet)

Figure 3.3: Data flow of the statement from Code snippet 21

documentation. The example can be seen in Code snippet 22. There is an option to chain `SELECT` clauses; result sets are then unionised. The data flow of the statement from this example can be seen in Figure 3.4 in more detail then in Figure B.3.

This syntax was probably adopted from Hive QL; we can see this syntax in [25]. However, even in the Hive QL, this feature is undocumented. Mention of this syntax can also be found in [26]: *In Spark version 2.4 and below, SQL queries such as `FROM <table>` or `FROM <table> UNION ALL FROM <table>` are supported by accident. In hive-style `FROM <table> SELECT <expr>`, the `SELECT` clause is not negligible. Neither Hive nor Presto support this syntax. These queries are treated as invalid in Spark 3.0.*

```
FROM t1
    SELECT a,b
    SELECT c,d
```

Code snippet 22: Undocumented but valid `SELECT` syntax

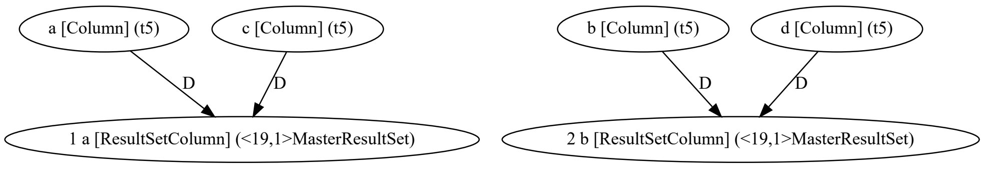

a [Column] (t5)
c [Column] (t5)
b [Column] (t5)
d [Column] (t5)
D
D
D
D
1 a [ResultSetColumn] (<19,1>MasterResultSet)
2 b [ResultSetColumn] (<19,1>MasterResultSet)

Figure 3.4: Data flow of the statement from Code snippet 22

3.5.1.2 INSERT statement

The INSERT statement in Databricks SQL adds new rows to a table. There are two syntax forms: the standard `INSERT INTO` statement and the `INSERT INTO...REPLACE WHERE` statement, both described in Code snippet 23 [20].

```
INSERT { OVERWRITE | INTO } [ TABLE ] table_name
    [ PARTITION clause ]
    [ ( column_name [, ...] ) ]
    query

INSERT INTO [ TABLE ] table_name
    REPLACE WHERE predicate
    query
```

Code snippet 23: Syntax of an `INSERT` statement

The `INSERT INTO` statement establishes a direct data flow between the source (`query`) and the target table (`table_name`). The `PARTITION` clause can be used to specify the partition key(s) for the table, and the optional column list allows inserting data into specific columns, leaving others with their default values or `NULL`.

```
INSERT INTO t2 (a, b)
VALUES ('John Doe', 123456);
```

Code snippet 24: An example of `INSERT INTO` statement with specified columns

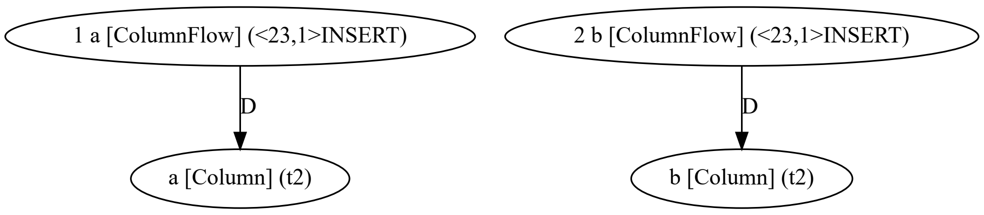

1 a [ColumnFlow] (<23,1>INSERT)
D
a [Column] (t2)
2 b [ColumnFlow] (<23,1>INSERT)
D
b [Column] (t2)

Figure 3.5: Dataflow of the statement from Code snippet 24

In Code snippet 24, direct data flow occurs between the source (VALUES clause) and the target table t2, inserting data into the specified columns a and b. The flow is illustrated in Figure 3.5 and Figure B.4

```
INSERT INTO t1 PARTITION (c = 42)
    SELECT a, b FROM t2 WHERE c = 'World';
```

Code snippet 25: INSERT INTO statement example with PARTITION clause

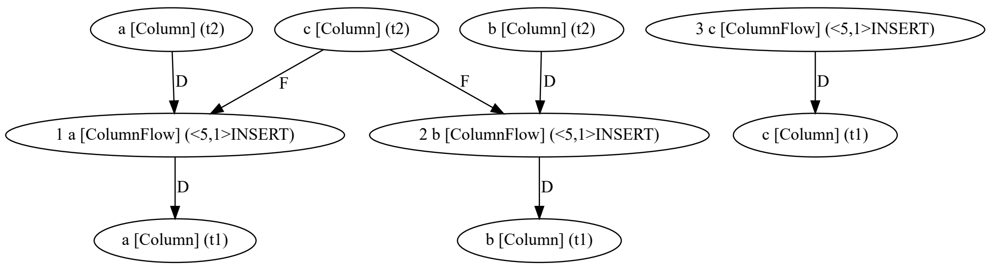


Figure 3.6: Data flow of the statement from Code snippet 25

In Code snippet 25, direct data flow occurs between the source (the result set of SELECT statement) and the target table t1. The PARTITION clause specifies the partition key, creating a separate partition for the c value 42. The data flow from this statement is illustrated by Figure 3.6 and Figure B.5

The INSERT INTO...REPLACE WHERE statement allows inserting new rows while replacing existing rows that match the specified predicate. It establishes a direct data flow between the source and the target table and a filter data flow results from the REPLACE WHERE predicate. The same is happening in UPDATE statement.

```
INSERT INTO students
REPLACE WHERE student_id = 123456
VALUES ('John Doe', '456 Main St', 123456);
```

Code snippet 26: An example of INSERT INTO...REPLACE WHERE statement

In Code snippet 26, direct data flow occurs between the source (`VALUES` clause) and the target table `students`, while filter data flow results from the `REPLACE WHERE` predicate, ensuring that the new row replaces any existing row with the same `student_id` value.

Despite not being documented in Databricks official documentation [20], the statement syntax in Code snippet 27 is also supported. This statement inserts the content of columns `a` and `b` from `t1` into table `t2` and the content of columns `a` and `b` from the same table`t1` into table `t3`, as seen in Figure 3.7. Unlike the similar `SELECT` syntax, this `INSERT` syntax is officially documented as a feature of Hive QL [27].

```
FROM t1
    INSERT INTO t2 SELECT a,b
    INSERT INTO t3 SELECT c,d;
```

Code snippet 27: Undocumented `INSERT SELECT` syntax

3.5.1.3 Other DML statements

`LOAD DATA` statement causes direct flow from a specified file to a given table, as can be seen in Code snippet 28. The data flow for this example can be seen in Figure 3.8.

```
LOAD DATA LOCAL INPATH '/user/path/to/file' OVERWRITE INTO
↪   TABLE test;
```

Code snippet 28: An example of `LOAD DATA` statement

The `MERGE INTO` statements merge a collection of updates, insertions, and deletions from a source table and integrate them into a target Delta table, which will cause a direct data flow from the source table to the target table. This statement is exclusively supported for Delta Lake tables. An example of

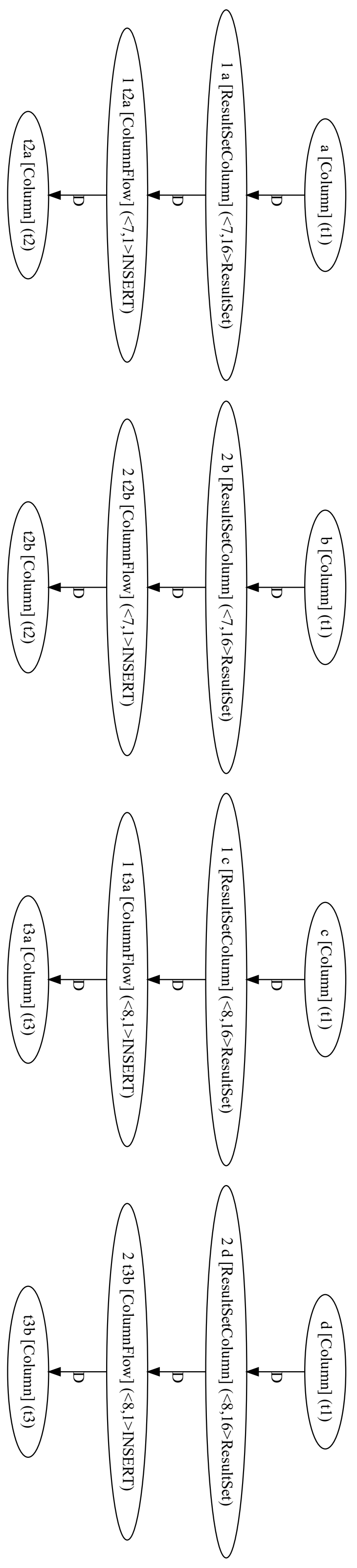

a [Column] (t1)
1 a [ResultSetColumn] (<7,16>ResultSet)
1 t2a [ColumnFlow] (<7,1>INSERT)
t2a [Column] (t2)
b [Column] (t1)
2 b [ResultSetColumn] (<7,16>ResultSet)
2 t2b [ColumnFlow] (<7,1>INSERT)
t2b [Column] (t2)
c [Column] (t1)
1 c [ResultSetColumn] (<8,16>ResultSet)
1 t3a [ColumnFlow] (<8,1>INSERT)
t3a [Column] (t3)
d [Column] (t1)
2 d [ResultSetColumn] (<8,16>ResultSet)
2 t3b [ColumnFlow] (<8,1>INSERT)
t3b [Column] (t3)
D

Figure 3.7: Data flow of the statement from Code snippet 27

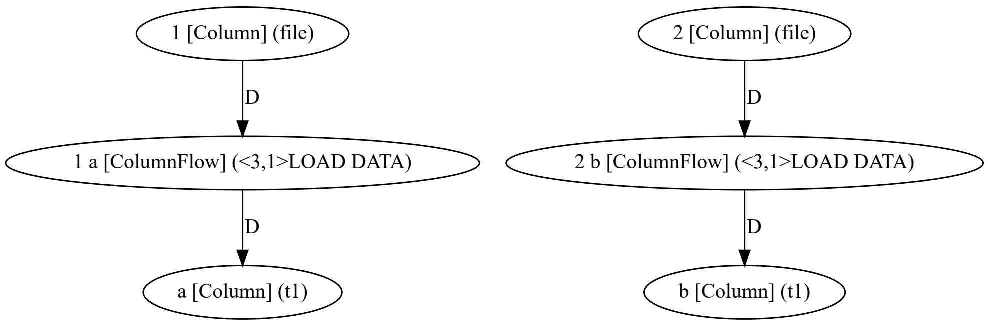


Figure 3.8: Data flow of the statement from Code snippet 28

```
MERGE INTO target USING source
  ON target.key = source.key
  WHEN MATCHED THEN DELETE
```

Code snippet 29: An example of `MERGE INTO` statement

this statement can be seen in Code snippet 29. Statement from this example will delete all target rows that have a match in the source table.

Lastly, the `UPDATE` statement modifies the column values of rows that satisfy a given predicate. If no predicate is supplied, it updates the column values for all rows. This statement creates the same flow as the `INSERT INTO...REPLACE WHERE` statement mentioned earlier. This statement is solely supported for Delta Lake tables.

3.5.2 DDL statements

DDL statements allow users to define, modify, and delete database structures such as tables, views, and indexes. Data definition language statements, such as `CREATE` statements and `ALTER` statements for tables, views, schemas, and catalogs define data sources. These data sources can then be stored and involved in lineage generated from other statements. `CREATE TABLE` statements may introduce a data flow, as they may copy data from other tables. A simple example can be seen in Code snippet 30; the data flow for this code is then illustrated by Figure 3.9.

```sql
CREATE TABLE t1 (a int, b int);

CREATE TABLE t2 AS SELECT * FROM t1;
```

Code snippet 30: An example of `CREATE TABLE` statement

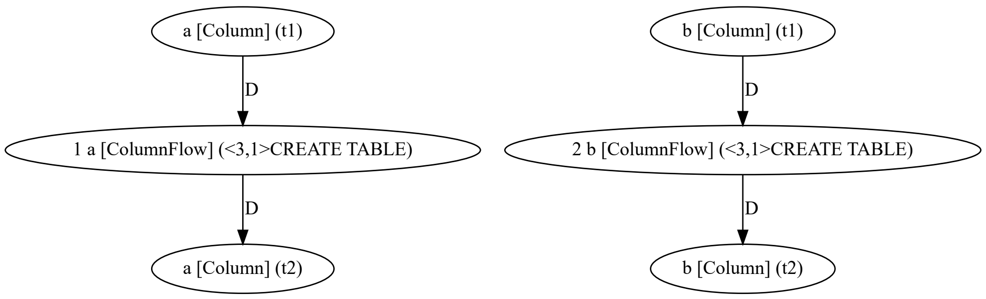

a [Column] (t1)
D
1 a [ColumnFlow] (<3,1>CREATE TABLE)
D
a [Column] (t2)
b [Column] (t1)
D
2 b [ColumnFlow] (<3,1>CREATE TABLE)
D
b [Column] (t2)

Figure 3.9: Data flow of the statement from Code snippet 30

Also, `CREATE FUNCTION` is a statement that may impact the data lineage as the function may contain select and therefore introduce new data flow. `DROP` statements are unimportant for us, as we do not rely on the correct ordering of statements; therefore, we want to persist metadata even from dropped tables.

Other statements are not as important, but they may be parsed in future versions of the prototype to ensure a better user experience without parsing errors on unimportant statements.

3.6 Parameterising options

Databricks offer two types of parameterising scripts based on the environment this script comes from. For Databricks notebook cells, it is Spark variables; for queries, it is parameters. This parametrisation introduces a need for pre-processing scripts before the analysis.

3.6.1 Spark Variables

In Databricks, it is possible to combine SQL and Python code by using Spark context variables. These variables can be accessed using the `${variable_name}` syntax in SQL queries. Spark variables can hold values such as numbers, text, or even code. Let's take the Python notebook cell in Code snippet 31 as an example. This cell sets several Spark variables. We can then use these variables in SQL cells in the same notebook.

```
%python
spark.conf.set("app.code1", "SELECT 1")
spark.conf.set("app.code2", "FROM t1")
spark.conf.set("app.text1", "'Hello'")
spark.conf.set("app.text2", "world")
spark.conf.set("app.number", 2)
```

Code snippet 31: An example of Python notebook cell

For instance, the cell in Code snippet 32 returns one row containing the number 1. Similarly, the query in the Code snippet 33 returns one row containing the strings `"Hello"`, `"world"`, and the number `2`.

From testing, it was discovered that these variables are replaced even recursively.

```
${app.code1} ${app.code2} LIMIT 1;
```

Code snippet 32: An example of SQL notebook cell

```
SELECT ${app.text1}, '${app.text2}', ${app.number} FROM t1
↪  LIMIT 1;
```

Code snippet 33: An example of SQL notebook cell

3.6.2 Query Parameters

Query parameters offer a way to pass arguments to Databricks SQL queries. These parameters can be accessed using the syntax `{{ param }}` in the SQL code. Parameters can be configured via a widget in the Databricks SQL query console. They can contain text, numbers, or drop-down values. Moreover, it is possible to have code as the content of a parameter. Drop-down options can also be generated using a different query. There is also an option for multiple values, which are replaced with a separator.

For example, the SQL code in Code snippet 34 uses a parameter `t1`. The parameter `t1` can be set to a value such as `FROM t1` or `FROM t1 WHERE t1.a > 2`, and both will work.

```sql
SELECT * {{ t1 }};
```

Code snippet 34: An example of parametrised SQL query

Chapter 4

Design

In this chapter, we discuss the design of our prototype for data flow analysis of Databricks SQL. The prototype uses various technologies, including Java, ANTLR, Maven, JUnit, and Spring. We also explore the different modules that make up the prototype.

4.1 Used technologies

This section discusses the technologies used to extend the Manta tool for scanning Databricks SQL. The primary reason for using these technologies is to maintain compatibility with the existing Manta codebase.

4.1.1 Java

Java is the programming language chosen for this project due to its compatibility with the existing Manta codebase. Some of the general advantages of Java include platform independence, object-oriented nature, robustness, and extensive libraries, which contribute to the development of a reliable and efficient tool [28].

4.1.2 ANTLR

ANTLR provides an effective method for analysing input code and generating ASTs, which is crucial for the Manta tool's functionality. The theory behind ANTLR and its parsing algorithms is discussed in section 1.5. ANTLR version 3 is employed in this project to generate custom Abstract Syntax Trees (ASTs) with rewrite rules, a feature not available in version 4.; therefore, ANTLR 3 is chosen as custom AST is used to analyse the input code, use of the ANTLR 4 would require more code to define the AST. Scanners of other SQL-based technologies in Manta also use ANTLR 3, so ANTLR 3 choice also supports compatibility with the rest of the product [29].

4.1.3 Maven

Maven is the build automation and project management tool used throughout the Manta product. Its benefits include dependency management, build lifecycle, and standard project structure. While Maven streamlines the development process, it can also introduce challenges such as managing complex dependency trees and addressing build failures. However, these potential disadvantages are mitigated through careful configuration and adherence to best practices [30].

4.1.4 Spring

The Spring Framework is used in the Manta product to manage dependencies, provide modularity, and offer an extensive ecosystem for addressing various project needs. While Spring simplifies the development process, it can sometimes introduce complexities such as steep learning curve and configuration challenges. By leveraging the advantages of Spring and addressing potential challenges, the project benefits from a robust and flexible architecture [31].

4.1.5 JUnit

JUnit is the testing framework employed in the Manta product. It facilitates unit testing and integrates well with Maven and Java, ensuring the developed code is reliable and functions as expected. In addition to serving as unit tests, our tests function as regression tests, executing each time the application is extended. This process ensures that the existing functionality remains intact and is not adversely affected by new changes. Some potential disadvantages of JUnit include the need for manual test case creation and the possibility of false positives or negatives if tests are not designed thoroughly. However, these challenges can be addressed by following best practices and ensuring comprehensive test coverage [32].

4.2 Modules

This section introduces the scanner unit prototype, designed for data flow analysis of Databricks SQL. The module is divided into four modules. Dividing the scanner unit prototype like this, mirroring the structure used in other scanner units for similar SQL technologies, offers several advantages. This modular approach promotes the separation of concerns, where each module focuses on a specific aspect of the data flow analysis process. Additionally, this design choice allows for easier integration with other projects or technologies, as the standardised structure facilitates understanding and collaboration.

The `manta-connector-databrickssql-model` module provides interfaces for AST nodes. These interfaces are crucial in ensuring compatibility between different modules within the prototype.

The `manta-connector-databrickssql-resolver` module is the core component for performing the parsing process, constructing the AST, and resolving semantics for Databricks SQL scripts. It also contains the parser generated by ANTLR and its source grammars, located in `antlr/parser` folder. Theoretical background for AST construction and general language analysis process were described in chapter 1.

The `manta-connector-databrickssql-testutils` module contains common test classes that are essential for validating the functionality of other modules. These test classes' testing methodology and purpose will be covered in chapter 6.

Lastly, the `manta-dataflow-generator-databrickssql` module is responsible for constructing the data flow graph from the resolved AST. To define the data flow graph, refer to section 1.7.

4.2.1 Dependencies

Modules are dependent on each other. The UML diagram in the Figure 4.1 illustrates the relationship between the modules and their dependencies. All modules are dependent on other Manta libraries and modules, which are commonly used by multiple scanners for different technologies. Resolver and model packages also rely on the dictionary component, which is located in the Databricks scanner module.

The `manta-connector-databrickssql-resolver` package depends on `manta-connector-databrickssql-model`.

The `manta-dataflow-generator-databrickssql` module depends on `manta-dataflow-generator-databrickssql-model`.

The package `manta-dataflow-generator-databrickssql` depend on the `manta-dataflow-generator-databrickssql-testutils` module; however, this package is omitted from the diagram.

4.2.2 Connector

The connector artefact is responsible for handling parsing, AST construction, and resolution for Databricks SQL scripts. This section delves into the design of the connector package, exploring its various parts and their roles.

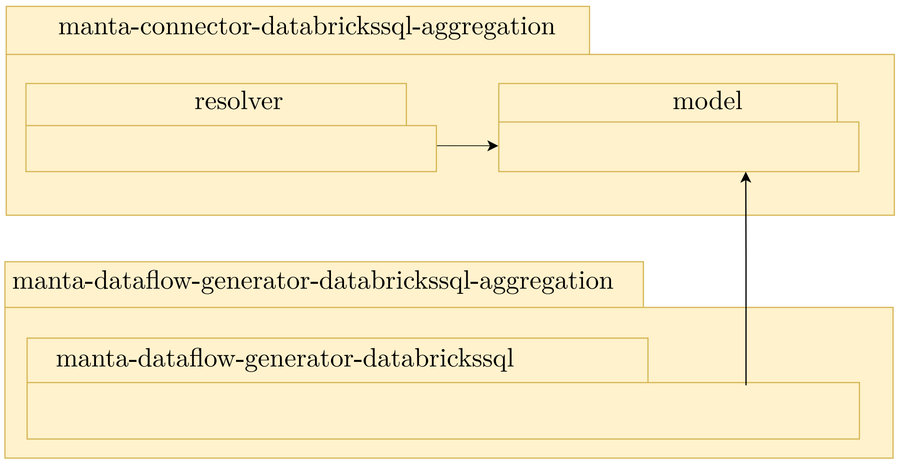

manta-connector-databrickssql-aggregation
resolver
model
manta-dataflow-generator-databrickssql-aggregation
manta-dataflow-generator-databrickssql

Figure 4.1: UML diagram showing dependencies between modules

The parser is generated using the ANTLR library, with its theory discussed in the section 1.5. The parser consists of classes `DatabricksSQLMain`, `DatabricksSQLMain_NonReservedKW`, `DatabricksSQLMain_Expressions`, and `DatabricksSQLLexer`. These classes are generated from lexer and parser grammars and they extend external Manta classes `MantaAbstractParser` and `MantaAbstractLexer`, providing additional functionality. The lexer is generated from a grammar file called `DatabricksSQLLexer.g`, while the parser is generated from a grammar file named `DatabricksSQLMain.g`, with its parts located in separate files `DatabricksSQLExpressions.g` with expression grammar and `DatabricksSQLNonReserveredKW.g` with the specification of non-reserved keywords. These grammar files reside within the `antlr3/parser` package.

The connector package also includes classes with the `Ast-` prefix, representing AST nodes and containing logic executed during resolution. These classes extend the common abstract class `DatabricksSQLAstNode` and implement model interfaces from the `manta-connector-databrickssql-model` module to ensure compatibility between modules. The AST nodes are constructed during parsing by the parser.

Parsing is provided externally by the `ParserService` interface and its implementation, `ParserServiceImpl`. These components offer methods to parse scripts either by supplying a file name or directly through a string. `ParserServiceImpl` is also responsible for preprocessing the input scripts.

We can see the relationship between essential classes inside the resolver on the Figure 4.2.

Preprocessing prepares the input scripts for subsequent analysis by parameters and variables substitution. This subsection describes the preprocessing process.

In the `ParserServiceImpl` class, preprocessing is performed based on the configuration passed as a parameter.

This configuration is stored in the `ParserService`
`Params` class, which maintains information about the preprocessing settings. The configuration includes a map and two booleans, `isParametrized` and `containsVariables`, which indicate whether a parameter (from a query) or a Spark variable are present. The map contains pairs of key-value strings representing the names of parameters or variables and their corresponding values.

The preprocessing process also handles the possibility of recursive variable substitution, where variables can be substituted within other variables. Considering this case, the preprocessing implementation ensures that all variables and parameters are correctly resolved before the input script is passed on for further analysis in the scanner unit.

4.2.3 Data flow Graph Generator

The data flow graph generator is responsible for creating a data flow graph from the resolved AST. This section explores the design and components of the data flow graph generator, with essential relationships illustrated in Figure 4.3.

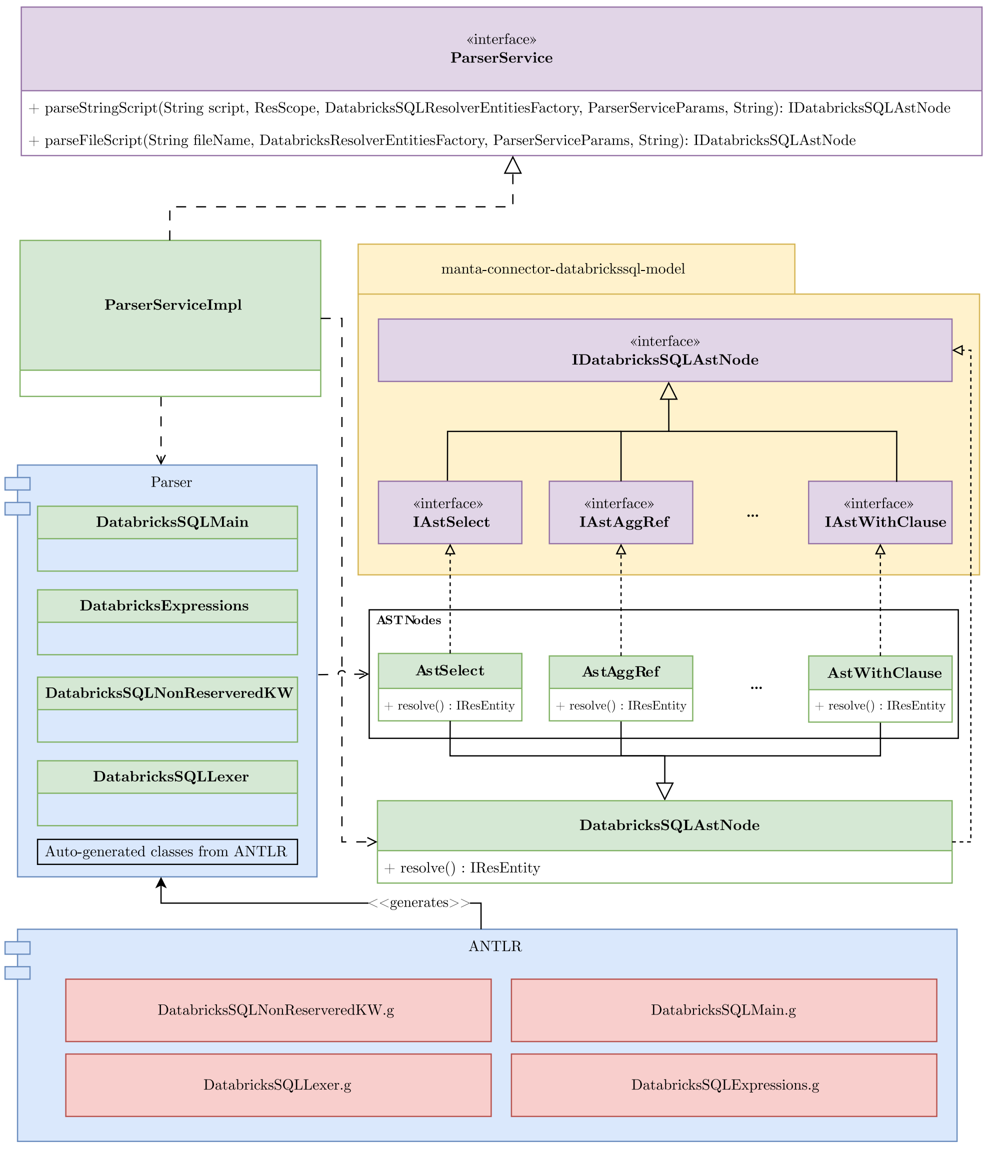


Figure 4.2: Schematic representation of key classes and their interactions within the connector module

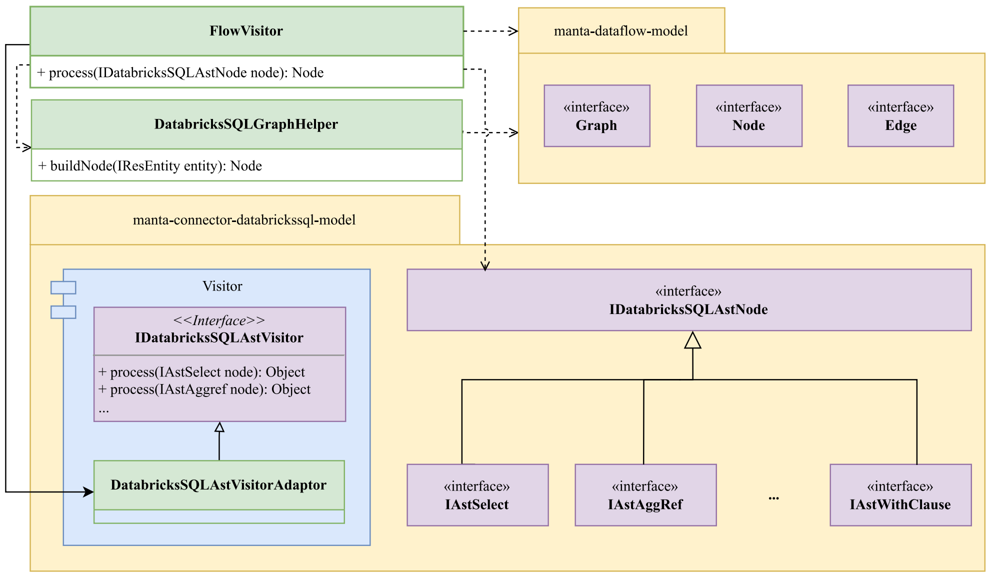

FlowVisitor
+ process(IDatabricksSQLAstNode node): Node
DatabricksSQLGraphHelper
+ buildNode(IResEntity entity): Node
manta-dataflow-model
«interface» Graph
«interface» Node
«interface» Edge
manta-connector-databrickssql-model
Visitor
<<Interface>> IDatabricksSQLAstVisitor
+ process(IAstSelect node): Object
+ process(IAstAggref node): Object
...
DatabricksSQLAstVisitorAdaptor
«interface» IDatabricksSQLAstNode
«interface» IAstSelect
«interface» IAstAggRef
...
«interface» IAstWithClause

Figure 4.3: Schematic representation of key classes and their interactions within the data flow generator module

The data flow generator relies on the Visitor design pattern [33] and the `manta-connector-databrickssql-model` module. The module contains an interface for the Visitor design pattern and class, which provides a default implementation of the visitor interface. The `FlowVisitor` class then implements the visitor logic required for creating a data flow graph. Each AST node must implement the accept method from the `IDatabricksSQLAstNode` interface, ensuring compatibility with the visitor pattern.

The `FlowVisitor` class is responsible for constructing the data flow graph, utilising a helper class called `DatabricksSQLGraphHelper`. This helper class extends an external class, `AbstractGraphHelper`, which contains implemented methods for graph building. The `DatabricksSQLGraphHelper` leverages classes from the `mantadataflow-model` module to create the data flow graph, streamlining the process and ensuring consistency across the scanner unit prototype.

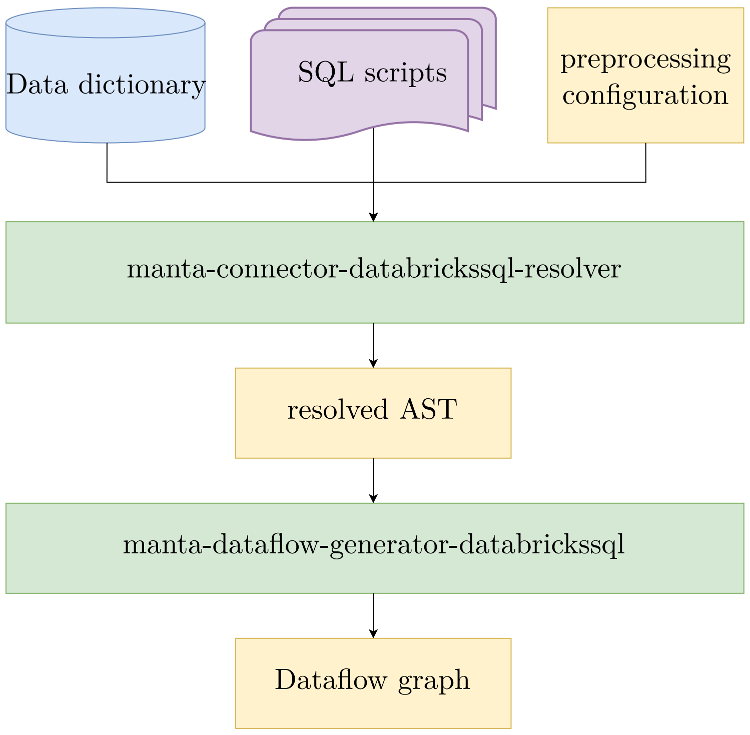

Data dictionary
SQL scripts
preprocessing configuration
manta-connector-databrickssql-resolver
resolved AST
manta-dataflow-generator-databrickssql
Dataflow graph

Figure 4.4: Schematic representation of workflow

4.3 Workflow

This section outlines the steps in processing the input, resolving the AST, and generating the data flow graph, as illustrated in Figure 4.4.

The scanner unit takes three main inputs: Data dictionary, preprocessing configuration, and SQL scripts. SQL scripts can include queries or notebook cells.

The first step in the execution is processing the input scripts with the `manta-connector-databrickssql-resolver` module. This module is responsible for preprocessing the input and creating a resolved AST from the provided inputs. By leveraging the parser components within the module, it converts the input scripts into a structured AST, which is then resolved. The AST then serves as an intermediate representation of the input SQL code for further analysis.

Once the resolved AST is obtained, it is passed to the `manta-dataflow-generator-databrickssql` module. This module creates the data flow graph from the resolved AST. By utilising the Visitor design pattern, helper classes, and the `mantadataflow-model` module, the data flow graph generator constructs a comprehensive data flow graph that represents the analysed Databricks SQL scripts. Then it also filters this data flow graph using `FilterTask` module, an external Manta component. Filtering creates a less detailed data flow graph, which is more readable and generally user-friendly.

Chapter 5

Implementation

This chapter covers the implementation details of our prototype for data flow analysis of Databricks SQL. We describe the crucial classes and constructs used in the implementation and provide a big picture of the solution for the reader to grasp the concept better. The implementation of our prototype is based on the design principles discussed in the previous chapter. The first section of this chapter covers the implementation of the parser module. We discuss the grammar used for generating the parser using ANTLR and the classes and constructs used in the implementation.

5.1 ANTLR Parser

As discussed earlier, the parser is automatically generated from ANTLR grammars. The main two are lexer grammar and parser grammar, located in `DatabricksSQLLexer.g` and `DatabricksSQLMain.g`, respectively. The corresponding Java classes are generated from these files during the build process.

5.1.1 Lexer grammar

The lexer is generated from a grammar defined in the `DatabricksSQLLexer.g` file. This grammar specifies all the tokens used in parser grammar as well as AST tree node tokens. Code snippet 35 shows a short example of lexer rules.

The reserved and non-reserved keywords are distinguished by their prefixes. Non-reserved keywords are prefixed with `KW_` (e.g., `KW_SCHEMA` and `KW_SELECT`).

```
UNION : 'UNION';
WHEN : 'WHEN';
WHERE : 'WHERE';

KW_SCHEMA : 'SCHEMA';
KW_SELECT : 'SELECT';

LEFT_PAREN : '(' ;
RIGHT_PAREN : ')' ;
```

Code snippet 35: An example of lexer rules

Fragments are used in the lexer grammar to create reusable rules that can be combined with other rules; however, these can't be used outside the lexer grammar. For illustration, see Code snippet 36.

```
fragment
Letter : 'a'..'z' | 'A'..'Z' ;

fragment
HexDigit : 'a'..'f' | 'A'..'F' ;

fragment
Digit : '0'..'9' ;
```

Code snippet 36: An example of fragment lexer rules

The hidden channel removes specific tokens from the output token stream. As an example, in Code snippet 37, whitespace characters (`WS`) and single-line comments `SINGLE_LINE_COMMENT`) are assigned to the hidden channel. However, this hidden channel preserves tokens for later usage if needed.

There are two types of identifiers in the lexer. Regular one and the delimited one, see `ID` token and `DELIMITED_ID` token in Code snippet 38. This

```
WS : (' '|'\r'|'\t') {$channel=HIDDEN;} ;

SINGLE_LINE_COMMENT : '--' (~('\n'|'\r'))* {$channel=HIDDEN;} ;
```

Code snippet 37: A lexer rules with a specification of a hidden channel

delimited identifier is used to represent identifiers, which are enclosed in back-ticks (`), such as `a.b`. Delimitation keeps the dot . as part of the name instead of using it to specify the outer context (e.g. the table of the column or a schema of the table).

```
ID : (Letter | Digit | '_')+ ;

DELIMITED_ID : BACKTICK (~(BACKTICK) | (BACKTICK BACKTICK))*
↪  BACKTICK ;
```

Code snippet 38: A ID rules

5.1.2 Parser grammar

The parser grammar is implemented in grammar files `DatabricksSQLMain.g`, `DatabricksSQLExpressions.g` and `DatabricksSQLNonReservedKW.g`. From these grammar files corresponding classes are generated by the ANTLR library. The parser processes the token stream generated by the lexer. The `DatabricksSQLMain.g` file contains the main parsing rules for Databricks SQL scripts. The Code snippet 39 illustrates some key components of the grammar.

```
databrickssql_script :
    bl = statement_list eof = EOF
    -> ^(AST_SCRIPT $bl?)
    ;

statement_list : single_statement (SEMICOLON single_statement)*
↪   ;

single_statement
    :   standalone_query_statement
    |   s = create_table_statement          ->
↪   ^(AST_CREATE_TABLE_STATEMENT<AstCreateTable>[contextState,
↪   SYMBOLTABLESCOPE_stack.peek().getScope()] $s)
    |   s = create_view_statement           ->
↪   ^(AST_CREATE_VIEW_STATEMENT<AstCreateView>[contextState,
↪   SYMBOLTABLESCOPE_stack.peek().getScope()] $s)
    |   s = create_schema_statement         ->
↪   ^(AST_CREATE_SCHEMA_STATEMENT<AstCreateSchema>[contextState,
↪   SYMBOLTABLESCOPE_stack.peek().getScope()] $s)
   ...
    ;
```

Code snippet 39: Simplified example of DatabricksSQLMain.g file

The `databrickssql_script` rule in Code snippet 39 represents the entry point for the parsing process and defines the overall structure of a Databricks SQL script. The `statement_list` rule represents a list of statements in the script and is simplified in this code snippet, while the `single_statement` rule specifies the different types of statements that can appear in the script, such as `standalone_query_statement` or `create_schema_statement`. We can see the definition of some AST nodes in the rules.

The expression logic for Databricks SQL scripts, including `SELECT` and `INSERT` statements, is implemented in the `DatabricksSQLExpressions.g` file. This file handles the logic for expressions and also ensures proper precedence while parsing and resolving expressions.

The `DatabricksSQLNonReservedKW.g` file defines the identifier rule, distinguishing between reserved and non-reserved keywords. The Code snippet 40 illustrates the implementation of identifier and non-reserved keywords recognition.

```
identifier
    :   ID
    |   DELIMITED_ID
    |   non_reserved_words
    ;

non_reserved_words // CHECK NON-RESERVED
    : KW_ABORT
    | KW_ACCOUNT
    | KW_ALL
    | KW_ALTER
    | KW_ALWAYS
   ...
    ;
```

Code snippet 40: Identifier rule in the DatabricksSQLNonReservedKW file

Parser is called via `ParserService` interface, which is implemented by `ParserServiceImpl` class. This class needs to get a data dictionary and additional parameters, including the current schema, the current catalog and a substitution map - a map of variables or parameters to be substituted during preprocessing. It accepts input via either a file or a string. The functioning of this class is illustrated by a sequential diagram in Figure 5.1.

First, the preprocessing is performed. Preprocessing means the process of substituting parameters and variables with their values passed with the map within the `ParserServiceParams` class. Parameter and variable substitutions are accomplished using regular expressions that match parameter or variable references in the input script. The matched references are then replaced with their corresponding values, either determined by Python analysis (in the case of variables) or extracted directly (in the case of parameters). Preprocessing also substitutes parameters and values with unknown values for delimited identifiers with `unknown_` prefix, so if these only contain some data object, these objects could be deduced.

After the script is preprocessed, the Lexer tokenise the input script and creates a token stream, which is then passed to the parser in `DatabricksSQLMain` class. The parser creates and returns the AST. The AST is then resolved and returned.

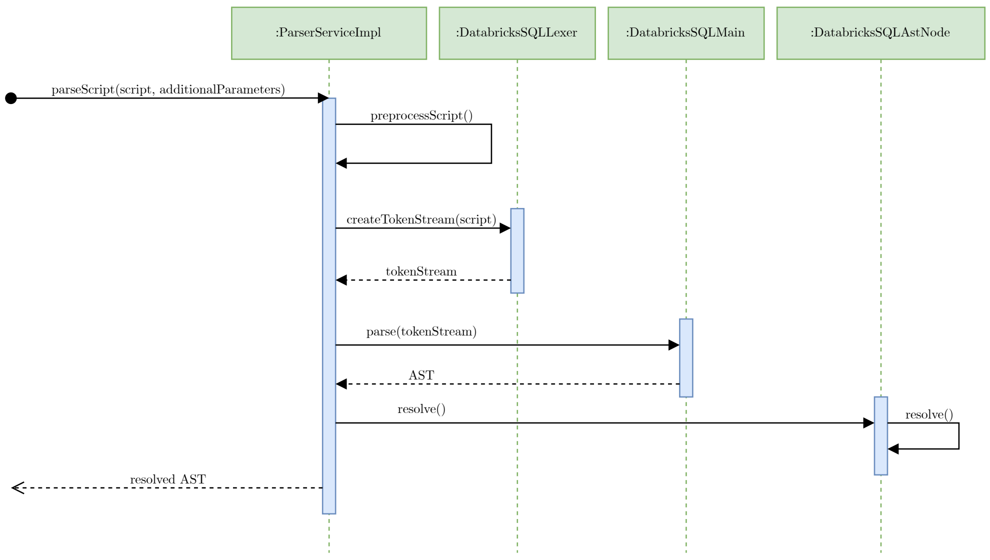


Figure 5.1: Sequential diagram of the ParserServiceImpl class

5.2 Resolving

The resolver component comprises the lexer, parser, and Abstract Syntax Tree (AST) nodes. The resolution process aims to identify the objects referenced in the script, so we can later analyse the data flow in the script.

5.2.1 AST nodes

The resolution process of an AST node is by default only delegating on its descendants in the tree, see Code snippet 41. AST node classes are responsible for implementing the specific logic associated with different statement types in the resolver. Each node type implements unique resolving logic, ensuring the resolver can accurately process and analyse various statements within the Databricks SQL script.

```
public IResEntity resolve() {
    // Don't do anything by default; delegate this to
    ↪  children.
    for (DatabricksSQLAstNode item : getChildren()) {
        item.resolve();
    }

    return null;
}
```

Code snippet 41: Default implementation of resolving

For DDL statements, the resolver identifies the name and the correct scope for the object, then creates the object in the dictionary if it is not already present. This process allows the resolver to handle and manage the objects within the script properly.

As an example, we explain the resolution of the `SELECT` statement. `SELECT` statements are more complex to resolve, as the resolver must identify all referenced columns and their associated tables. The resolver processes the `WITH` clauses and adds a table to the context - map of objects available in the scope. Then resolves all the `SELECT` statements connected by set operators - identifies tables and result set columns within those `SELECT`s, including references in `WHERE`, `GROUP BY` or `HAVING` clauses - and resolves the additional clauses for the whole statement, such as `ORDER BY`, `LIMIT`, `PARTITION SORT BY` or `DISTRIBUTE BY` clauses. This comprehensive process ensures that all relevant references within the `SELECT` statement are accurately resolved, so we know which object they are referencing.

5.2.2 Deduction

The resolver might encounter unknown objects or type mismatches during the resolution process. The resolver attempts to deduce the referenced object to give the user a better result, making assumptions based on its context. For example, if the unknown object is in the result set, it is likely a column, while if it is in the FROM clause, it is likely a table. This technique allows the resolver to manage potential errors and uncertainties during the resolution process.

5.3 Data flow generator

The Data flow generator aims to build a data flow graph. The input for the Data flow generator is the resolved AST, while the output is a data flow graph representing the data flow between tables, columns, and other database objects in the script.

The critical components of the Data flow generator are the `FlowVisitor` and `DatabricksSQLGraphHelper` classes. The `FlowVisitor` class implements the Visitor design pattern, as described in subsection 4.2.3, and includes a `process` method to analyse nodes in the resolved AST. This method returns a node object used in the output data flow graph.

The `DatabricksSQLGraphHelper` class is used by the `FlowVisitor` class to create nodes and edges for the data flow graph. This class extends the `AstGraphHelper` class, which is part of Manta's external libraries.

An example of the `process` method in the `FlowVisitor` class for processing a `WITH` clause is shown in Code snippet 42. In this example, the `process` method finds the target node of the WITH clause, creates target column nodes using the `DatabricksSQLGraphHelper` class, and connects them to the corresponding source column nodes.

```
public List<Node> process(IAstWithClause node) {
    LOGGER.trace("visited IAstWithClause: {}", logNode(node));

    // target
    IResEntity resWithAlias = node.getReferencedObject();
    List<Node> targetColumnNodes =
    ↪   graphHelper.buildColumnNodes(resWithAlias);

    // process the source
    IAstMasterQuery astWithSelect = node.findWithQuery();
    List<Node> sourceColumnNodes =
    ↪   processAstNode(astWithSelect);

    // connect
    graphHelper.addDirectFlow(sourceColumnNodes,
    ↪   targetColumnNodes);

    return targetColumnNodes;
}
```

Code snippet 42: FlowVisitor.process method example

Chapter 6

Testing

In this chapter, we describe the tests developed for the individual modules of the Databricks SQL data flow analysis system. These tests ensure that each module functions correctly. We then delve into the details of the tests for each module, highlighting the specific functionality being tested and the expected results.

This chapter focuses on the testing process for the two main modules of the prototype: the `manta-connector-databrickssql` and the `manta-dataflow-generator-databrickssql`. Thorough testing of these modules is crucial for ensuring the correctness and reliability of the system.

Tests have been designed to cover various aspects of both modules. They serve not only as a means to verify the correctness of the implementation but also as regression tests to prevent the introduction of new errors in future development. This chapter is divided into two main sections: *Connector Tests* and *Data flow Generator Tests*. In these sections, we discuss the different tests for each module, their purposes, and examples of test scenarios.

6.1 Connector Tests

In this section, we describe the various tests designed to evaluate the correctness and robustness of the `manta-connector-databrickssql` module. All tests in the module extend the `AstBasicTest` class which is responsible for initiating the test scenarios. This structure allows for consistency and ease of extension across various tests.

AstResolverTest

The `AstResolverTest` is designed for debugging purposes. It assists in identifying issues during the AST resolution process and helps to verify the correctness of the implemented algorithms.

AstResolverBasicTest

The `AstResolverBasicTest` tests the error recovery mechanism implemented in the `manta-connector-databrickssql` module. It verifies that the system can detect and recover from errors in the input SQL scripts, so the unsupported statement or even part of it affects only that statement, not the entire script. It is also possible to test logs and test against invalid scripts.

AstInvariantsTest

The `AstInvariantsTest` is designed to check whether the AST contains all tokens from the input script. This test ensures that the input script can be reconstructed from the AST by matching the input script against the reconstructed one.

AnnotatedFilesResolverTest

The `AnnotatedFilesResolverTest` verifies that the AST does not contain error nodes and deducted entities unless explicitly allowed. This test class is designed for SQL scripts based on special annotations. These annotations in the test files describe an expected result after resolving a code, which allows for a more accurate assessment of the correctness of the `manta-connector-databrickssql` module.

We can see the example of a simple test input in Code snippet 43. The annotations closed in `/**/` comments specify checks to be done. For example, annotation `/*=t1a*/` checks that all the identifiers with this annotation refer to the same object in the dictionary. This test can also be used to verify the structure of a resulting AST.

```
create table t1/*=t1*/ (
    a/*=t1a*/ int,
    b/*=t1b*/ int
);

select  ta.b/*=t1b*/ as c/*=t1b*/,
        ta.a/*=t1a*/ as d/*=t1a*/
    from t1 /*=t1*/ ta
    where ta.a/*=t1a*/ = 5
    having ta.b/*=t1b*/ > 7
    LIMIT 5;
```

Code snippet 43: The example of `AnnotatedFilesResolverTest` input

6.1.1 PreprocessingTest

The `PreprocessingTest` is focused on evaluating the preprocessing step in the `manta-connector-databrickssql` module. This test ensures that the input SQL scripts are properly preprocessed and that any issues related to preprocessing are identified and resolved. It verifies preprocessing by substituting both parameters and variables. It ensures that the preprocessing result matches the expected output.

6.2 Data flow Generator Tests

The *Data flow Generator* module is responsible for building a data flow graph from the resolved Abstract Syntax Tree (AST). In this section, we describe the various tests designed to evaluate the correctness and robustness of this module.

AstFilesFlowTest

The `AstFilesFlowTest` is designed to test if the generated data flow graph matches the expected result. The test compares the output graph with an expected graph stored in a file with the `_expected.txt` suffix. The test also verifies the result of the filter task against an expected filtered graph stored in the file with the `_filtered_expected.txt` This test also ensures that the structure is correct even when introducing a new change.

ScriptPathTest

The `ScriptPathTest` serves two primary purposes. First, it tests the correct data flow node structure. Second, the `ScriptPathTest` is also designed for debugging purposes. Developers can identify and resolve potential issues in the Data flow Generator module by analysing the data flow node structure. We can also generate a visualisation of the data flow graph using this test using `-Dmanta.test.printGraph` option.

ErrorReportingTaskTest

The `ErrorReportingTaskTest` validates that errors can be correctly reported, thus ensuring trust in the rest of the tests, which rely on the correct error reporting.

Conclusion

In this thesis, we aimed to study the Databricks SQL dialect, its syntax and semantics, and analyse data flow in Databricks SQL scripts by extending the Manta tool. We successfully achieved our research objectives, gaining insights into the Databricks SQL language and implementing a prototype scanner unit for the Manta tool.

Our primary findings include the ability to analyse data flow in Databricks SQL scripts by extending the Manta tool, a development with practical implications for clients. As a result, clients can now examine data flow within their Databricks systems using the updated Manta tool, benefiting developers, data engineers, and other stakeholders working with Databricks SQL and the Manta project.

However, there are limitations to our research and implementation. The prototype scanner unit needs further development to handle features omitted from the prototype without failing on parsing or subsequent data flow generation.

Future work could thus extend the scanner and integrate it into the Manta tool to scan Databricks SQL queries. Additionally, implementing a transformation logic module for Databricks SQL would enable better analysis of expression logic within the SQL scripts, further enhancing the tool's capabilities.

There is also the possibility of performing better lambda function analysis to improve provided data lineage of lambda function usage.

In conclusion, this thesis contributes to understanding data flow analysis in Databricks SQL and enhancing the Manta tool. As we address the challenges and build upon the achievements presented in this work, we hope to positively impact developers, data engineers, and other stakeholders working with Databricks SQL and the Manta project.

Bibliography

1. *Ultimate Guide to Data Lineage* [online]. MANTA, 2022 [visited on 2023-04-16]. Available from: `https://getmanta.com/ultimate-guide-to-data-lineage`.

2. *About data lineage* [online]. Google, 2023-03 [visited on 2023-04-16]. Available from: `https://cloud.google.com/data-catalog/docs/concepts/about-data-lineage`.

3. HOPCROFT, John; ULLMAN, Jeffrey D. *Introduction to automata theory, languages, and computation.* CNIB, 1995.

4. MELICHAR, Bořivoj; ČEŠKA, Milan; JEŽEK, Karel; RICHTA, Karel. *Konstrukce překladačů.* Vydavatelství ČVUT, 1999. ISBN 80-01-02028-2.

5. AHO, Alfred V; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers: principles, techniques, and tools.* Vol. 2. Addison-wesley Reading, 2007. ISBN 0-321-48681-1.

6. *Minted – highlighted source code for latex* [online]. CTAN Atom, 2022-12 [visited on 2023-03-16]. Available from: `https://ctan.org/pkg/minted?lang=en`.

7. AHO, Alfred V.; ULLMAN, Jeffrey D. *The Theory of Parsing, Translation, and Compiling.* USA: Prentice-Hall, Inc., 1972. ISBN 0139145567.

8. PARR, Terence; FISHER, Kathleen. LL (*) the foundation of the ANTLR parser generator. *ACM Sigplan Notices*. 2011, vol. 46, no. 6, pp. 425–436.
9. *Introduction to data lakes* [online]. Databricks, 2022-05 [visited on 2023-03-20]. Available from: `https://www.databricks.com/discover/data-lakes/introduction`.
10. *What is a Data Lakehouse and answers to other frequently asked questions*. Databricks. Available also from: `https://www.databricks.com/blog/2021/08/30/frequently-asked-questions-about-the-data-lakehouse.html`.
11. *What is Delta Lake?* [online]. Databricks, 2023-02 [visited on 2023-03-20]. Available from: `https://docs.databricks.com/delta/index.html`.
12. MALIK, Rijul Singh. *Databricks vs spark: Introduction, comparison, Pros and Cons* [online]. MLearning.ai, 2022-08 [visited on 2023-04-29]. Available from: `https://medium.com/mlearning-ai/databricks-vs-spark-introduction-comparison-pros-and-cons-35958a1bd7e4`.
13. *Spark SQL, DataFrames and datasets guide* [online]. Apache Spark [visited on 2023-04-29]. Available from: `https://spark.apache.org/docs/2.2.0/sql-programming-guide.html`.
14. *Apache spark on databricks* [online]. Databricks, 2023-02 [visited on 2023-04-29]. Available from: `https://docs.databricks.com/spark/index.html`.
15. *Delta live tables SQL language reference* [online]. Databricks [visited on 2023-03-16]. Available from: `https://docs.databricks.com/workflows/delta-live-tables/delta-live-tables-sql-ref.html`.
16. *Configuration parameters*. Databricks, 2023-03. Available also from: `https://docs.databricks.com/sql/language-manual/sql-ref-parameters.html`.

17. *Develop code in Databricks Notebooks* [online]. Databricks, 2023-01 [visited on 2023-03-21]. Available from: `https://docs.databricks.com/notebooks/notebooks-code.html`.

18. *Data Objects in the databricks lakehouse.* Databricks, 2023-02. Available also from: `https://docs.databricks.com/lakehouse/data-objects.html`.

19. *PostgreSQL 15.2 documentation* [online]. The PostgreSQL Global Development Group, 2023-02 [visited on 2023-05-04]. Available from: `https://www.postgresql.org/docs/15/index.html`.

20. *SQL language reference* [online]. Databricks, 2023-03 [visited on 2023-04-25]. Available from: `https://docs.databricks.com/sql/language-manual/index.html`.

21. *Built-in functions* [online]. Databricks, 2023-03 [visited on 2023-04-16]. Available from: `https://docs.databricks.com/sql/language-manual/sql-ref-functions-builtin.html`.

22. *Functions* [online]. Databricks, 2022-11 [visited on 2023-04-25]. Available from: `https://docs.databricks.com/sql/language-manual/sql-ref-functions.html`.

23. *Alphabetical list of built-in functions* [online]. Databricks, 2022-11 [visited on 2023-04-16]. Available from: `https://docs.databricks.com/sql/language-manual/sql-ref-functions-builtin-alpha.html`.

24. *Query Data* [online]. Databricks, 2023-02 [visited on 2023-04-25]. Available from: `https://docs.databricks.com/data-governance/unity-catalog/queries.html`.

25. BUTANI, Harish; FRANCKE, Lars; LEVERENZ, Lefty. *Common table expression* [online]. Apache Software Foundation, 2014-09 [visited on 2023-04-29]. Available from: `https://cwiki.apache.org/confluence/display/Hive/Common+Table+Expression`.

26. *Migration guide: SQL, datasets and DataFrame* [online]. Apache [visited on 2023-04-29]. Available from: `https://spark.apache.org/docs/3.2.4/sql-migration-guide.html`.

27. *Language Manual DML* [online]. Apache Software Foundation, 2021-04 [visited on 2023-04-29]. Available from: `https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML%5C#LanguageManualDML-InsertingdataintoHiveTablesfromqueries`.

28. *What is Java technology and why do I need it?* [online]. Oracle, 2022 [visited on 2023-04-16]. Available from: `https://www.java.com/en/download/help/whatis_java.html`.

29. PARR, Terence. *Antlr V3 documentation* [online]. 2013. [visited on 2023-03-16]. Available from: `https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687234/ANTLR+v3+documentation`.

30. *Introduction* [online]. The Apache Software Foundation, 2022-12 [visited on 2023-04-16]. Available from: `https://maven.apache.org/what-is-maven.html`.

31. JOHNSON, Rod et al. *Spring Framework Overview* [online]. VMware, Inc., 2023-03 [visited on 2023-04-16]. Available from: `https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html`.

32. *JUnit* [online]. The JUnit Team, 2023 [visited on 2023-04-16]. Available from: `https://junit.org/junit5/`.

33. GAMMA, Erich; JOHNSON, Ralph; HELM, Richard; JOHNSON, Ralph E; VLISSIDES, John. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

Appendix A

Acronyms

ACID Atomicity, Consistency, Isolation, Durability

ANSI American National Standard Institute

ANTLR ANother Tool for Language Recognition

API Application Programming Interface

AST Abstract Syntax Tree

ATN Augmented Transition Network

AWS Amazon Web Services

DDL Data Definition Language

DFA Deterministic Finite Automata

DML Data Manipulation Language

IDE Integrated Development Environment

JAR Java ARchive

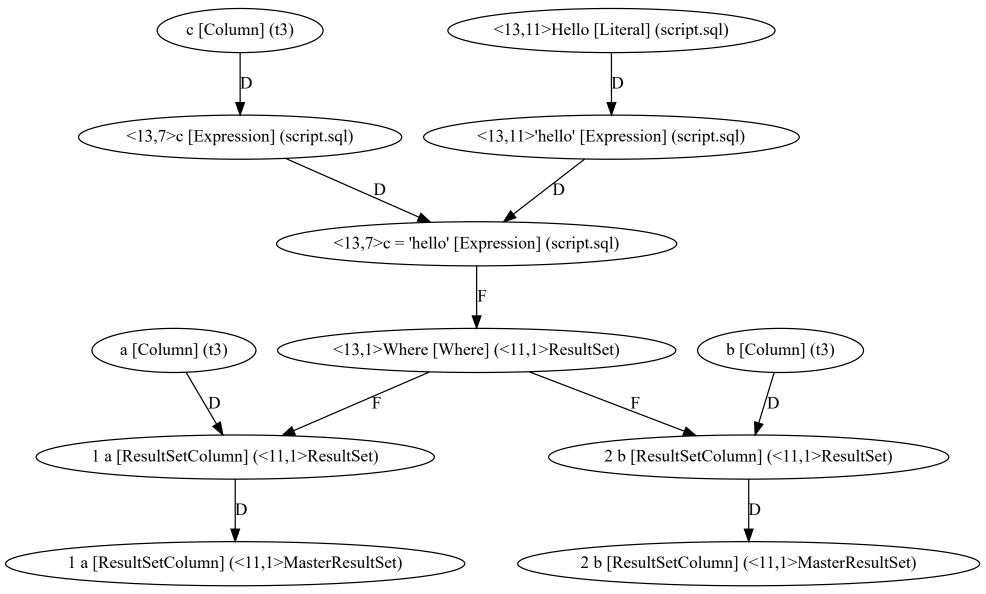
MPP Massively Parallel Processing

PDA Push-Down Automaton

SQL	Structured Query Language
UDF	User-Defined Functions
UML	Unified Modeling Language

Appendix B

Data flow graphs



c [Column] (t3)
<13,11>Hello [Literal] (script.sql)
D
D
<13,7>c [Expression] (script.sql)
<13,11>'hello' [Expression] (script.sql)
D
D
<13,7>c = 'hello' [Expression] (script.sql)
F
a [Column] (t3)
<13,1>Where [Where] (<11,1>ResultSet)
b [Column] (t3)
D
F
F
D
1 a [ResultSetColumn] (<11,1>ResultSet)
2 b [ResultSetColumn] (<11,1>ResultSet)
D
D
1 a [ResultSetColumn] (<11,1>MasterResultSet)
2 b [ResultSetColumn] (<11,1>MasterResultSet)

Figure B.1: Detailed data flow of the statement from Code snippet 20

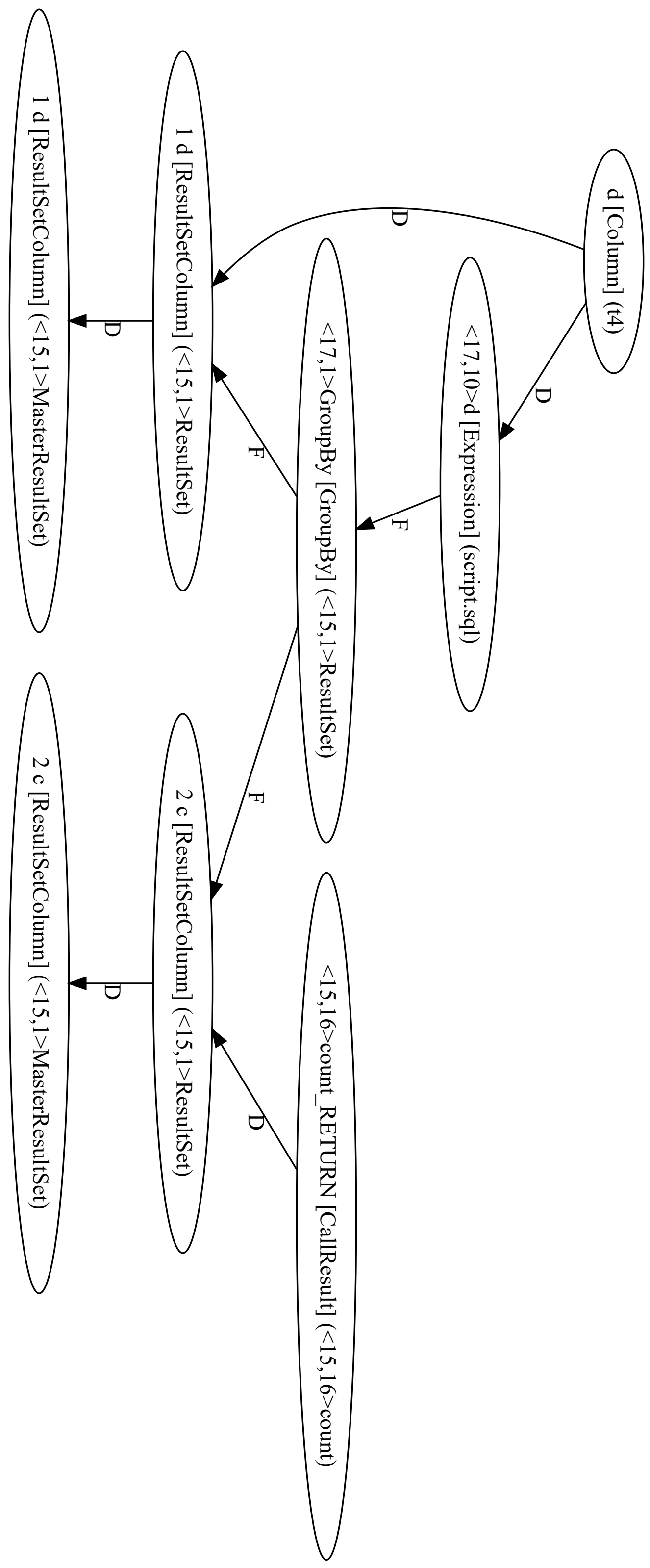

d [Column] (t4)
D
D
<17,10>d [Expression] (script.sql)
F
<17,1>GroupBy [GroupBy] (<15,1>ResultSet)
F
F
1 d [ResultSetColumn] (<15,1>ResultSet)
D
1 d [ResultSetColumn] (<15,1>MasterResultSet)
<15,16>count_RETURN [CallResult] (<15,16>count)
D
2 c [ResultSetColumn] (<15,1>ResultSet)
D
2 c [ResultSetColumn] (<15,1>MasterResultSet)

Figure B.2: Detailed data flow of the statement from Code snippet 21

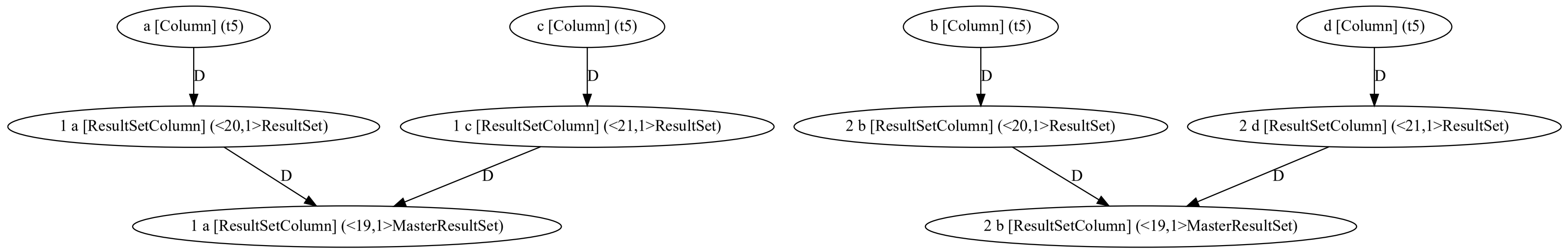

a [Column] (t5)
c [Column] (t5)
b [Column] (t5)
d [Column] (t5)
D
D
D
D
1 a [ResultSetColumn] (<20,1>ResultSet)
1 c [ResultSetColumn] (<21,1>ResultSet)
2 b [ResultSetColumn] (<20,1>ResultSet)
2 d [ResultSetColumn] (<21,1>ResultSet)
D
D
D
D
1 a [ResultSetColumn] (<19,1>MasterResultSet)
2 b [ResultSetColumn] (<19,1>MasterResultSet)

Figure B.3: Detailed data flow of the statement from Code snippet 22

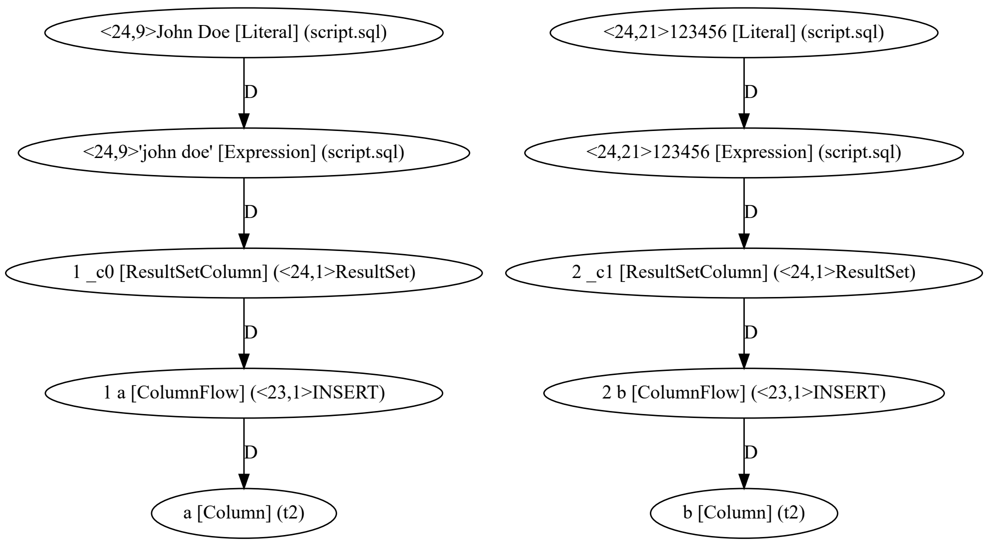

<24,9>John Doe [Literal] (script.sql)
D
<24,9>'john doe' [Expression] (script.sql)
D
1 _c0 [ResultSetColumn] (<24,1>ResultSet)
D
1 a [ColumnFlow] (<23,1>INSERT)
D
a [Column] (t2)
<24,21>123456 [Literal] (script.sql)
D
<24,21>123456 [Expression] (script.sql)
D
2 _c1 [ResultSetColumn] (<24,1>ResultSet)
D
2 b [ColumnFlow] (<23,1>INSERT)
D
b [Column] (t2)

Figure B.4: Detailed data flow of the statement from Code snippet 24

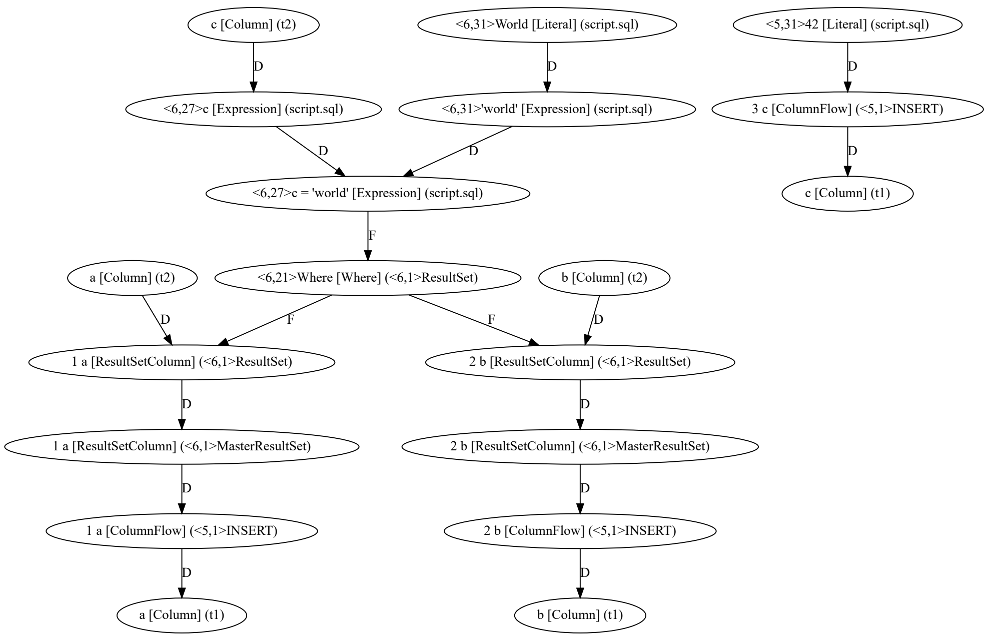

c [Column] (t2)
<6,31>World [Literal] (script.sql)
<5,31>42 [Literal] (script.sql)
D
<6,27>c [Expression] (script.sql)
<6,31>'world' [Expression] (script.sql)
3 c [ColumnFlow] (<5,1>INSERT)
D
<6,27>c = 'world' [Expression] (script.sql)
c [Column] (t1)
F
a [Column] (t2)
<6,21>Where [Where] (<6,1>ResultSet)
b [Column] (t2)
1 a [ResultSetColumn] (<6,1>ResultSet)
2 b [ResultSetColumn] (<6,1>ResultSet)
1 a [ResultSetColumn] (<6,1>MasterResultSet)
2 b [ResultSetColumn] (<6,1>MasterResultSet)
1 a [ColumnFlow] (<5,1>INSERT)
2 b [ColumnFlow] (<5,1>INSERT)
a [Column] (t1)
b [Column] (t1)

Figure B.5: Data flow of the statement from Code snippet 25

Appendix C

Contents of attachments

```
text........................................the thesis text directory
 ├─ thesis.pdf..........................the thesis text in PDF format
 └─ thesis.rar.............................the thesis text source code
```