FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Data flow analysis of scripts in SAP Hana SQL dialect
Student:	Bc. Ondřej Hlaváč
Supervisor:	Ing. Jan Trávníček, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Study the SAP Hana dialect of SQL language – its syntax and semantics.
Familiarize yourself with the Manta project and the way it represents data flow.
Analyze whether it is possible to retrieve information about data flow between data structures in an SAP Hana database from scripts in SAP Hana language using static analysis of these scripts.
Propose an approach to parsing and representing SAP Hana scripts. Describe its statements relevant to the later data flow analysis.
Continue with a design of analysis of SAP Hana scripts capable of retrieval of the data flow between data structures in SAP Hana database from SAP Hana scripts and an approach to represent this data flow in the Manta project.
Create a prototype implementation of a tool that is able to retrieve data flow between data structures in an SAP Hana database from SAP Hana scripts and is able to store this data flow in the Manta project.
Propose and implement testing of your prototype.

Electronically approved by Ing. Michal Valenta, Ph.D. on 7 October 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Data flow analysis of scripts in SAP Hana SQL dialect

Bc. Ondřej Hlaváč

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

May 5, 2022

Acknowledgements

I would like to express my gratitude to everyone who helped me or supported me to complete my thesis. First and foremost, I would like to thank my supervisor Ing. Jan Trávníček, Ph.D., for his help, time and guidance. Invaluable help also came from the whole Manta team, notably, I would like to thank the parser team leader Mgr. Jiří Toušek. My appreciation also goes towards my family and friends for all their support, not only during the time I was working on this thesis but also during the whole university studies.

Declaration

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

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

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

In Prague on May 5, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Hlaváč, Ondřej. *Data flow analysis of scripts in SAP Hana SQL dialect.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstract

This thesis deals with the analysis of data flows specified in the SQL statements used by the SAP Hana database. First, it describes what data lineage is and why it is useful, then the different data lineage retrieval approaches. After that, it shows selected SAP Hana's SQL statements and their data flow. It also describes the design and implementation of a prototype tool to automatise the analysis as a component of the Manta Flow. This prototype tool can generate graphs describing the data movements/flows of given SQL scripts.

Keywords SAP, SAP Hana, Hana, SQL, parsing, Manta, Manta Flow, data flow, dataflow, data warehouse

Abstrakt

Tato práce se zabývá analýzou datových toků specifikovaných v příkazech SQL dialektu využívaným databází SAP Hana. Nejdříve popisuje, co data lineage je, a různé postupy jejího získání. Dále popisuje jednotlivé SQL příkazy a jejich datové toky. Zahrnuje taktéž návrh a implementaci prototypového nástroje, integrovaného do Manta Flow, který dokáže tvořit grafy na základě poskytnutých vstupních skriptů.

Klíčová slova SAP, SAP Hana, Hana, SQL, parsování, Manta, Manta Flow, datové toky, datový sklad

Contents

List of Figures

List of Tables

Introduction

Companies nowadays need complex systems to stay competitive and see detailed business reports. The reporting is commonly done through multiple systems and, sometimes, can lead to conflicting information from various data sources. That is why data analysts try to automatise processes. [16] Automatised processes are better than manual processing but can lead to the loss of data origin information. This scenario can be seen especially in large data warehouses.

Tracking the data movement is called data lineage. Data lineage is a vital part of data warehouses. Data warehouses are complex systems comprising multiple stages, databases, and tools. Businesses typically need to know where the data from each system are consumed for legal or internal auditing reasons. Mapping the lineage might be challenging as it relies on documentation of the scripts. That is prone to human error since the documentation can differ from the actual scripts. To solve this, project Manta with its Manta Flow tool automates this process by analysing the warehouse's databases, scripts, and ETL tools.

Chapter 1

Theoretical background

This chapter describes the theoretical concepts required for understanding the technologies used in this thesis.

1.1 Graph

A graph G is an ordered triple $(V(G), E(G), \Psi_G)$ consisting of a nonempty set $V(G)$ of vertices, a set $E(G)$, of edges, disjoint from $V(G)$, and an incidence function Ψ_G, a that associates with each edge of G an unordered pair of (not necessarily distinct) vertices of G. If e is an edge and u and u are vertices such that $\Psi_G(e) = (uv)$, then e is said to join u and v; the vertices u and v are called the ends of e. [3]

Graphs can be represented graphically. Vertices can be shown as circles and edges as lines connecting two vertices. The listing 1.1 shows an example definition of a graph that can be represented graphically, as shown in the figure 1.1.

```
1 G = (V(G), E(G), Ψ_G)
2 V(G) = A, B, C
3 E(G) = e_1, e_2
4 Ψ_G(e_1) = AB
5 Ψ_G(e_2) = BC
```

Listing 1.1: Example graph G

1.1.1 Oriented graph

The aforementioned graph definition defines edges as non-directional. The so-called unoriented graph definition might be sufficient for many applications, but this thesis requires an extended version – the oriented graph. Oriented graphs also have vertices, edges, and an incidence function, but the value of the incidence function is an ordered pair. The graphical representation can

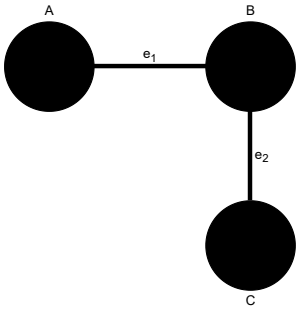

Figure 1.1: Possible graphical representation of the graph 1.1

reflect the direction as an arrow on one side of the line representing a (directed) edge.

1.2 Language

An arbitrary set of chains (that is, words, cf. Word) over some (finite or infinite) alphabet V (sometimes also called a dictionary), that is, a set of expressions of the form $\omega = a_1...a_k$*, where* $a_1...a_k \in V$*; the number k, usually denoted by* $|\omega|$*, is the length of the chain* ω*. One also considers the empty chain, denoted by* ϵ*; one sets* $|\epsilon| = 0$*.* [11]

1.3 Context-free grammar

Context-free grammar is a 4-tuple of $G = (V, \Sigma, P, \sigma)$, where

1. V is a finite non-empty set
2. Σ is a non-empty subset of V
3. P is a finite non-empty set of pair (ξ, Υ), with ξ in $V - \Sigma$ and Υ in V^*
4. σ is an element of $V - \Sigma$

- Each element of $V - \Sigma$ is called a variable.
- Each element of Σ is called a (terminal) letter.
- Each element (ξ, Υ) in P is called a production rule and is written $\xi \to \Upsilon$.
- σ is called a start variable, and represents whole language.

1.3.1 Derivations

Let G be a Context-free grammar defined in 1.2. The grammar is a set of variables, terminal symbols and rules for generating new symbols.

```
1 G = (V, Σ, P, σ)
2 V = {S, A, B, c, d}
3 Σ = {c, d}
4 P = {S → AB, A → Ac, A → d, B → cBd, B → c}
5 σ = S
```

Listing 1.2: Example grammar G

The rules replace the non-terminals (defined on the left side of the rules) with the right side. The start symbol represents all the strings the grammar can generate. The example 1.3 uses the rules. One usage of a rule is called a derivation step. [7]

If the leftmost non-terminal symbol only is being replaced, the derivation is called leftmost derivation.

```
1 S => AB => AcB => dcB => dccBd => dcccd
```

Listing 1.3: Example derivation in the grammar G

Determining whether a grammar can generate some input can be achieved by finding a derivation. The construction of derivations is referred to as parsing. [7]

1.3.2 LL(k) grammar

For a natural number $k \geq 0$, a context-free grammar $G = (V, \Sigma, P, \sigma)$ is an **LL(k) grammar** if

- for each terminal symbol string $\omega \in \sum^*$ of length up to k symbols,
- for each nonterminal symbol $A \in V$, and
- for each terminal symbol string $\omega_1 in \sum^*$,

there is at most one production rule $r \in R$ such that for some terminal symbol strings $\omega_2, \omega_3 \in \sum^*$,

- the string $\omega_1 A \omega_3$ can be derived from the start symbol S,
- ω_2 can be derived from A after first applying rule r, and
- the first k symbols of ω and of $\omega_2\omega_3$ agree.

[2]

1.3.3 Context-free language

$L \subseteq \Sigma^*$ is a context-free language if there exists a context-free grammar $G = (V, \Sigma, P, \sigma)$ such that $L = L(G)$, where $L(G) = \{w \in \sum^* \mid \sigma \rightarrow^* w\}$ is said to be the context-free language generated by G. [1] If the grammar is LL(k), then the language generated by the production rules is called restricted Context-free Language.

1.3.4 Backus-Naur form

Programming languages are commonly designed to conform to Context-free grammar, and as such, can they be described as one, using the same notation. However, to simplify the notation, the Backus-Naur form (BNF) notation is used. Because the Backus-Naur form describes programming languages, it is sometimes called a metalanguage. The syntax is very similar to the grammar description notation – it differs because it distinguishes the non-terminal symbols in the notation (enclosing them in <and >). The symbol ::= is used to indicate metalinguistic equivalence – used in replacing non-terminal symbols on the left with the right side. [6] The example 1.4 shows a possible integer representation in the BNF notation.

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<unsigned_integer> ::= <digit> | <unsigned_integer><digit>
<integer> ::= <unsigned_integer> | + <unsigned_integer> | - <
    unsigned_integer>
```

Listing 1.4: Example number definition in BNF notation

1.3.4.1 Extended Backus-Naur form

The Extended Backus-Naur form (abbreviated EBNF) is an addition to the original BNF. It changes the notation for terminal and non-terminal symbols as terminal symbols are denoted with quotes, and non-terminal symbols are not enclosed. These changes have advantages in that the described language may contain the symbols of the metalanguage. The extensions also add some additional syntax for shortening the rules, such as the repetition notation. EBNF also denotes the end of the rule (by a semicolon) so that it is not ambiguous and explicitly marks rule concatenation by a comma symbol. For the complete reference, see [22]. The example 1.5 shows the exact integer definition as the BNF 1.4. As it is apparent, the definition is shorter and much more human-readable.

```
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
    '9';
unsigned_integer = {digit}, digit;
integer =  [('+' | '-')], unsigned_integer;
```

Listing 1.5: Example number definition in EBNF notation

Symbol	Description
<>	Angle brackets are used to surround the name of a syntactic element (BNF nonterminal) of the SQL language.
::=	The definition operator is used to provide definitions of the element appearing on the left side of the operator in a production rule.
[ ]	Square brackets indicate optional elements in a formula. Optional elements can be specified or omitted.
{ }	Braces group elements in a formula. Repetitive elements (zero or more elements) can be specified within brace symbols.
|	The alternative operator indicates that the portion of the formula following the bar is an alternative to the portion preceding it.
[...]	Ellipsis with square brackets around it indicates optional repetition of the preceding element or grouped elements.
!!	Introduces standard English text. This symbol is used when the definition of a syntactic element is not expressed in BNF.

Table 1.1: SAP EBNF notation [14]

1.3.4.2 SAP Extended Backus-Naur form

SAP defines its version of EBNF to document syntax for the SAP Hana. Its terminal and non-terminal symbol notation come from the BNF with the extensions from EBNF. The table 1.1 lists the notation conventions. The example 1.6 shows the exact integer definition as the EBNF 1.5 and the BNF 1.4.

```
1 <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
2 <unsigned_integer> ::= <digit>...
3 <integer> ::=  [{+ | -}] unsigned_integer
```

Listing 1.6: Example number definition in SAP EBNF notation

1.4 Finite-state automaton

Automatons are concept machines with a defined input set of symbols, called the alphabet, that either accept a string (finite sequence of symbols from the alphabet) or not. Formally, it is a 5-tuple $A = (S, A, i, \delta, T)$, where S is a set of states, A is the finite input alphabet, i is the initial state (element of S),

δ is a transition function (S x A $\rightarrow$ S), and T is a set of terminal states (a subset of S).[5]

The automaton then keeps track of the current state and the unread symbols from the input. It begins the processing in the initial state and on the first symbol of the input. Deciding whether to accept the input uses the transition function to compute the next state upon reading the input. The transition function defines the next state for a pair of the current state and input. If the input contains no more symbols and the current state belongs to the set of terminal states, the input is accepted. Otherwise, the automaton rejects the input (if it cannot continue applying the transition function).

1.4.1 Push down automaton

Push down automaton is an extension of the finite State automaton – adding a pushdown store that can be used during the process of deciding whether the input is accepted or not. Formally, it is a 6-tuple $A = (S, A, \Gamma, \delta, Z_0, q_0)$, where S is a set of states, A is the finite input alphabet, Γ is a finite non-empty set of push down symbols, δ is a transition function (S x (A $\cup \epsilon$) x $\Gamma \rightarrow$ S x Γ^*), Z_0 is an element of Γ and starting symbol on the tape, and q_0 is the initial state (element of S). [1] The defined Push Down Automaton accepts the input when all of the symbols in the pushdown store have been processed and the store is empty.

1.5 Parsing

Parsing is an action of analysing a string of symbols according to formal grammar rules. The concrete implementation of the parser required depends on the language being analysed. This thesis deals with a programming language defined in BNF-like notation containing recursive elements, requiring context-free grammar [7].

It turns out that given any context-free grammar, there is a Push down automaton (PDA) that accepts precisely the language of the grammar. Tools that take a string and determine whether or not it is a sentence are called recognisers. [7]

When the language has been described by context-free grammar, there exist rules for the string generation. If the recogniser for that language can record the rules used for a particular string, it is called a parser.

1.5.1 LL parser

LL parser is a parser that analyses restricted Context-free language by derivating left-to-right and leftmost.

1.5.1.1 Lookahead

Lookahead is a property of a parser that can peek at the unread input and decide which rule to use while derivating. This feature is then used to choose between ambiguous derivation steps (more than one rule is applicable to the input).

1.5.1.2 LL(k) parser

LL(k) parser is a LL parser with a lookahead of $k \geq 0$ characters.

1.5.1.3 LL(*) parser

LL(*) parser uses regular expressions rather than a fixed constant or backtracking [1] *with a full parser to do lookahead. The analysis tries to construct a deterministic finite automaton for each nonterminal in the grammar to distinguish between alternative productions (rules). If the analysis cannot find a suitable DFA for a nonterminal, it fails over to backtracking.* [4]

[1] More than one derivation step option; the parser tries all of them by returning to the conflicting state when the derivation does not succeed.

Chapter 2

Technologies description

2.1 Data lineage

Data lineage maps all direct and indirect data flows in a system between data entities [21]. In the context of a data warehouse (which is a typical use for SAP Hana database [15]), this means describing the origins of data in reports. Data lineage can be represented in natural language, used by human data analysts, or by a graph. The data lineage graph is always an oriented graph that shows data entities as nodes and flows as edges. For purposes of this thesis, the edges of the graph will be distinguished into two groups – direct flows that represent the movement of data from one entity to another and filter flows that mean filtering of data (but not direct movement).

2.2 Manta

This thesis uses the framework of the Manta Flow tool. Manta Flow is a utility that analyses source codes of many databases, ETL tools, programming languages, and others. The tool internally represents data movements between databases and systems. Then Manta Flow shows them as a graph of data lineage.

Manta Flow comes with a framework for working with different systems – it provides tools for storing objects (data dictionary), helper methods for working with some common processing tasks, or graph representation and generation. The data dictionary may also add additional context from the Dictionary extractor module. Manta has a dictionary extractor for the SAP Hana database – implemented as a part of [17]. The Dictionary extractor provides knowledge of tables, types, schemas, or other database objects stored in the database.

Manta Flow usually runs analysis on multiple systems and merges the resulting graphs – providing a complete picture of data movement throughout complex and diverse systems. The graph processing step also involves graph

cleanup as the output of this tool involves information not needed for simple data lineage – such as constants and variables.

2.3 SAP Hana

SAP Hana is a tenant, in-memory, column-store database with built-in online and analytical data processing features. It also includes ETL capabilities and an application server. [19]

Its features allow for high performance in BI (business intelligence) and make it suitable for warehouse usage. The database engine is very modular and allows a wide range of configurations, such as switching the table storage from column-store to row-store, multiple snapshotting (saving the memory to a hard drive), graph processing, or running application containers.

SAP Hana uses a dialect of the SQL called `SQLScript`. As the name says, this language also comes with scripting capabilities used in functions or procedures. Scripting is widely used in SQL dialects and adds complexity to the data lineage. Its advanced features allow developers to move some business logic into the database and save processing time by reducing data movement and optimisations in the database engine.

2.4 Acquiring data lineage

Data lineage can be acquired in multiple ways. There is even an option not to care about it at all. [13] describes possible scenarios used.

2.4.1 Business data lineage

It is a valid option to build lineage by asking the people working on a project what data are used or exported. This method may introduce issues such as incorrect people's knowledge, two people contradicting themselves or even missing part of the lineage.

2.4.2 Manual analysis

When working on a relatively small project, it is possible to read all relevant source code and write down what data is used and exported. This approach also has some significant drawbacks – namely, requiring expensive and skilled human workers that can read the code and decide. Alternatively, the need to keep the lineage updated as many systems undercome changes through their lifetime – may put the existing lineage out of sync with the system and introduce errors. And lastly, as before – human error as in the previous method.

2.4.3 Tracing/tagging

A more automated method may be to tag the data in the source systems and then look for those tags on the output. This solution may look like a great solution, but the main issue is integrating multiple systems as the engine of the execution environment must track all data movements. Moreover, this does not show a complete lineage for every run. It only shows *execution lineage*. That means it may generate a completely different lineage for every run if there are some conditional switches for data sources that depend on the data shown.

2.4.4 Self lineage

Another possible way is to use only one engine for every transformation with built-in data lineage generation capabilities. Self lineage is the most stable and error-prone method since every transformation and data movement is controlled by a single entity. The apparent disadvantage comes from the same feature – it is impossible to use any other system. This fact means that it is constrained to the capabilities of a single system.

2.4.5 Decoded lineage

Building a complete lineage requires reading all of the sources and understanding them. That is the goal of the Manta project, and this thesis is a part of it. Reading source codes and building data lineage from it comes with drawbacks as well – it is not possible to show the execution lineage (lineage of the current output). Only the static lineage is possible. It is also very demanding to implement data linage analysis as it requires building a parser and analyser for every given language used by each system. Furthermore, with the growing complexity of each system, it is unfeasible for companies that use the systems (databases and ETL tools in the case of Manta) to automatise the analyses by themselves.

2.5 Static code analysis

Reading and analysing the scripts is necessary when decoded analysis of the data lineage. That process is called static code analysis. Static code analysis is a process of analysing a program's code without running it. It is performed by compilers and code analysers that try to find patterns that lead to errors. They also use a technique called data-flow analysis that keeps track of data passed through variables and operations. [8] That is useful for data lineage as the data movement can be shown in a graph.

The process itself is comprised of **Lexical analysis**, **Syntax analysis**, and **Semantic analysis**. The figure 2.1 shows the inputs and outputs of each phase.

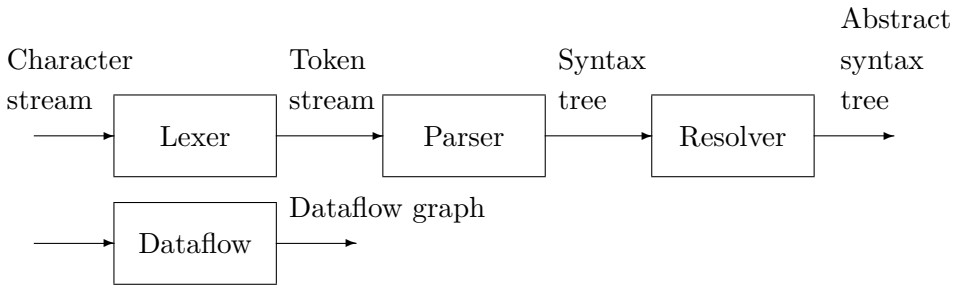

Figure 2.1: Analysis process

2.5.1 Lexical analysis

Lexical analysis (Lexer) is the first part of the analyser. The input is a raw character stream. The analyser has definitions of groups of characters, called *lexemes*. The benefit behind this step is to reduce and simplify the input – the Lexer removes irrelevant parts (such as whitespaces or comments). Further analyser steps then do not have to include those in their language representation. The lexer itself has a a definition of the valid tokens in the language, usually by regular expressions. [12]

In the case of SQL, the lexer should also differentiate between reserved and non-reserved keywords as the parser uses that information. Reserved keywords can not be used as names for objects (identifiers). A more detailed description of reserved words is in the dialect section 3.3.6.

The figure 2.2 shows how the lexer tokenises the sample SELECT statement.

SELECT	a	,	b	FROM	table	;
Keyword	ID	COMMA	ID	Keyword	ID	SC

Figure 2.2: Example SELECT statement lexer processing

2.5.2 Syntax and semantic analysis

The token stream from the lexer is then passed to the parser that either accepts or discards the input based on the language grammar. In the later stages, it then builds a representation of the given input as the Syntax Tree or Abstract Syntax Tree (AST). The Syntax Tree transformation to the Abstract Syntax Tree usually means discarding some irrelevant syntax that is not useful for later analysis or adding some details that can be then used. [10]

The figreffig:selectparser shows a sample AST built from the example 2.2.

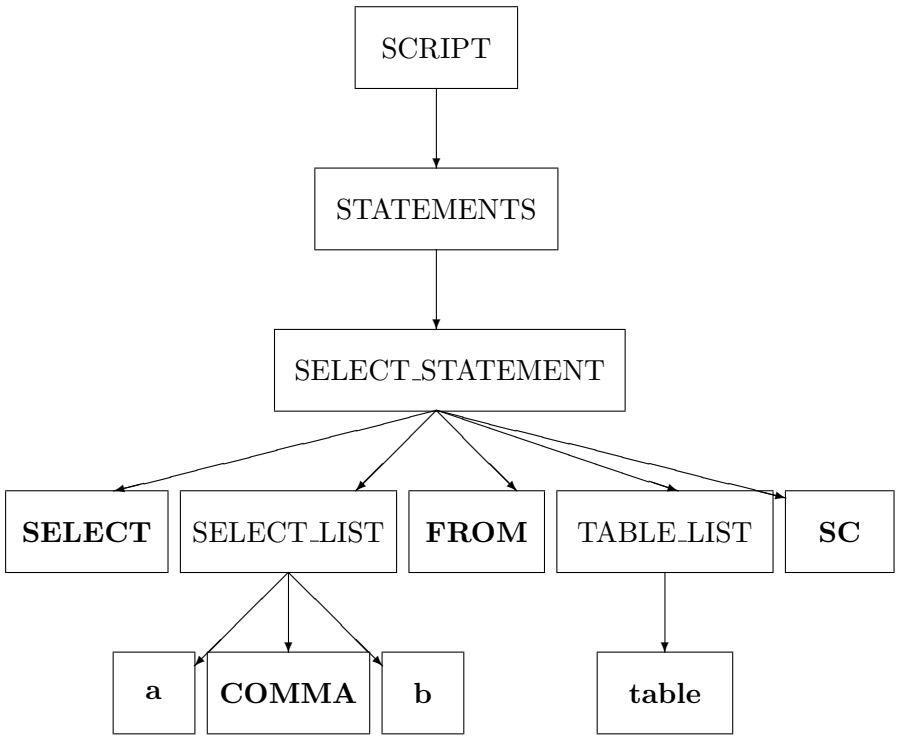

Figure 2.3: AST of the example SELECT statement 2.2

2.5.3 Generation of data flows

The AST created by the parser contains so-called AST nodes. AST node is some logical unit generated by the Parser – the most notable logical unit in

SQL is the SQL statement. The nodes are used as a reference for the data flows. In case of the example 2.3 `SELECT_STATEMENT` node indicates that there are data flows from the sources/tables inside `TABLE_LIST` node to the nodes inside `SELECT_LIST` node. This structure allows the module to create graph nodes for data lineage and connect them. Of course, this example is very simplified and to get accurate results, the data flow generator needs context and standardised interfaces for the nodes.

Chapter 3

Analysis

This chapter presents the analysis of the requirements for the tool. First, it describes functional and non-functional requirements in the first subsection. Then, the second section shows what technologies are used. The last part introduces the SQL dialect used (SQLScript).

3.1 Requirements analysis

This section lists the functional and non-functional requirements of the prototype script analysis tool. The requirements come from the Manta's needs as Manta Flow can scan databases as stand-alone or scan the whole environment – as in scripts that may be embedded in some other system (e.g. SAP Hana scripts in other database's scripts). The tool should be then used as a plug-in module for Manta Flow.

3.1.1 Functional requirements

3.1.1.1 FR1: Script analysis

The tool shall analyse SQLScript/SAP Hana scripts and generate a data lineage graph describing data movement between data entities used in SAP Hana.

3.1.1.2 FR2: Input

The tool shall accept inputs in the form of files and/or strings.

3.1.1.3 FR3: Data flow processing

The tool shall process both direct and indirect data flows.

3.1.1.4 FR4: Error processing

The tool shall inform the user when an error has been encountered and should recover at least on the following statement.

3.1.2 Non-functional requirements

3.1.2.1 NFR1: Manta framework

The tool shall integrate with the Manta framework. This means using compatible language and conventions.

3.1.2.2 NFR2: Time complexity

The tool shall take an appropriate amount of time to analyse scripts.

3.2 Technologies used

3.2.1 Java

The programming language of choice is the Java programming language. There are multiple advantages and drawbacks to Java. Advantages may include significant prevalence amongst software developers, according to [18]. Widespread language usage makes for easy access to mature libraries and tools for automating some parts of the development process. Java is also multiplatform, bytecode-compiled language, resulting in great compatibility with different systems. The main disadvantage remains in lower performance, but as with other Manta's modules, this does not seem to be that much of an issue for this use case. Most performance issues in script analyses can be solved either by improving the algorithms or by making the analysis run in parallel.

3.2.2 ANTLR

ANTLR is a parser generator that supports Java. ANTLR dramatically reduces the complexity of the code as it allows for efficient and straightforward representation of the language in the grammar that is then used for code generation of the parser in the selected language. It uses LL(*) parsing algorithm to generate the parser.

ANTLR uses a syntax similar to EBNF, simplifying the process of specifying the language. Other data flow analyser modules already use ANTLR in Manta Flow.

3.2.3 Apache Maven

Apache Maven is a dependency and build manager for Java/JVM languages. Building software is a complex process for any modern program, and with the

increasing number of dependencies, managing versions becomes very hard to ensure. For that, Apache Maven has been chosen. It allows for defining build steps that are very useful for the parser generation from ANTLR.

3.3 SAP Hana SQL dialect

SAP Hana uses a dialect of the Structured Query Language (SQL – **ISO/IEC 9075**). The query language is similar to other SQL dialects (such as PostgreSQL or T-SQL used by the Microsoft SQL Server). Definitions in this section use the SAP Extended Backus-Naur form, and some of them are simplified for the usage of this thesis. Simplified definitions do not contain all clauses, or the rules are not entirely defined. They do have only the necessary clauses for the data flow analysis.

3.3.1 SQLScript

The norm for Structured Query Language – **ISO/IEC 9075** defines only declarative-style statements. This results in unnecessary copying of data and inefficiency as most programming languages are imperative. They work with tuples (for example, looping over table record by record). This programming style is straightforward to understand and very hard to optimise for the compiler. SAP Hana tries to solve this by extending SQL with imperative statements to ease the development. After executing queries with the extended features, the query optimiser transforms the operations into a data-flows execution plan. Data-flow execution plans can often be executed in parallel and significantly reduce execution times on modern machines. SAP Hana especially benefits from this as it is a memory-based database, and hence the storage is not a bottleneck. [20]

SAP Hana SQL scripts are comprised of statements delimited by the semicolon symbol. [14]

The dialect is not case sensitive, except for the delimited identifiers 3.3.3 – all keywords can be used in lower case or upper case with no impact on semantics.

3.3.2 Data types

Data type characterises the data saved in the database. It defines how the values are represented. In SAP Hana, there are different classes of data types. Certain data types might be required in the context of some expressions (for example, indexing arrays). See the table 3.1 for the complete reference. Note that some data types have aliases that are not listed in the documentation [14]. Data types are used when defining data structures that store data – such as columns of a table or variables.

Classification	Data type
Datetime types	DATE, TIME, SECONDDATE, TIMESTAMP, LONGDATE, DAYDATE
Numeric types	TINYINT, SMALLINT, INTEGER, INT, BIGINT, SMALLDECIMAL, DECIMAL, DEC, REAL, DOUBLE, FLOAT
Boolean type	BOOLEAN
Character string types	VARCHAR, NVARCHAR, ALPHANUM, SHORTTEXT, CHAR, NCHAR
Binary types	VARBINARY
Large Object types	BLOB, CLOB, NCLOB, TEXT, BINTEXT, VARBINARY
Multi-valued types	ARRAY
Spatial types	ST_GEOMETRY, ST_POINT

Table 3.1: Data types and data type classes [14]

SAP Hana also allows creating custom data types with the `CREATE TYPE` statement. Custom types behave almost precisely like tables, but they don't contain any data. This means that tables can also be used as custom types – for function parameters or variables.

3.3.3 Identifiers

Identifiers represent names of database objects like tables, views, or users. SAP Hana, like most SQL dialects, differentiates delimited and undelimited identifiers. Undelimited identifiers must start with a letter and must contain only letters, digits, or underscore symbol. Delimited identifiers are enclosed in double quotes and can contain any character. Apart from that, delimited identifiers are case-sensitive. Meaning that 3.1 and 3.2 are two different tables that can coexist in one schema. On the contrary, 3.1 and 3.3 refer to the same table and can not be executed in the same schema. The reason for that is that undelimited identifiers are normalised to upper-case, so the table name in 3.1 is saved in the database as `TABLE`. [14]

The identifiers can contain multiple segments. The period symbol splits segments. 2-segment identifier refers to the schema. 3-segment identifier refers to the (remote) database, schema, and object.

```
1 CREATE TABLE Table (c int);
```

Listing 3.1: Create table statement with undelimited identifier 1

```
1 CREATE TABLE "Table" (c int);
```

Listing 3.2: Create table statement with delimited identifier

```
1 CREATE TABLE TABLE (c int);
```

Listing 3.3: Create table statement with undelimited identifier 2

3.3.4 Functions

Functions are blocks of code that can be reused in multiple places without repeating the code. Functions have parameters and return values. Return values have multiple types – scalar and tabular. This differs from the way functions output data. Scalar functions output either one value or numerous single-value variables. Multiple return variables are not a common approach in SQL dialects, as most SQL databases only allow one output value. Tabular functions return a table.

Functions also differ between parameters – there are **IN**, **OUT**, and **INOUT** parameters. **IN** parameter is used as an input and any value is discarded after the execution ends. **OUT** parameter pass the value at the end of execution back, outside of the function body scope, to the variable in the function call. **INOUT** is a combination of both **IN** and **OUT** parameters – it inputs value and returns it back.

There are two kinds of functions – user-defined and built-in. User-defined functions have a defined body in a supported language. Built-in functions are present in the database engine by default and may not have a definition in any scripting language (they can be implemented outside of the script environment).

3.3.5 Expressions

Expressions are clauses that can be evaluated to temporary constant values. [14] Expressions are used to make calculations or some other data transformations. It may be helpful to show what transformations were used on the data in data lineage.

There are multiple types of expressions.

3.3.5.1 Simple expressions

Simple expressions comprise constant values (such as number literals or string literals) and operators performing operations on them.

There are two kinds of operators in SAP Hana – Unary and Binary. Unary operators apply to one operand [2]. Binary operators apply to two operands [2].

[2]Target of the operation.

Operator	Usage
Unary plus operator	+<expression >
Unary minus operator	-<expression >
Logical negation operator	NOT(<expression >)

Table 3.2: Unary operators [14]

Operators	Usage
Multiplicative operators	expression*expression expression/expression
Additive operators	expression+expression expression-expression
Comparison operators	expression=expression expression!=expression expression<expression expression>expression expression<=expression expression>=expression
String concatenation	expression||expression

Table 3.3: Binary operators [14]

Tables 3.2 and 3.3 list the unary and binary operators defined in SAP Hana, respectively.

Operators have defined precedence – the order in which they are evaluated when there are multiple operators in one expression. It is possible to enclose the expression in parenthesis to change the precedence.

3.3.5.2 Case expressions

Case expressions are conditional clauses that can be used to choose between values based on conditions. The listing 3.4 shows the syntax. This statement has multiple branches as values and one condition.

```
1 <simple_case_expression> ::=
2             CASE <expression>
3             WHEN <expression> THEN <expression>
4             [{ WHEN <expression> THEN <expression>}...]
5             [ ELSE <expression>]
6             END
```

Listing 3.4: Case expression definition in the SAP EBNF notation [14]

3.3.5.3 Function expressions

Functions expressions are references to functions. The process of referencing a function is called *calling*. Function call requires parameter values. The listing 3.5 shows the syntax of the function expressions.

```
<function_expression> ::= <function_name> ( <expression> [{, <
   expression>}...] )
```

Listing 3.5: Function expression definition in the SAP EBNF notation [14]

3.3.5.4 Aggregate expressions

Aggregate expressions use aggregate functions to calculate a single value from multiple rows. They need to be addressed separately in the parser as they come with a special syntax. The functions can only be used when the GROUP BY clause is used in the SELECT statement 4.

```
<aggregate_expression> ::=
 COUNT(*)
 | COUNT ( DISTINCT <expression_list> )
 | <agg_name> (  [ ALL | DISTINCT ] <expression> )
 | STRING_AGG ( <expression> [, <delimiter>] [<
    aggregate_order_by_clause>])

<agg_name> ::= CORR | CORR_SPEARMAN | COUNT | MIN | MEDIAN | MAX
    | SUM | AVG | STDDEV | VAR  | STDDEV_POP | VAR_POP |
    STDDEV_SAMP | VAR_SAMP
<delimiter> ::= <string_constant>
<aggregate_order_by_clause> ::= ORDER BY <expression> [ ASC |
    DESC ] [ NULLS FIRST | NULLS LAST]
```

Listing 3.6: Aggregate expression definition in the SAP EBNF notation [14]

3.3.5.5 Subqueries

`A subquery is a SELECT statement enclosed in parentheses. The SELECT statement can contain no more than one select list item. A scalar subquery can only return a zero or a single value when used as an expression.` [14] The data flows here are processed recursively from the enclosed SELECT statement 3.3.7.1. The graph then outputs the single select item.

3.3.6 Reserved words

Reserved words are words with special meaning to the SAP Hana parser and can not be used as undelimited identifiers. Reserved words reduce the complexity of parsers and grammar definitions, limiting the number of alternatives for the possible parse tree. The complete list can be found in the documentation [14].

3.3.7 Statements

Each statement of the language has some functionality and usually some form of switches to configure additional options. This section lists SQL statements relevant to data flow analysis for data lineage. Most statements do not generate any data flow relevant to data lineage and are not listed.

3.3.7.1 Query

SAP Hana queries the data by the standard SELECT statement. Queries are used to retrieve data from the database and are the most critical data flow generators. The SELECT statement also allows many clauses and switches, even SELECT statement nesting, and hence it can generate very complex data flows throughout transformations. Listing 3.7 shows a simplified definition of the SELECT statement. All clauses are documented in [14].

```
<select_statement> ::=
[ <with_clause> ] <subquery> | [ <with_clause> ] ( <subquery> ) |
    { <subquery> | ( <subquery> ) } INTO { <table_ref> | <
    variable_name_list> } [ ( <column_name_list> ) ]

<subquery> ::= <select_clause> <select_into> <from_clause> [ <
    where_clause> ]
 [ <group_by_clause> ]
 [ <having_clause> ]
 [ <set_operator> <subquery> [ {, <set_operator> <subquery> }...
    ] ]
 [ <order_by_clause> <limit_clause> ]

 <select_clause> ::=
 SELECT [ TOP <unsigned_integer> ] [ ALL | DISTINCT ] <
    select_list>
```

Listing 3.7: Simplified SELECT statement definition in the SAP EBNF notation [14]

1. Base query

   The base SELECT statement is comprised of `SELECT <select_items> FROM <table_name> WHERE <expression>`. Select items are columns or expressions referencing sources in the FROM clause.

2. FROM clause

   The SELECT statement defines the sources and outputs of data. Outputs are represented by the <select_list>clause. The FROM clause defines inputs – it can contain multiple tables, table variables, array functions, a SELECT statement in brackets, or table with a JOIN clause.

   JOIN clause is a method to merge two (or more) sources. The clause defines a condition to join the table expression. Then, it creates a virtual table (result set) that contains all the tables.

3. WHERE clause

   An important part of the query is the ability to filter out the data. That is done using the WHERE clause. The clause is comprised of expressions that reference sources from the FROM clause.

4. GROUP BY clause

   Group by clause aggregates the rows in the result set. The clause is designed alongside the Aggregate functions described in 3.3.5.4. When the GROUP BY clause is specified, all columns in the select list must be specified either in the GROUP BY clause or specify the Aggregate function. This ensures that all columns are comprised of unique values or aggregated.

5. HAVING clause

   HAVING clause is the filtering clause for the queries using the GROUP BY clause. HAVING is a filtering clause that is very similar to the WHERE clause with the difference that it filters out aggregated columns.

6. WITH clause

   WITH clause (also abbreviated CTE – Common Table Expression) creates a temporary result set (table). They are usually used to simplify the queries (to prevent code duplication) and to reduce querying the same tables multiple times. The data flows are simple – the CTE is represented as a temporary result set (same as the SELECT) used in the FROM clause.

7. Set operations

   Set operations add multiple SELECT statements together. In SAP Hana, it is possible to unite, intersect, or remove rows (except) from one or more SELECT statements.

   Unite has two modes – UNITE and UNITE ALL. The difference is that UNITE does not keep duplicate rows, while UNITE ALL does.

8. ORDER BY clause

   ORDER BY is a clause that changes the order of the result rows. The sorting can be specified as ascending or descending.

9. LIMIT clause

   The clause removes part of the results as it limits the number of rows. It specifies the number of rows and offset – how many rows are skipped until the limit is applied.

10. SELECT INTO

    SELECT INTO is a different way of writing the select statement but also comes as a clause. The SELECT INTO clause is located after the select items and causes the results to be stored in variables. This requires the SELECT to have only one result row. The statement variant causes the result set (resulting queries) to be inserted into a table, and syntactically, it is placed after the FROM clause.

3.3.7.2 INSERT statement

Insert statement adds a row to a table. The simplified syntax of the statement is 3.8. As defined, the insert statement can insert values from the VALUES clause that contain expressions or values represented by a select statement.

```
<insert_statement> ::= INSERT INTO <table_name>
 [ ( <column_list_clause> ) ]
 { { <value_list_clause> | <select_statement> }
```

Listing 3.8: Simplified INSERT statement definition in the SAP EBNF notation [14]

3.3.7.3 UPDATE statement

For changing the data in tables, the update statement is used. The simplified syntax of this statement is shown in 3.9. The statement itself sets values defined in the set clause in the table. The FROM clause provides additional context for the values in the SET clause. The values changed can be limited by the WHERE clause.

```
<update_statement> ::= UPDATE [ <top_clause> ] <table_name> [ AS
    <alias_name> ]
   [ <from_clause> ]
   <set_clause>
   [ WHERE <condition> ]
<set_clause> ::=
 SET {<column_name> = <expression>
 | ( <select_statement> ) },...
```

Listing 3.9: Simplified UPDATE statement definition in the SAP EBNF notation [14]

3.3.7.4 MERGE INTO statement

When processing data from multiple sources, it might be necessary to choose a different source for some kind of a condition and probably some values range restriction. For those cases, the MERGE INTO statement might ease the complexity. It has a target table in its definition, table reference, and a condition for values in those two tables. Then it defines matched/not matched branches.

Each branch has an action assigned – INSERT, UPDATE, or DELETE. Then, the matched branch is executed when the condition is met at a particular row. In all other rows, the not matched branch is used. If the branch defines INSERT, it inserts a row into the target table when called. UPDATE branch updates the current row, and DELETE removes it.

```
<merge_into_statement> ::= MERGE INTO <target_table>
 USING <table_reference>
 ON <search_condition>
  { <when_matched_clause>
  | <when_not_matched_clause>
  | <when_matched_clause> <when_not_matched_clause>
  }
<when_matched_clause> ::=
 WHEN MATCHED [ AND <search_condition> ] THEN { UPDATE SET <
    set_clause_list> | DELETE}
<when_not_matched_clause> ::=
 WHEN NOT MATCHED [ AND <search_condition> ] THEN INSERT [ ( <
    insert_column_list> ) ] ( <insert_value_element> [ {, <
    insert_value_element> } ... ] )
```

Listing 3.10: Simplified MERGE INTO statement definition in SAP EBNF notation [14]

3.3.7.5 CREATE TABLE statement

The statement creates tables to store data in. It comes with a wide range of options and clauses, most notably the option to store data in rows (`CREATE ROW TABLE`) as opposed to the default COLUMN store of SAP Hana. CREATE TABLE also specifies columns of the table. Columns define the structure of the data stored and have a data type specified – the section 3.3.2 shows possible data types. SAP Hana also allows creating a table from a source table and inserting data after creating the table. This can be done in two ways – using `CREATE TABLE LIKE` clause or using `CREATE TABLE AS` clause. The first clause mentioned above creates a copy of a defined table and the second clause creates a table based on the SELECT statement's result set. Both clauses do not copy data by default, and the keyword `WITH DATA` must be specified to copy the data. The listing 3.11 shows the simplified syntax of this statement.

```
<create_table_statement> ::= CREATE TABLE <table_name>
 { (table_element, ...) | LIKE <table_name> [WITH [ NO ] DATA] |
    [(<column_name>, ...)] AS (<select_statement>)}
```

Listing 3.11: Simplified CREATE TABLE statement definition in the SAP EBNF notation [14]

3.3.7.6 Variable assignment statement

Storing results is an integral part of the programming capabilities of the `SQLScript` programming language. Variable assignments have the form of

a variable reference, followed by the equals sign and the assigned expression.

3.4 Data flow

This section shows the example data flow for the statements listed in 3.3.1. Data flow nodes represent result sets (entities with data) and operation nodes (nodes of statements or entities that only transform the data). Nodes have a hierarchical structure (columns have a parent – a table, for example). Data flow edges (data flows) have two kinds – direct flow and filter flow. Direct flows represent data source scenario – the node directly uses the values. Filter flows only use the data for filtering output.

3.4.1 Query

Query – represented by the SELECT statement, usually creates the most complex data flow graph as it is the primary statement for the data retrieval.

1. Base query

   The base query creates nodes for the expressions and two result set nodes – one for the master select and the child select, with flows from the table columns to the items of the result set, and filter flows to the references in the WHERE clause expression. Examples 3.12 and 3.13 are represented in the data flow diagrams 3.1 and 3.2 respectively.

2. JOIN clause

   JOIN clause creates a result set for the table expression in the FROM clause. The figure 3.3 depicts the data flows of the query 3.14.

3. GROUP BY clause

   GROUP BY adds filter flows for the references used. The figure 3.4 shows the data flow of the query 3.15 that compiles GROUP BY with the HAVING clause.

4. HAVING clause

   HAVING clause adds filter flows for the references used. The figure 3.4 shows the data flow of the query 3.15 that compiles GROUP BY with the HAVING clause.

5. WITH clause

   WITH clause creates a node for the source – a temporary resultset that has direct flows from the select defined. The figure 3.5 depicts the data flows of the query 3.16.

6. Set operations

   Set operations behave the same for the data flows. As the SELECT statement is split between the master select and child select, all the set operations add a child select node to the master select.

7. ORDER BY clause

   ORDER BY adds filter flows for all the columns in the clause.

8. LIMIT clause

   Limit behaves the same way as ORDER BY – adding filter flows.

9. SELECT INTO SELECT INTO clause adds direct flows to the variables from the select item list. SELECT INTO statement adds direct flows to the specified table.

```
1 CREATE TABLE tbl (a int, b int);
2 SELECT a, b FROM tbl;
```

Listing 3.12: Example simple SELECT statement query

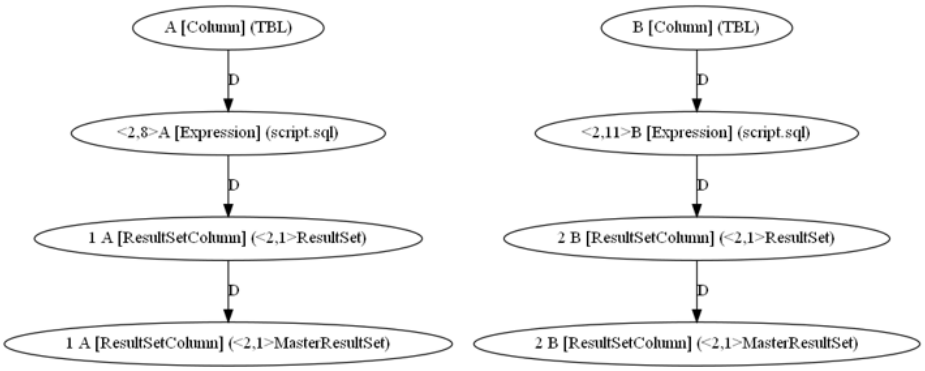

Figure 3.1: Data flow graph of the Example simple SELECT statement query 3.12

```
1 CREATE TABLE tbl (a int);
2 SELECT a FROM tbl WHERE a > 5;
```

Listing 3.13: Example SELECT statement with WHERE clause query

```
1 CREATE TABLE tbl (a int, b int);
2 CREATE TABLE tbl2 (c int, d int);
3 SELECT a, c FROM tbl JOIN tbl2 ON b = d;
```

Listing 3.14: Example SELECT statement with JOIN clause query

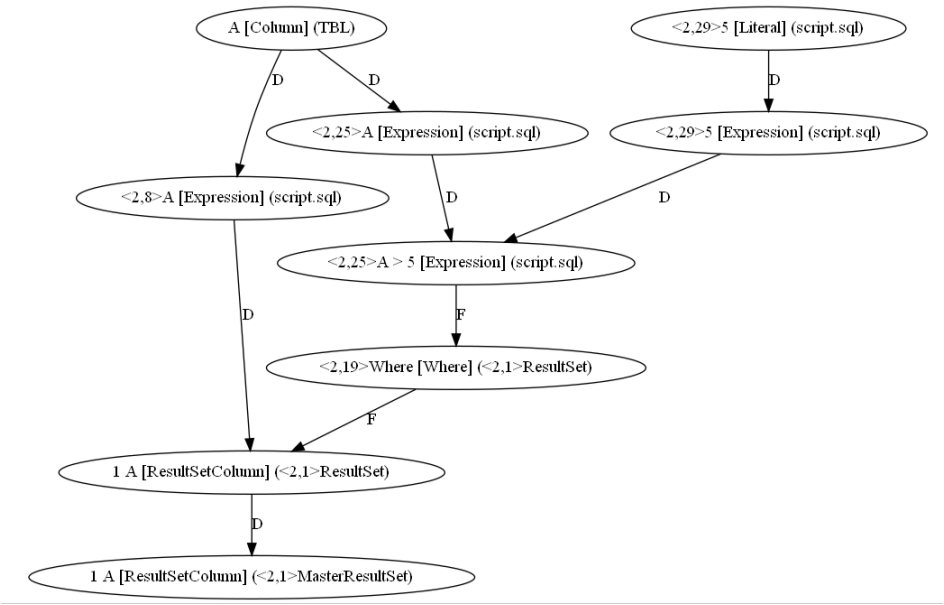

Figure 3.2: Data flow graph of the Example SELECT statement with WHERE clause query 3.13

```
1 CREATE TABLE tbl (a int, b int);
2 SELECT SUM(a), b  FROM tbl GROUP BY b HAVING COUNT(b) > 10;
```

Listing 3.15: Example SELECT statement with the GROUP BY and HAVING clauses query

```
1 CREATE TABLE tbl (a int, b int);
2 WITH q1 AS (SELECT * FROM tbl) SELECT * FROM q1;
```

Listing 3.16: Example SELECT statement with the WITH clause query

3.4.2 Functions

Data flows of functions are complex. A function can be represented by node with child parameter nodes for each input parameter and child output parameter node. The input parameter type determines the direction of the flow. **IN** parameter has a flow to the function node, **OUT** parameter has a flow to the variable in the function call. **INOUT** has both flows. The output/function type has to be determined by the definition of the function. To get a more detailed data flow, it is necessary to process the function body. And connect the parameters (that behave as variables in the function body) and the output node – which can have variables or columns as child nodes.

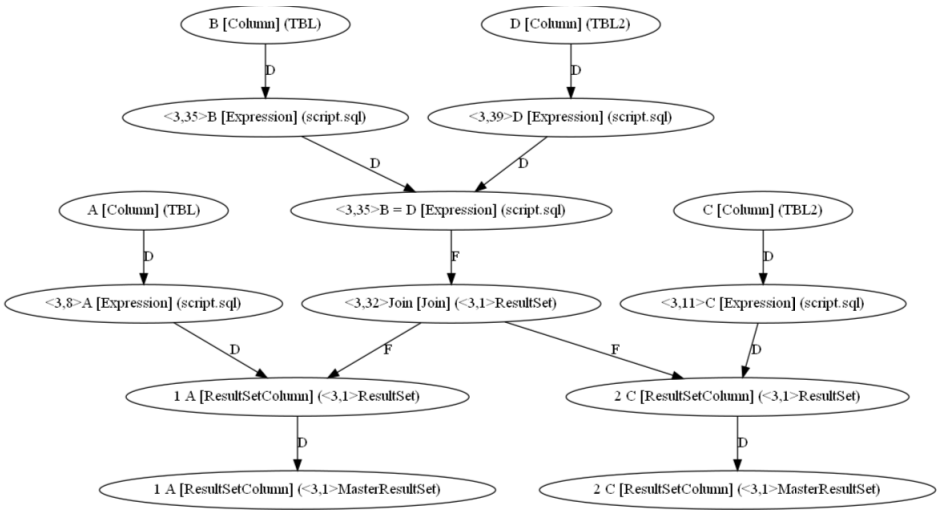

Figure 3.3: Data flow graph of the Example SELECT statement with the JOIN clause query 3.14

3.4.3 INSERT statement

The INSERT statement is very similar to the SELECT INTO statement because it represents the VALUES clause as a SELECT statement. This comes with the advantage of simple data flows. The statement connects the results of the SELECT with the inserted table. The figure 3.6 shows possible data flows of the example 3.17.

```
1 CREATE TABLE tbl (a int, b int);
2 INSERT INTO tbl VALUES (1, 2);
```

Listing 3.17: Example INSERT statement query

3.4.4 UPDATE statement

The UPDATE statement creates direct flows from the SET clause, augmented by the filter flows from the WHERE clause. The figure 3.7 shows possible data flows of the example 3.18.

```
1 CREATE TABLE tbl (a int, b int);
2 UPDATE tbl SET a = 5 WHERE b = 10;
```

Listing 3.18: Example UPDATE statement query

3.4.4.1 MERGE INTO statement

Lineage follows the operations of the statement. The Merge Into statement is represented by a node with children for each branch. The data flow flows into the target table from the action branches. The action branches have filter

flows from the values condition and, possibly, since branches can have another condition, filter flows from them. The figgure 3.8 shows possible data flows of the example 3.19.

```
1 CREATE TABLE tbl1 (a int, b int);
2 CREATE TABLE tbl2 (a int, b int);
3 MERGE INTO tbl1 USING tbl2 ON tbl1.a = tbl2.a
4  WHEN MATCHED THEN UPDATE SET tbl1.b = tbl2.b
5  WHEN NOT MATCHED THEN INSERT VALUES(tbl2.a, tbl2.b);
```

Listing 3.19: Example MERGE INTO statement query

3.4.4.2 CREATE TABLE statement

Creating tables doesn't add any data flows – only the table node. But the CREATE TABLE statement still does add data flows – though `CREATE TABLE AS` and `CREATE TABLE LIKE` clauses with the essential `WITH DATA` option. Figures 3.9 and 3.10 show data flow graphs of the `CREATE TABLE LIKE` 3.20 and `CREATE TABLE AS` 3.21 queries, respectively.

```
1 CREATE TABLE tbl1 (a int, b int);
2 CREATE TABLE tbl2 LIKE tbl1 WITH DATA;
```

Listing 3.20: Example CREATE TABLE LIKE query

```
1 CREATE TABLE tbl1 (a int, b int);
2 CREATE TABLE tbl2 AS (SELECT * FROM tbl1);
```

Listing 3.21: Example CREATE TABLE AS query

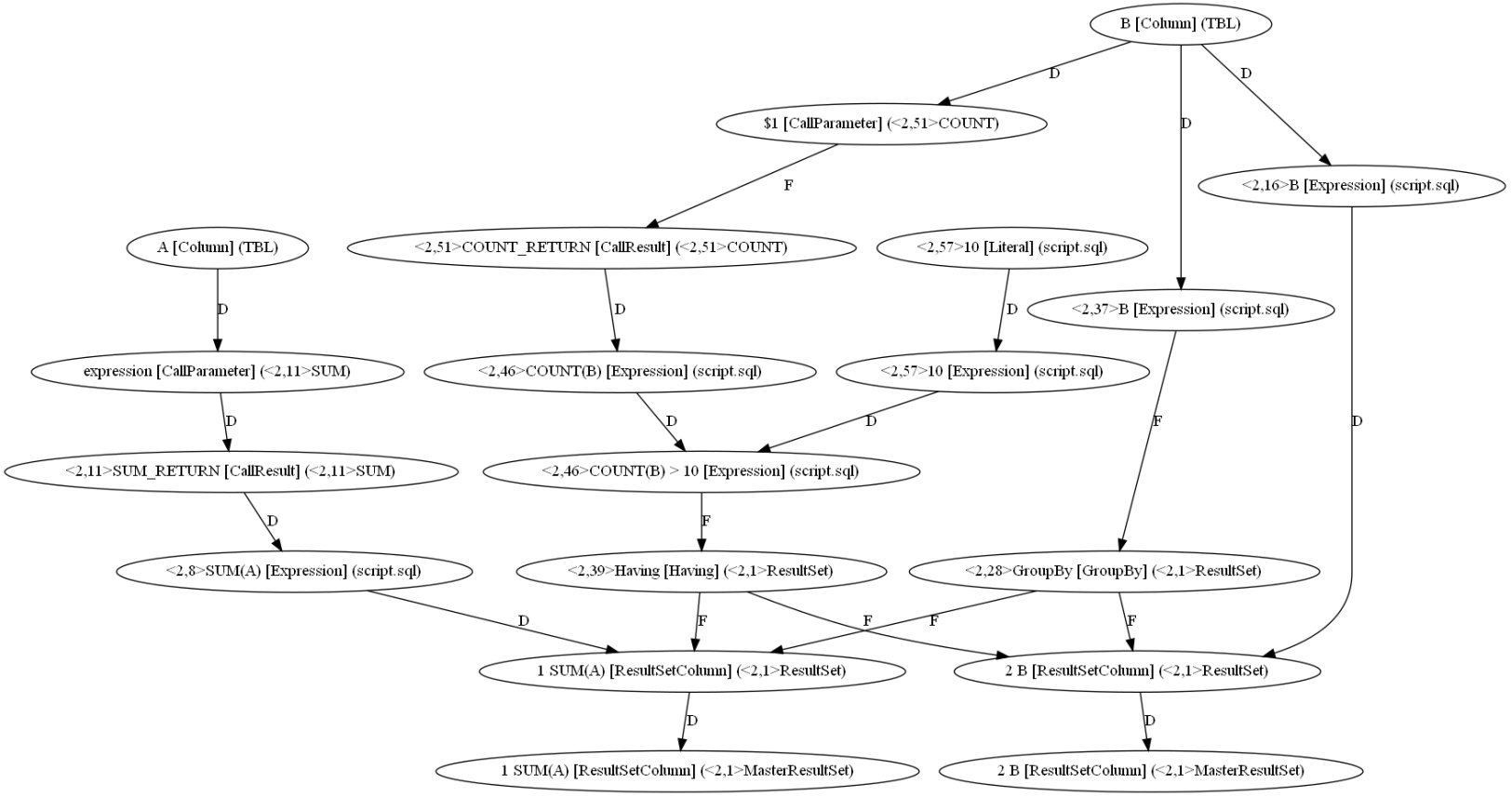

Figure 3.4: Data flow graph of the Example SELECT statement with the GROUP BY and HAVING clauses query 3.15

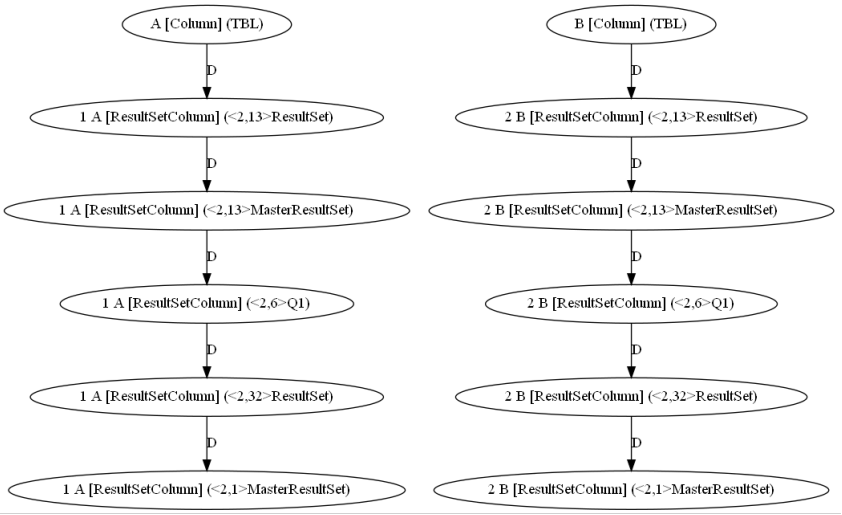

Figure 3.5: Data flow graph of the Example SELECT statement with the WITH clause query 3.16

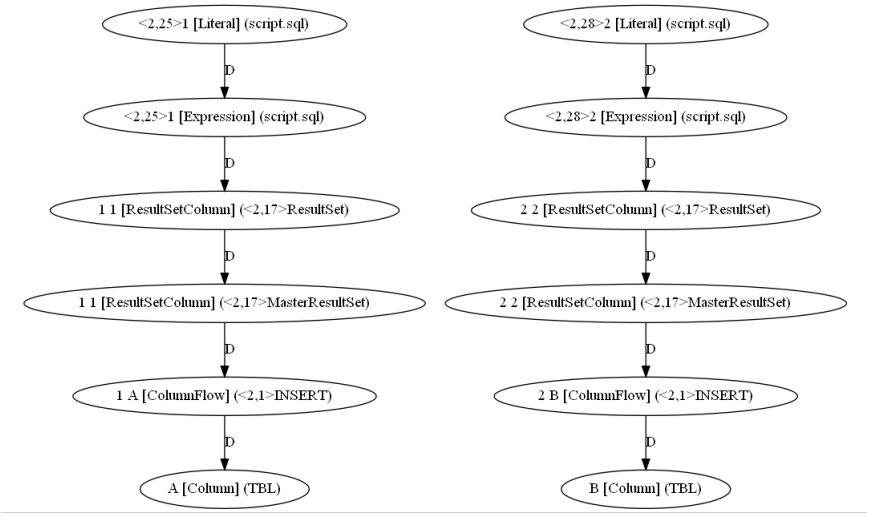

Figure 3.6: Data flow graph of the Example INSERT statement query 3.17

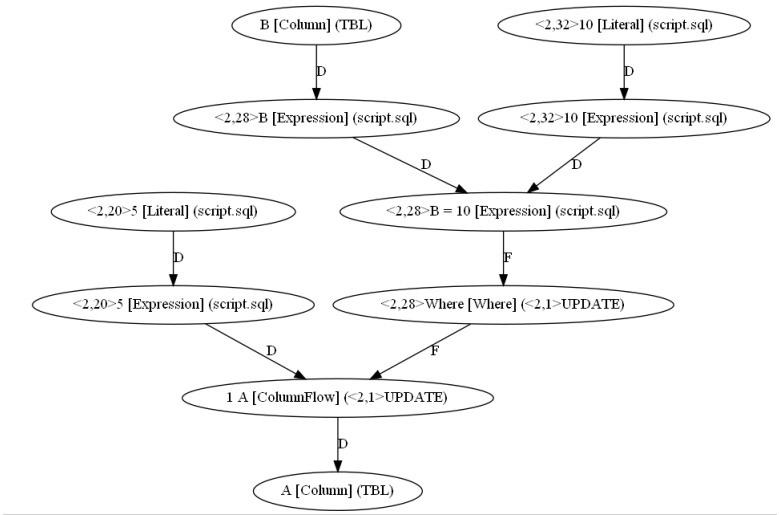

Figure 3.7: Data flow graph of the Example UPDATE statement query 3.18

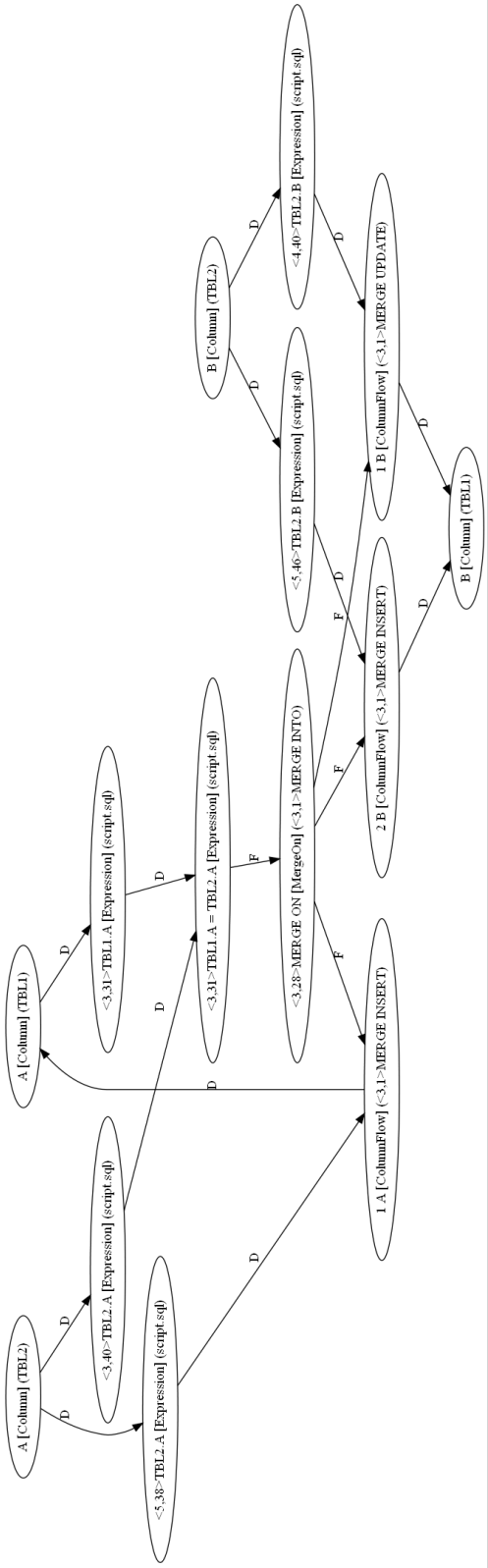

Figure 3.8: Data flow graph of the Example MERGE INTO statement query 3.19

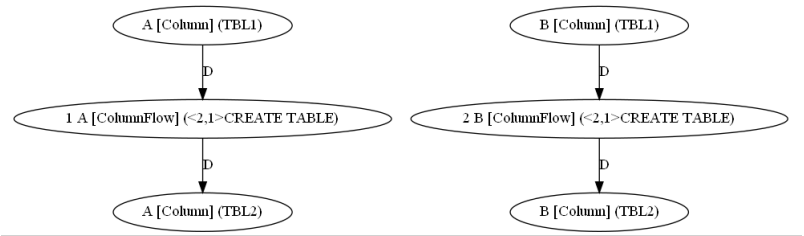

Figure 3.9: Data flow graph of the Example CREATE TABLE LIKE statement query 3.20

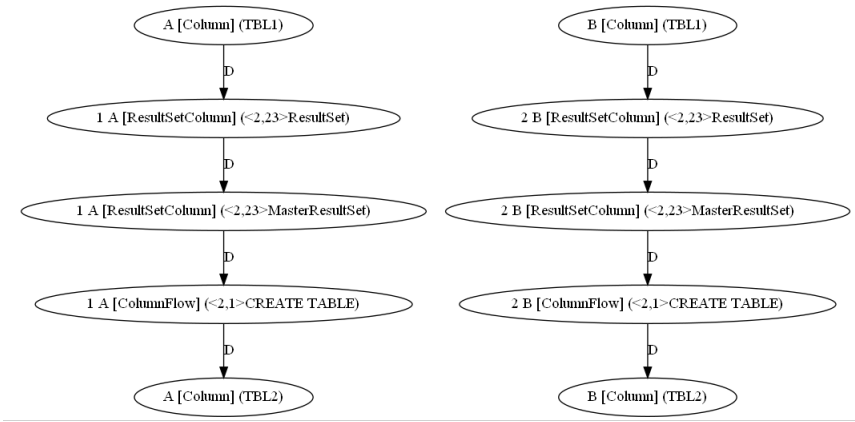

Figure 3.10: Data flow graph of the Example CREATE TABLE AS statement query 3.21

Chapter 4

Design

This chapter introduces the design of the prototype tool structure. It is based on the facts presented in the first chapter and SAP Hana SQL dialect analysis. All of the modules have dependencies on common parts of the Manta Flow tools that it uses as a framework. These modules are out of scope for this thesis.

Mainly, it shows the detailed design of the data flow analysis tool and its modules:

- Data dictionary

  The objects extracted from the database or recognised in the scripts must be represented accordingly and stored. That is the purpose of the Data dictionary module. It contains the object factory and dialect rules. The object factory is responsible for creating and configuring the object representations. Dialect specifies hierarchy rules for the entities created (what entity types can have a particular entity type as a parent) and built-in functions, types, and definitions of objects in the `SYS` schema. The full description can be found in [17].

- Model

  The Model module contains interfaces for nodes and some helper classes used by the Resolver and Data flow generator. This module does not include implementation for the AST nodes.

- Parser and Resolver

  The Parser and Resolver module reads the input files (or strings) and contains the SAP Hana SQL language parser. It also builds the AST with custom nodes that implement interfaces defined in the Model module. While resolving the AST, the data dictionary is being updated too.

- Data flow generator

  After the AST has been built and the data dictionary completed, this module visits the nodes and builds the graph. The graph representation is based on the common data flow generator module and is not part of this thesis.

4.1 Parser and Resolver module

The first step of the tool is to parse the input. The interface for the parser and resolver must meet the functional requirement 3.1.1.2 and so it must implement a method for string processing and file loading. The figure 4.1 shows the methods of the main interface that invokes the processing – `ParserService`.

`ParserService`, implemented by `ParserServiceImpl`, manages the process of transforming the input into the AST. The first step is to tokenise the input using the lexer. As mentioned in 3.2.2, lexer and parser are generated by the ANTLR from the grammar definitions. The rules define the tokens in `SapHanaLexer`. The tokens that are not a part of the reserved words are then used by `SapHanaNonReservedKW` to define identifiers by matching what is an identifier – as explained in 3.3.3.

`SapHanaMain` and `SapHanaExpressions` contain the parser rules. The rules represent some logical unit, for example, a script, a statement, a clause from a script, or a reference to a named object. These rules also use what is called a rewrite rule that defines a custom structure of the AST [9]. The custom structure allows for adding new nodes or even defining classes used for instantiating AST nodes. Custom classes in the AST nodes ensure that custom code for resolving is inserted into the tree. The AST nodes, instantiated from custom classes, apart from resolving references (identifiers of objects), also add helper functions that query the surrounding nodes and eliminate the need to use direct XPath queries on the tree nodes in the Data flow generator module. The functions are simple XPath selectors.

XPath is a querying language that addresses nodes in an XML document. The parser builds a hierarchical structure – AST. Since the AST is very similar to an XML file, XPath can be used to select its nodes. It offers a variety of functionalities to select nodes and eases development.

The main rule for the whole script is to be defined in `SapHanaMain`. Each rule is translated to a function that reads the token stream [9]. After tokenising the input character stream into the token stream, the `ParserService` executes it to parse the input.

4.1.1 References representation

As shown in 3.3.3, identifiers can represent different objects. References are constructed from identifiers to disambiguate what is meant by the last iden-

tifier. The example 4.1 shows schema identifier in the table reference. The identifiers, delimited by a period, are called segments. Segment types depend on the context of the reference. For example, there can be two segments before the table name identifier. The first is always a schema identifier, and the second is a server's name that stores the table.

```
1 test_schema.table
```

Listing 4.1: Example reference

4.2 Data flow generator module

When generating the data flow graph, this module is used. The generation of the graph is based on the visitor pattern to separate the functionality into a different module. The module is primarily implemented in the `FlowVisitor` class. It contains functions called `process` with specific parameters for each AST node that generates data flows. As the `ISapHanaAstNode` requires all nodes to implement the `accept` function, the function can be recursively called by the `FlowVisitor` and process all the nodes.

The processing involves defining graph nodes (vertices in the graph) and flows (edges in the graph). The graph nodes also have a hierarchical structure as they can have child nodes – such as tables having columns. The `SapHanaGraphHelper` class is responsible for creating both nodes and flows. The graph nodes differentiate between an object node (such as a table node) and an operation node (such as a merge statement).

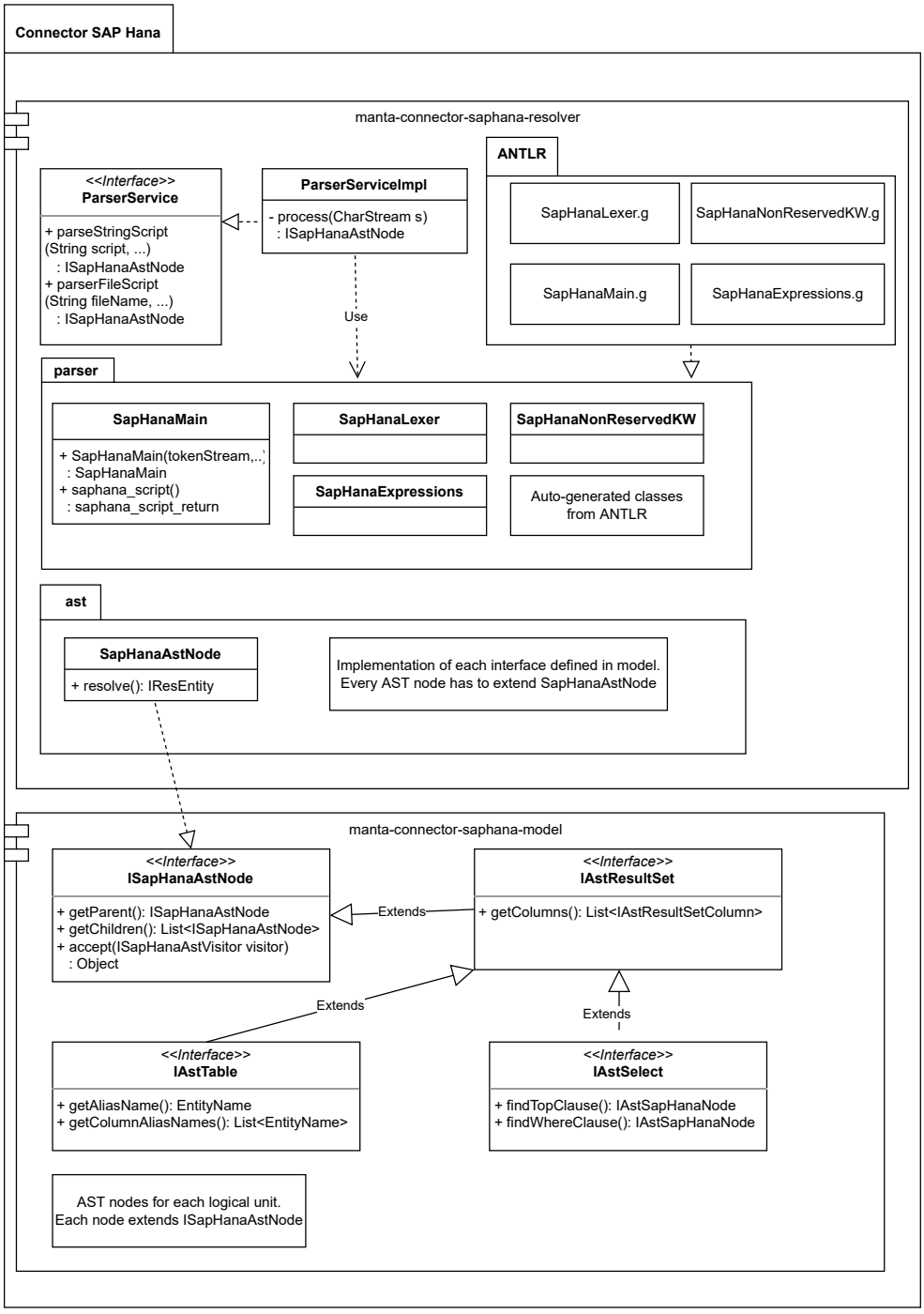

Figure 4.1: Resolver and Model class diagram

Chapter 5

Implementation

This chapter describes the concrete implementation of the modules and issues encountered during the development process.

5.1 Parser

The processing by a parser is the first step of the analysis. It is split into a Lexer part that tokenises the input and the Parser that uses the tokens to define logical units used in the scripts. Both Lexer and Parser are defined as ANTLR grammar rules (.g files) and are translated into Java code as part of the build process. The custom classes for some AST nodes are implemented in the `ast/impl` folder and implement interfaces from the Model module. This section will describe the ANTLR rules.

5.1.1 Lexer rules

The Lexer defines tokens. In the project, they are split into multiple kinds:

- Identifiers

  As discussed in 3.3.3, identifiers have two forms – delimited and undelimited. The lexer rules have to reflect that. The identifiers have been implemented as two rules – `REGULAR_ID` 5.1 and `DELIMITED_ID` 5.2. The delimited identifier consumes every symbol after the opening quotation mark until the ending quotation mark. SAP Hana represents period symbols as part of the name in this case, as opposed to some SQL dialects.

- Reserved words

  Reserved words, as described in 3.3.6, are differentiated from the non-reserved words by name. In Lexer, this has no functional impact. This is because Manta provides the so-called GrammarChecker, which ensures

that all defined keywords, reserved and non-reserved, are used in rules for identifiers in the parser rules.

- Comments and whitespaces

  Comments are also defined as character sequences from the comment start symbol sequence to the ending sequence. There are two types of comments – single-line and multi-line. Single-line comments have double hyphens as the starting sequence and newline as the end. Multi-line comments start with slash and asterisk and end with asterisk and slash (in the reverse order). [14] They are defined as such in the Lexer – 5.3 and 5.4 respectively. Important to note here is the `$channel = HIDDEN;` option. This configuration ensures that the Parser ignores the comments and they do not take part in any other process.

  Whitespaces are defined similarly – 5.5 – the Lexer rule consumes the whitespace symbols and ignores them.

- Special characters/sequences of characters

  The last thing for the lexer is to define sequences used by expressions or characters denoting some action – such as semicolons, or left or right brackets.

```
REGULAR_ID
  : (UNDERSCORE | LETTER | CROSSHATCH) (UNDERSCORE | LETTER |
    DIGIT)*
  ;
```

Listing 5.1: Undelimited Identifier implementation in the Lexer

```
DELIMITED_ID
  : QUOTATION_MARK (~(QUOTATION_MARK) | QUOTATION_MARK
    QUOTATION_MARK)* QUOTATION_MARK
  ;
```

Listing 5.2: Delimited Identifier implementation in the Lexer

```
SINGLE_LINE_COMMENT
  : (
    '--' (~(CR|LF))* (NEWLINE | EOF)
    ) { $channel = HIDDEN; }
  ;
```

Listing 5.3: Single-line comment Lexer rule

```
MULTI_LINE_COMMENT
  : (
    '/*' .* '*/'
    ) { $channel = HIDDEN; }
  ;
```

Listing 5.4: Multi-line comment Lexer rule

```
WHITESPACE
  : ( BLANK )+
    { $channel = HIDDEN; }
  ;
```

Listing 5.5: Whitespaces Lexer rule

5.1.2 Parser rules

The parser processes the token stream per the defined parser rules. The rules also form the AST structure, and so they begin with the `saphana_script` rule as the root of the tree. Statement aggregation is implemented in the `statement_list` rule. `ParserServiceImpl` instantiates `SapHanaMain.java` and calls the main rule function. The main function returns the root node of the AST created for the parsed input. On the root node, it is possible to call the `resolve()` function that recursively resolves all the references in the tree.

Statements are merged into a single rule `single_statement` that is part of the rule `statement_int`. The `statement_int` rule also contains syntactic predicate [3] for variable assignments.

5.1.2.1 Expressions

Expressions are represented in the `SapHanaExpressions.g` parser rules file as a `expression` rule. Expressions are implemented in the following hierarchical order:

1. Boolean operations (OR, AND, NOT)
2. Comparison operators (EQUALS, LESS THAN, GREATER THAN, BETWEEN, LIKE (REGEXP), IS, IN)
3. Concatenation operator
4. Arithmetic operators in their respective mathematical, order (addition, multiplication)
5. Unary operators (PLUS, MINUS)
6. Atom values (literal values, EXISTS predicate, CONTAINS predicate, CASE expression, SELECT as a subquery, reference to an object)

Parenthesis in the expression is represented as a subquery and thus recognised as `select_statement`.

[3] Predicate is an expression used for indicating the validity of continuing the parse. [4]

5.1.2.2 Query

Query is represented in the `SapHanaExpressions.g` parser rules file as a rule `select_statement`. This rule is divided into two logical parts (AST nodes) – `AstMasterSelect`, and `AstSelect`. `AstMasterSelect` is the outer/parent node, and can contain multiple `AstSelects`, representing SELECT statements in the set operations.

As aforementioned in the Expressions description 5.1.2.1, `select_statement` is designed to also act as an expression. This ensures that parenthesis can be used either in specifying the order in arithmetic operation or denoting a subquery. Syntactic predicates are used for this to build the AST correctly.

5.1.2.3 Other statements

The rest of the statements are implemented in `SapHanaMain.g`.

5.2 Resolver

The resolver is implemented as a part of the custom AST nodes. The custom AST nodes, as shown in 4.1, implement a method `resolve()` that handles the resolving – creates and finds representations of database objects in the Data dictionary.

Resolving is implemented as idempotent – when the `resolve()` is called multiple times on the same entity, it does not cause any side-effects (the resolving does not create the entities in the Data dictionary more than once).

5.2.1 Context resolving

This hierarchical resolving is, in some cases, augmented by a function `resolveInternal(Deque<Map<EntityName, IResObject>> context)`. The function is intended for statements that require additional context for references (for example, nested statements). This can be shown in an example listing 5.6 – the context for the inner query contains a reference for `tbl1`. The resolveInternal function in the inner query then finds it while resolving the column/reference `a`.

This is implemented as a record in the Queue (Deque). The map contains names and references to the Data dictionary objects. The Queue has been used to represent possible nesting of the statements – each level may add references with the same name. References searching then looks into the last/most recent map in the context and, if not found, continues on the upper level. The figure 5.6 shows the sequence diagram of resolving the example 5.6. Note that the AST_SELECT calls `resolveInternal(context)` on the second AST_MASTER_SELECT.

```
1 CREATE TABLE tbl1 (a int, b int);
2 CREATE TABLE tbl2 (c int, d int);
3 SELECT * FROM tbl1 WHERE (SELECT c FROM tbl2 WHERE d = a) = 5;
```

Listing 5.6: Example context resolving

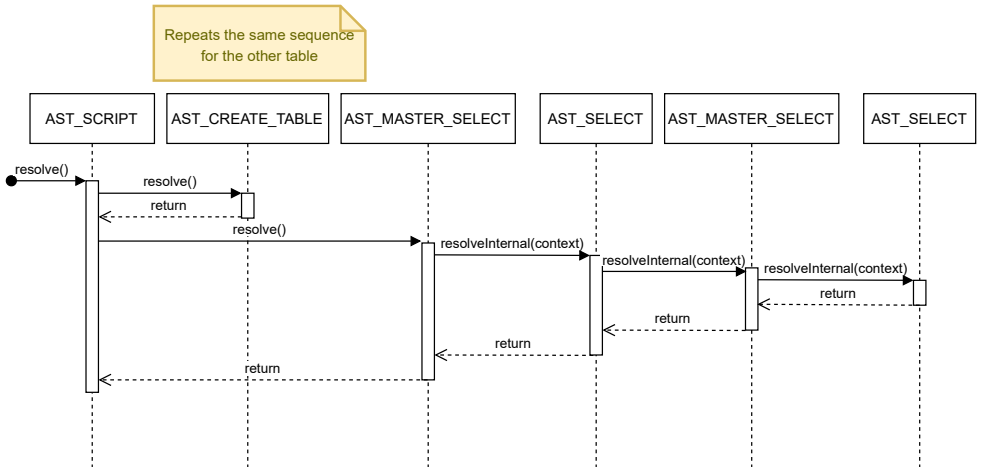

Figure 5.1: Sequence diagram of resolving the example 5.6

5.3 Data flow generation

As proposed in 4.2, the `FlowVisitor` is run by calling the `process(SapHanaAstNode node)` function on an instance with the script (root) node of the AST. This commences the data flow graph building process. The `FlowVisitor` has generic `process(SapHanaAstNode node)` that calls `accept(ISapHanaVisitor flowVisitor)` with an reference to itself as the parameter on all child nodes. This repeats until the `FlowVisitor` encounters an entity that has a custom process function implemented. For example, the CREATE TABLE statement is represented by the `AstCreateTable` class in the Parser and Resolver module. This class implements interface `IAstCreateTable` located in the Model module. Since the Data flow module is designed not to have a dependency on the Parser and Resolver module, the process function is defined for the `IAstCreateTable` interface.

The nodes and edges generation is, as mentioned, handled by the `SapHanaGraphHelper` class. The class extends `AstGraphHelper` which is an internal class of the Manta framework with definitions of the graph and edge representations/-classes. `SapHanaGraphHelper` augments this base class with SAP Hana's specific node and edge generating functions.

Each `process` function then invokes functions from `SapHanaGraphHelper` according to the output of the helper functions defined in the AST node inter-

faces and the dictionary object. In the case of the CREATE TABLE example, the processing creates graph nodes for the columns of the table (defined as children of the dictionary object). It optionally creates flows/edges in the graph when the table has some initialisation clause defined (CREATE TABLE LIKE or CREATE TABLE AS) – defined by the functions `findSourceSelect` and `isSelectLikeWithData`.

Chapter 6

Testing

While implementing the resolving and data flow module, it was essential to see the resulting structures – the AST for the Parser and Resolver module and the data flow graph for the Data flow module. With those AST representations in place, it was possible to automatise the tests to verify that the representations are still the same. The prototype tool uses JUnit for the tests.

All tests depend on the `manta-connector-saphana-testutils` module that implement the base testing class `SapHanaTestBase`. The class inherits from Manta's internal helper testing class and is used as a base for all tests in all testing classes in all SAP Hana modules.

The primary issue with testing the Parser, Resolver, or the Data flow generator is that, due to the complexity of the query language, the tests can't possibly cover all use cases. The testing approach chosen was to make sure the basic structure and use cases work and then try out all the examples found in the documentation. The tests were then automated, with the results saved and compared on every run of the tests. Both Parser and Resolver and the Data flow generator modules tests are placed in the JUnit test folder.

6.1 Parser and Resolver module testing

The module includes the `AstBasicTest` class, which is the parent class for all the tests. The class is responsible for initiating the test scenario. All other test classes extend this class.

The test that is used for debugging purposes – it parses and resolves only the test script – is called `AstResolverTest`. The test only checks for errors in the AST structure and passes if there are none. But the primary purpose is to display the AST structure when the logger is set to debug output level.

The test class`AstResolverBasicTest` contains tests for error recovery. Then there are two tests that use prepared test scripts – `AstInvariantsTest` and `AnnotatedFilesResolverTest`. `AstInvariantsTest` checks whether the AST contains all tokens (whether the input script can be reconstructed from

the AST). This ensures that all rewrite rules do not leave out any tokens. `AnnotatedFilesResolverTest` verifies that the AST does not contain error nodes and deducted entities. Input scripts are located in the folder SimpleTests in the test resources.

6.2 Data flow generator testing

Similar to the Parser and Resolver module, the Data flow generator module defines the base test class `BaseAstFlowTest`. This class defines helper functions for processing data flow graphs. All test classes then extend this class.

The test class `ScriptPathTest` has two purposes – the first purpose is to verify the correct data flow nodes structure – as they have a hierarchical structure (a script has statements, a table has columns). This is done on an example structure of folders – the input script is located in 2 nested folders. The second purpose is to be used for debugging purposes.

The test class `AstFilesFlowTest` automates the repeated runs as it implements a mechanism for comparing the data flow graph structure. Each statement, and sometimes particular statement, is represented by one input `.sql` file and one `_expected.txt` file that contains data flow graph. This method provides a way to automatically verify that the structure has not changed after introducing changes anywhere in the processing pipeline.

Conclusion

This thesis aimed to analyse the query language of the SAP Hana database and the design and implementation of a prototype tool that could be used as a module for the Manta Flow data flow analysis tool.

First, this thesis introduced the theoretical concepts used. The analysis part described the suitable technologies and listed the SQL statements that impact the data flow. After that, it described them in appropriate detail. The last part showed the possible data flows of these statements with examples.

The design chapter proposed the architecture for the Parser and Resolver module, alongside the architecture for the Data flow module. It explained the structure and process of parsing, resolving, and generating data flows.

The last chapter described the process and architecture for testing the prototype tool.

All goals of the thesis were accomplished. The module has been successfully integrated with the Manta Flow tool and is being deployed to the customers at the time of writing this thesis. The next step might involve adding the data flow analysis of the Calculation views that have already been requested as a part of the common way SAP Hana is being used. The Calculation views analysis was not included because they are a different part of the SAP Hana system that is not based on SQL. Calculation views depend on a client execution engine that lets the users define queries in a graphical environment. The representation of such Calculation views is saved in an XML file and would need a different and probably simpler analyser as they resemble the working of ETL tools.

Bibliography

1. GINSBURG, Seymour; SPANIER, Edwin H. Finite-Turn Pushdown Automata. *SIAM Journal on Control*. 1966, vol. 4, no. 3. Available from DOI: 10.1137/0304034.
2. ROSENKRANTZ, D.J.; STEARNS, R.E. Properties of deterministic top-down grammars. *Information and Control*. 1970, vol. 17, no. 3, pp. 226–256. ISSN 0019-9958. Available from DOI: https://doi.org/10.1016/S0019-9958(70)90446-8.
3. BONDY, John Adrian; MURTY, Uppaluri Siva Ramachandra, et al. *Graph theory with applications*. Macmillan London, 1976.
4. PARR, Terence J; QUONG, Russell W. Adding semantic and syntactic predicates to LL (k): pred-LL (k). In: *International Conference on Compiler Construction*. 1994, pp. 263–277.
5. LAWSON, Mark V. *Finite automata*. Chapman and Hall/CRC, 2003.
6. MCCRACKEN, Daniel D.; REILLY, Edwin D. Backus-Naur Form (BNF). In: *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., 2003. ISBN 0470864125.
7. ECONOMOPOULOS, Giorgios Robert. *Generalised LR parsing algorithms*. 2006. PhD thesis. Citeseer.
8. LOURIDAS, P. Static code analysis. *IEEE Software*. 2006, vol. 23, no. 4, pp. 58–61. Available from DOI: 10.1109/MS.2006.114.
9. PARR, Terrance. *ANTLR v3 documentation* [online]. 2009 [visited on 2022-04-15]. Available from: https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687234/ANTLR+v3+documentation.
10. COOPER, Keith D; TORCZON, Linda. *Engineering a compiler*. Elsevier, 2011.

11. MATHEMATICS, Encyclopedia of. *Formal language* [online]. 2011 [visited on 2022-04-30]. Available from: http://encyclopediaofmath.org/index.php?title=Formal_language&oldid=46955.
12. PLATZER, André. Lecture Notes on Lexical Analysis. 2013.
13. KRÁTKÝ, Tomáš. *Different Approaches To Data Lineage* [online]. 2018 [visited on 2022-04-08]. Available from: https://getmanta.com/blog/different-approaches-to-data-lineage/.
14. SAP. *SAP HANA SQL Reference Guide for SAP HANA Platform* [online]. 2019 [visited on 2022-04-04]. Available from: https://help.sap.com/docs/SAP_HANA_PLATFORM/4fe29514fd584807ac9f2a04f6754767/b4b0eec1968f41a099c828a4a6c8ca0f.html.
15. SAP HANA & Data Warehousing for non-experts. *SAP Community blogs* [online]. 2019 [visited on 2022-04-03]. Available from: https://blogs.sap.com/2019/05/15/sap-hana-data-warehousing-for-non-experts/e.
16. WATTANAJANTRA, Asavin. *Business reporting: With great growth comes great complexity* [online]. 2019 [visited on 2022-04-20]. Available from: https://www.sage.com/en-gb/blog/business-reporting-growth-complexity/.
17. HLAVÁČ, Ondřej. *SAP Hana database metadata extraction tool*. 2020. Bachelor's thesis. České vysoké učení technické v Praze.
18. STACKOVERFLOW. *2021 Developer survey* [online]. 2021 [visited on 2022-04-04]. Available from: https://insights.stackoverflow.com/survey/2021.
19. SAP. *What is SAP HANA* [online]. 2022 [visited on 2022-04-03]. Available from: https://www.sap.com/products/hana/what-is-sap-hana.html.
20. SAP. *What is SQLScript?* [Online]. 2022 [visited on 2022-04-03]. Available from: https://help.sap.com/docs/SAP_HANA_PLATFORM/de2486ee947e43e684d39702027f8a94/297af2926307446cbbfb1a8f96fec941.html.
21. KRATKY, Tomas. *The Ultimate Guide to Data Lineage in 2022*. November 2021. MANTA, [n.d.].
22. *Information technology — Syntactic metalanguage — Extended BNF*. Geneva, CH, 1996-12. Standard. International Organization for Standardization.

Appendix A

Acronyms

AST Abstract Syntax Tree

SQL Structured Query Language

CFG Context-free Grammar

CFL Context-free Language

LL Left-to-right, Leftmost

BNF Backus-Naur form

EBNF Extended Backus-Naur form

DFA Deterministic finite automaton

PDA Push down automaton

ETL Extract-Transform-Load

ANTLR ANother Tool for Language Recognition

Appendix B

Contents of enclosed SD card

- `readme.txt` ..... the file with SD card contents description
- `impl` ..... the directory with the implementation
  - `manta-connector-saphana-manta-connector-saphana-model` sources of the Model interfaces
  - `manta-connector-saphana-resolver` ..... sources of the Parser and Resolver module
  - `manta-dataflow-generator-saphana` ..... sources of the Data flow generation module
- `text` ..... the thesis text directory
  - `thesis` ..... the directory of LaTeX source codes of the thesis
  - `DP_Ondrej_Hlavac.pdf` ..... the thesis text in PDF format