University of Magdeburg

Faculty of Computer Science

Bachelor Thesis

Implementing a Relational Algebra Translator into the SQLValidator environment

Author:

Max Hartmann

2022-05-25

Advisors:

Prof. Dr. Gunter Saake

OvGU - Institut für Technische & Betriebliche Informationssysteme (ITI)

Dr.-Ing. David Broneske

DZHW - Abteilung Infrastruktur & Methoden Kommissarische Abteilungsleitung

M.Sc. Victor Obionwu

OvGU - Institut für Technische & Betriebliche informationssysteme (ITI)

Hartmann, Max:
Implementing a Relational Algebra Translator into the SQLValidator environment
Bachelor Thesis, University of Magdeburg, 2022.

Abstract

The database language SQL is one of the most widely used languages when it comes to relational database systems and has become an industry standard around the world. Therefore, mastering the language is an essential skill that universities need to teach their tech students in database courses. In order to understand the query language part, aside from practical SQL, the underlying mathematical foundation, relational algebra, must be taught and applied as well. To teach practical SQL skills in a compact way, the Otto-von-Guericke University Magdeburg provides a web-based tool called *SQLValidator* to their students. Unfortunately, it does not yet have the functionality to offer relational algebra exercises in the same manner, resulting in an incomplete online learning experience for students.

For this reason a relational algebra extension was developed for the *SQLValidator* environment of the database course. This extension allows students to apply and internalize relational algebra concepts in a task-based learning environment that fits seamlessly into the already established learning platform. To facilitate the input of special characters a virtual keyboard was integrated. The translation process is performed by a special parser framework. Furthermore, the syntactic analysis of the input is taken into account, while existing *SQLValidator* components are provided with interfaces for the realization of semantic analysis which is already implemented for SQL related tasks. To be able to test the extension on real problems, two task sheets, which were previously worked on offline in exercise groups, were translated and tested on the system. Not many complications occurred during this test procedure. This evaluation shows that this extension can have the potential to be used in a live environment for students.

Acknowledgement

Hereby I would like to thank my supervisor M.Sc. Victor Obionwu for the good supervision and keeping morale high during my bachelor thesis. Furthermore I would like to thank Dr.-Ing. David Broneske for the constructive criticism and scheduled meetings. Lastly, my thanks go to Professor Dr. Gunter Saake for allowing me to do this work at his chair of databases.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

1.1 Motivation

In modern software engineering, nearly every software interaction requires some sort of database. Any type of data can be stored in tables which later can be queried to find specific tuples in a database. In order to access the data and setup or manipulate a database within the relational data handling idea mentioned above, there is a commonly used database language called Structured Query Language (SQL). Due to the widespread use of SQL, it is a core skill for any aspiring software developer to know its fundamentals. However, to fully understand relational database systems, it is unfortunately not enough to solely know SQL. In order to get a deeper understanding of the topic, it is also necessary to get a grasp of relational algebra in addition to SQL. Relational algebra is a theoretical foundation for relational databases that represent a mathematical form of queries. Since learning this fundamentals can be hard for students, it is highly convenient to create a system that can check the input and provide you some error messages for an improved learning experience. For this reason and for supporting students, its highly useful to provide a web tool to students in which they can learn SQL and relational algebra in an integrated way. This approach also has the advantage that it hides most technical and infrastructure related necessities behind the frontend so students can fully concentrate on learning. To round up the learning experience with this system, it provides functionalities in a structured and task focused manner.

The SQLValidator is an online learning platform for students of the Otto-von-Guericke-University Magdeburg which provides an extensive training ground for solving SQL related tasks. In order to cover the whole teaching content digitally that the “Datenbanken 1” course provides for its students, the SQLValidator is currently missing one feature: relational algebra task handling. These kind of exercises are currently solved by hand on a sheet of paper which was sufficient enough in an analog learning environment like the exercise groups the students had to physically attend. Nowadays, due to practicability and convenience reasons, a digital version would be a modern solution approach. It allows better individual task monitoring for the course supervisors, fit thematically neat into the existing environment and adds an automated control of the solutions.

1.2 Structure and goal

This thesis goes through some basic theories behind the translation process, addresses related works, goes into detail about the concept with its implementation, and evaluates the system on some already existing tasks. The translator will read a relational algebra statement as an input made with the help of a virtual on-screen keyboard and performs three basic steps.
At first, after the input phase is done, the translator performs a basic syntax validation on the entered query. After that, the semantic verification of the current SQLValidator is performed. This will handle the type checking of the input and verify the column references in a SQL manner after the translation took place. Lastly, the query gets evaluated by using the database engine of the validator. So in total the relational algebra syntax will be validated right after the input, following the translation to SQL to get it semantically verified in order to evaluate that query by executing it on the validators database. Thus, the central goal of this thesis is the implementation and evaluation of a relational algebra extension.

2. Background

This chapter gives some brief insights about fundamental techniques and theories. At first some programming related basics with a brief introduction for some theoretical computer science concepts like Backus-Naur-Form or regular expression are addressed. Subsequently, the relational model for database management will be discussed before coming to the core topics this thesis is about: SQL and relational algebra in combination with the SQLValidator.

2.1 JavaScript and jQuery

"*JavaScript is a lightweight, interpreted, or **just-in-time compiled** programming language with **first class functions**"* [MDN22a].
The just-in-time compilation allows the developer to compile the JavaScript code during the execution of a program rather than before it. This behaviour fits seamlessly into any environment in web development, such as the PHP one of the SQL-Validator, and allows calling functionalities without an overhead "compiling time" on page loading. Thereby JavaScript runs on the client side of the web, which affects the behaviour of the web page on the occurrence of an event [MDN22b]. Besides its use case as a scripting language for web pages, many non browser environments like Node.js, Apache CouchDB and Adobe Acrobat also use it.
First-class functions describe a behaviour in which functions are treated as variables. So it is possible that a function can be assigned as a value to a variable, can be passed as an argument to another function or can be returned by another one [MDN22c]. That is the core mechanic behind widely used callback functions in JavaScript. Following Listing 2.1 shows some code examples for these features [MDN22c]. This usage of functions allows to invoke some characteristics or functionalities for custom objects or variables later on in the same code fragments. It enables the addition of features to a later assigned object so to say.

```javascript
//-----------------------Assigning-----------------------
const temp = function() {
  console.log("Assigning");
}
temp(); //invoke it using the variable
//output: Assigning

//-----------------------Passing-----------------------
function addPrefix() {
  return "SQL";
}
function callName(prefix, suffix) {
  console.log(prefix() + suffix);
}
//Pass 'addPrefix' as an argument to 'callName' function
callName(addPrefix, "Validator");
//output: SQLValidator

//-----------------------Returning-----------------------
function process() {
  return function() {
      console.log("ITI");
  }
}
//using a variable
const myFunc = process();
myFunc();

//using double parenthesis
process()();
//output of both: ITI
```

Listing 2.1: Assigning, passing and returning a function in JavaScript

All in all, JavaScript can function as a **procedural** and as an **object oriented** language. The procedural characteristics derive from the structuring in functions and calling those series of code. On the other hand it is object oriented because objects are created programmatically by attaching methods and properties to otherwise empty objects at run time, as opposed to class definitions in languages like C++ or Java. "*Once an object has been constructed it can be used as a blueprint (or prototype) for creating similar objects*" [MDN22b].

"***jQuery** is a fast, small, and feature-rich JavaScript library*" [Fou22]. The simplified access to DOM elements of a HTML script, their manipulation, extended event handling, animation and Ajax across different browsers are its key features. Ajax, for example, handles loading of data from a server without refreshing of the web page and is heavily used in the SQLValidator. The jQuery library also allows the integration of plugins. With its jQuery UI extension, it offers several more functionalities such as user interface interactions, effects, widgets and themes. The used virtual on screen keyboard is such a plugin designed with jQuery UI properties.

2.2 Backus-Naur-Form

The Backus-Naur-Form is a metalinguistic notation that is used to describe computational processes. It was introduced in the “Revised report on the algorithmic language ALGOL 60” [Bac63]. The paper presents three language representations: the *reference language*, which is the committees working and defining language, the *publication language*, used for communication processes and the *hardware representation*, that is a condensed set of the reference language that uses the limited numbers of character that are restricted by any standard input device. Furthermore, the syntax focuses just on the reference representation of the language so that all objects that are defined within it are represented by a given set of symbols.
“The basic concept used for the description of calculating rules is [an] arithmetic expression containing as constituents numbers, variables and functions” [Bac63, p. 6]. These expressions in combination with arithmetic rules are self-contained units of the language. This explicit formulas is also called *assignment statements*. In order to express characteristics like iterative repetitions of computing statements, certain non-arithmetic statements and statement clauses are added. The formation of compound statements with **begin** and **end** is a representation for that. Labels are also allowed for that purpose.

⟨*basic symbol*⟩ ::= ⟨*letter*⟩|⟨*digit*⟩|⟨*logical value*⟩|⟨*delimiter*⟩

⟨*letter*⟩ ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|
U|V|W|X|Y|Z

⟨*digit*⟩ ::= 0|1|2|3|4|5|6|7|8|9

⟨*logical value*⟩ ::= **true** | **false**

⟨*delimiter*⟩ ::= ⟨*operator*⟩|⟨*separator*⟩|⟨*bracket*⟩|
⟨*declarator*⟩|⟨*specificator*⟩

⟨*operator*⟩ ::= ⟨*arithmetic*⟩|⟨*relational*⟩|⟨*logical*⟩|⟨*sequential*⟩

⟨*sequential*⟩ ::= **go to** | **if** | **then** | **else** | **for** | **do**

. . .

Figure 2.1: Excerpt of basic components of a Backus-Naur-Form after [Bac63]

The Figure 2.1 shows an explicit grammar in Backus-Naur-Form. It defines a *basic symbol* within the reference language symbol set as a symbol of the sets of *letters*, *digits*, *logical values* and *delimiters* which are further specified below it. In order to answer the question, if a certain symbol is in the allowed character set, one can resolve the basic rules one after another and check if it occurs in one of them. If that is the case one can categorize the searched symbol as a *basic symbol*. That describes the process of using the assignment rules of a Backus-Naur-Form rudimentarily. All in all it is a very abbreviated way to depict context-free grammar narrowed down to symbol formalism. It is context-free because there is a finite number of steps you can take to resolve the rules separately. The fixed set of sequential operators is also restricting the number of repetitions to a denumberable amount in itself.

2.3 Regular Expressions

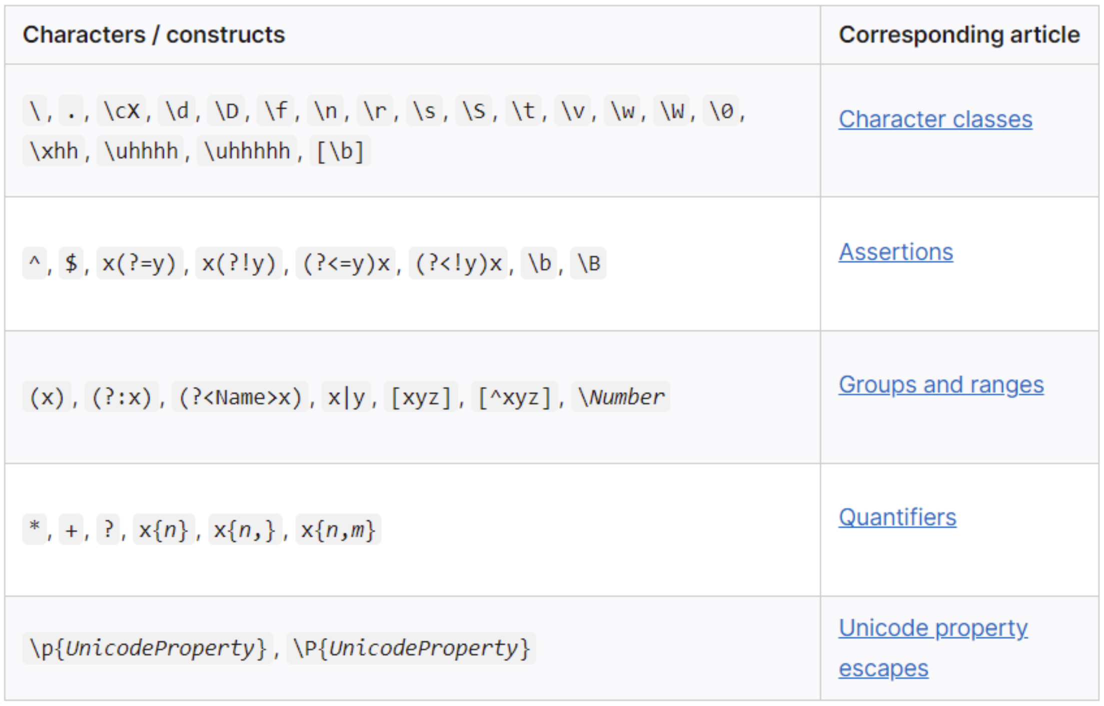
Stephen Cole Kleene described 1951 in his research "Representation of Events in Nerve Nets and Finite Automata" [Kle51] the behaviours of predefined events in nerve nets like definitiveness and "*regular events*" to transform the theoretical concept of finite automata algebraically from it, which is the foundation of theoretical computer science to this day. He described a regular expression over a finite alphabet Σ and following theorems about it to depict the formation of a regular expression:

- $\emptyset$ for denoting the possible empty set of Σ
- ε for denoting the empty string for a set with no content
- *literal character* for denoting a set of a single character in Σ containing only one element

In addition to the core definition of a regular expression, he constructed operations on it to form more complex structures. Given two regular expressions Q and R, following operations produce regular expressions:

- *concatenation* (QR): denotes a set of strings as a result by concatenating an accepted expression of the set Q and one of the set R in this particular order
- *alternation* $(Q|R)$ or $(Q \vee R)$: denotes the union of the sets Q and R combining both element lists
- *Kleene star* $(Q*)$: denotes the smallest superset of Q that contains ε and is closed under string concatenation [Kle51, pp. 49-53]. This results in a set of all strings that can be created by concatenating any finite number (including zero) of strings from Q.

The concept of regular expressions is mainly being used today for matching character combinations in strings in a compact manner. Especially in JavaScript exist certain additional patterns and character only for this purpose (see Figure 2.2). A classic regular expression, constructed after the rules mentioned above, can be extended with character classes, assertions, quantifiers, unicode property escapes, groups and ranges [MDN22d]. Specifically the *character classes* were used, for filtering specific characters in the input string like]; *quantifiers*, like $*$ in normal regular expressions; and *ranges*, to search for multiple characters in the same string using | to distinguish them from one another; the most. JavaScript also allows characters like ρ that are not in the ASCII-character set to form an own character class.



Characters / constructs	Corresponding article
`\`, `.`, `\cX`, `\d`, `\D`, `\f`, `\n`, `\r`, `\s`, `\S`, `\t`, `\v`, `\w`, `\W`, `\0`, `\xhh`, `\uhhhh`, `\uhhhhh`, `[\b]`	Character classes
`^`, `$`, `x(?=y)`, `x(?!y)`, `(?<=y)x`, `(?<!y)x`, `\b`, `\B`	Assertions
`(x)`, `(?:x)`, `(?<Name>x)`, `x|y`, `[xyz]`, `[^xyz]`, `\Number`	Groups and ranges
`*`, `+`, `?`, `x{n}`, `x{n,}`, `x{n,m}`	Quantifiers
`\p{UnicodeProperty}`, `\P{UnicodeProperty}`	Unicode property escapes

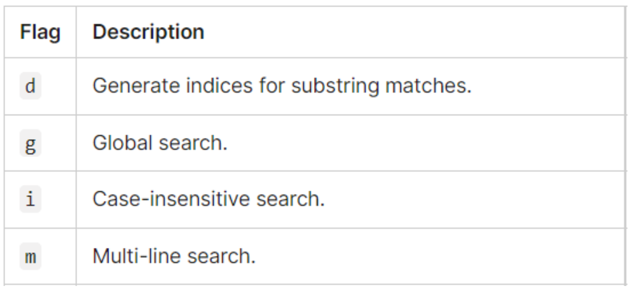
Figure 2.2: Special characters in regular expressions in JavaScript [MDN22d]

To include flags like Figure 2.3 with a regular expression one have to declare them like one of the both options in Listing 2.2 show. The flag will become an integral part of the expression and cannot be added or removed any further later on.

```
const re = /pattern/flags;
const re = new RegExp(’pattern’.’flags’);
```

Listing 2.2: RegExp construction with flags

Flags add extra functionalities like global searching and case-sensitivity to regular expressions. They can be used separately or together in any order. The commonly used *g* flag for example is used to find every occurrences of a certain character set within the whole string instead of only detecting the first appearance. To find every occurrence of a word or phrase disregarding its spelling in the original string, the *i* flag was considered. Such functionalities allow condensed input syntax processing and syntax error handling.



Flag	Description
`d`	Generate indices for substring matches.
`g`	Global search.
`i`	Case-insensitive search.
`m`	Multi-line search.

Figure 2.3: Excerpt of advanced searching with flags in JavaScript [MDN22d]

2.4 Relational model for database management

Edgar Frank Codd, the founder of the relational database concept, presented in his paper "*Relational databse: a practical foundation for productivity*" [Cod07] the practical approach most database systems are still using today. The design goal of this concept was to fulfill his three objectives: [Cod07, Motivation]

1. *data independence*: Provide a sharp and clear boundary between the logical and physical aspects of database management.
2. *communicability*: Keep the overall model simple, so that all kinds of users and programmers could have a common understanding of the data, and could therefore communicate with one another about the database.
3. *set-processing*: To introduce high-level language concepts to enable users to express operations upon large chunks of information at a time and thereby providing the foundation for set-oriented processing.

In the relational model data is addressed by value, rather than by position, which replaces positional addressing by totally associative addressing. The model structures data in tables, which allows unique addressing of every datum by means of the relation name, primary key value, and attribute name. Tables portray thereby the conceptual representation of relations even though they miss out on mapping n-ary relations altogether. This construct enables users as well as programmers *"to leave it to the system to (1) determine the details of placement of a new piece of information that is being inserted into a database and (2) select appropriate access paths when retrieving data"* [Cod07, p. 394]. Furthermore the relational model is a data model consisting of its three integrated parts that define it as such: [Cod07, p. 396]

1. *structural part*: domains, relations (of assorted degrees), attributes (becoming columns of tables), tuples, candidate keys, and primary keys
2. *manipulative part*: algebraic operators (like `select`, `project`, `join` etc.) which transform relations into relations
3. *integrity part*: entity integrity and referential integrity

These sectors are strongly coupled to one another, which let the evidence arise that their behavioural properties are pinned down in such a sharp demarcation that they not allow infinitely many possibilities or endless speculations to occur.

2.5 SQL

The **S**tructured **Q**uery **L**anguage, also known as SQL, is a standard language for relational databases [Saa18]. Its design is conceptually based on natural spoken English combined with programming language formalism to generate formally unambiguous statements. Based on that trait, SQL combines multiple representations in one another to depict the whole use case of relational database management.

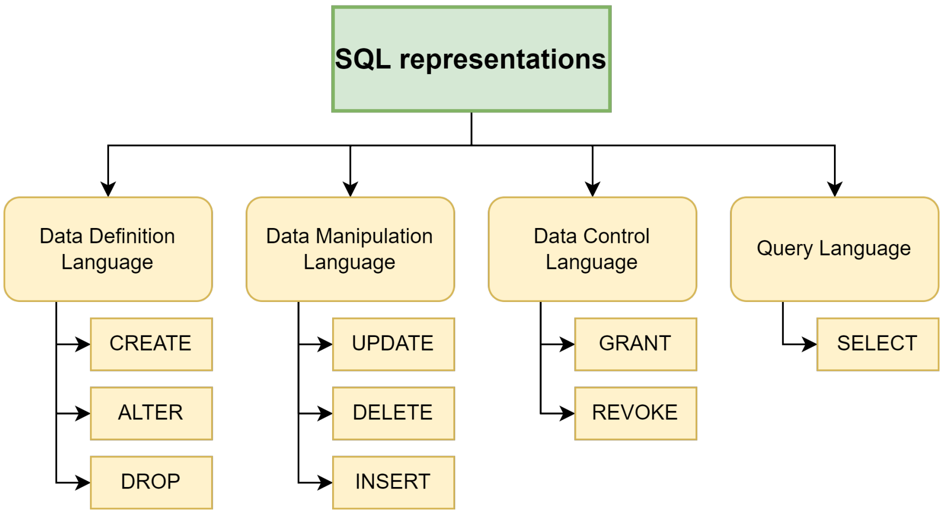


Figure 2.4: SQL language representations with a selection of important commands

In the following paragraphs the different representations and their keywords will be described before referring one key aspect to relational algebra (see Listing 2.3). Before any operations can be made one need to define the table beforehand. For this and other table defining purposes the **D**ata **D**efinition **L**anguage (DDL) representation includes keywords like `create, alter` or `drop` in combination with `table`. The `create` command allows the user to design the table to their desires, set datatypes for their columns, set keys (for highlighting uniquely records and setting up table relations over columns) or define necessary column entries by using `not null`. `Alter` grants editing functionalities to the previous created table. `Drop` deletes columns in combination with `alter` or the whole table including its *data dictionary entry* when used independently.

```
 1  CREATE TABLE Students
 2  (
 3    StudentId int primary key,
 4    FirstName varchar(20) NOT NULL,
 5    LastName varchar(20) NOT NULL,
 6    Faculty varchar(5) NOT NULL,
 7    Address varchar(40) NOT NULL
 8  );
 9  ALTER TABLE Students ADD Degree varchar(2);
10  --adds "Degree" column
11  ALTER TABLE Students DROP Address;
12  --deletes "Address" column
13  DROP Students;
14  --deletes whole table
```

Listing 2.3: Creation, altering and dropping of a specific Students table

The **D**ata **M**anipulation **L**anguage (DML) representation includes keywords like `update`, `delete` and `insert` to change isolated tuples as long as their integrity conditions are not violated (see Listing 2.4). To fill the table with data in form of one or more tuples the `insert` command is applied. `Update` is used to change parts of one or more tuples. In order to get rid off a tuples content in a table entry the `delete` is applicable. However, an `insert` operation must not violate the key conditions, an `update` must not result in a foreign key relationship violation and a `delete` does not entail various integrity violations.

```
INSERT INTO Students
  values('218679','Max','Hartmann','INF');
--note that it is not mandatory to fill out the "Degrees" part
--therefore "Degrees" for this tuple will be NULL
```

Listing 2.4: Inserting, updating and deleting data from the previous Students table example (Listing 2.3) before line 13

The **D**ata **C**ontrol **L**anguage (DCL) representation includes keywords like `grant` or `revoke` to generate a system in which users can get hierarchically authorization assignment of read and write access on the table with its content (see Listing 2.5). In order to allow the user access on certain operations on the database management system, SQL provides the `grant` command. To revert those accesses one can use the command `revoke`.

```
GRANT select, update ON TABLE Students TO ExaminationOffice;
--grants group "ExaminationOffice" the use of select & update
REVOKE update ON TABLE Students TO ExaminationOffice;
--revokes the select privileges from "ExaminationOffice" group
```

Listing 2.5: Granting and revoking rights to use certain operations

Lastly, the most important representation of SQL for relational algebra purposes, the **Q**uery **L**anguage (QL) (see Listing 2.6), will be featured. It describes the main purpose of databases, which is to lookup stored data in a sorted manner. The standard form of a `select` statement (shown in line 1 of Listing 2.6) consists of a column

calling and referencing part to the according table initiated with a `from`, and of a `where` clause followed by different *conditions*.
SQL offers various formalism to nuance the column definition part of a query. The $*$ placeholder replaces the column names by expressing every existing column from a table or condition. In order to filter duplicates of tuples effectively beyond set oriented formalism the keyword `distinct` exists. The *condition* of the `where` command allows to combine a query over multiple tables that stand in a relation to each other. It adds also set-oriented functionalities by using keywords like `natural join, left outer join` or `right join`. There is also an implicit *cross join* according to the mathematical concept of the cartesian product of tables (see line 7 of Listing 2.6). As a result tuples are queried over both tables in which the tuples of the first table get combined with tuples from the second table. If in both tables columns with the same name exist, SQL creates two columns accordingly e.g. `Students.StudentId` and `Courses.StudentId` when both `Students` and `Courses` have a "StudentId" column. In addition, SQL also allows logical operators like `and, or, not, any, all` and many more. For relational algebra purposes the focus on `and, or` and `not` is sufficient.

```
1 SELECT column1 , ..., columnN FROM table_name WHERE condition;
2 --standard form for a complete SQL statement
3
4 SELECT * FROM Students where StudentId > 2000;
5 --outputs every data about students with a StudentId > 2000
6
7 SELECT * FROM Students , Courses;
8 --outputs cross join of both tables with every column
9 --assuming "Students" & "Courses" have the same column "StudentId"
```

Listing 2.6: Some SELECT queries for reference

2.6 Relational Algebra

The so called "relational algebra" includes basic query-related operations for tables, which allow the calculation of new result tables from stored database tables in order to extract specific data from a database. It is the concept on which the QL representation with its query statements is build upon. In mathematics, an algebra is defined by a range of values and operations defined on them. In a database context, the tables of database queries correspond to the values meanwhile the operations are functions that are calculating the query results [Saa18, chapter "Grundoperationen: Die Relationenalgebra"]. Since all operations have tables as input and calculate a new table as a result, these operations can be combined arbitrarily (and allow complex queries by nesting them). The core concept of tables are relations, which are sets of tuples with named attributes. Some operations are therefore obvious: *classic set operations* on set of tuples, such as forming the union or intersection set, and *renaming attributes*. In order to map every query arbitrarily one needs three more Operations: *the projection, the selection* and *the join* (see Figure 2.5).

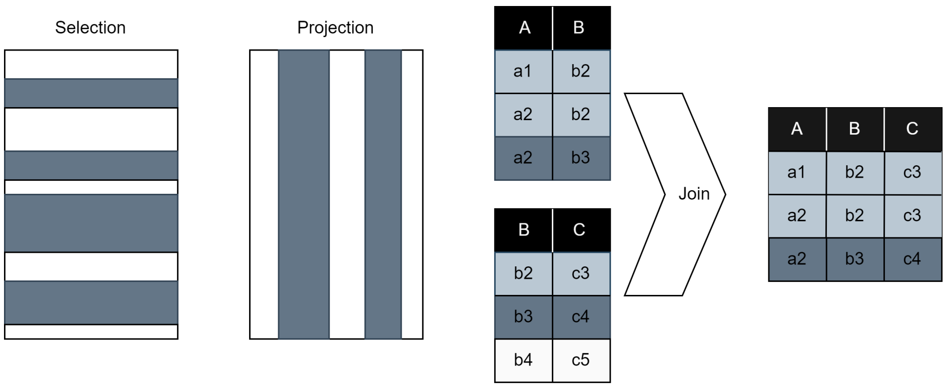


Figure 2.5: Selection, projection and join schematic depicted oriented to [Saa18]

2.6.1 Operations and their translation to SQL syntax

Firstly, the *selection* operation is considered. The operator symbol is σ. This operation is used to select rows of tables on the basis of a selection predicate. The selection predicate is shown lowered to the right of the σ.

$$\sigma_{StudentId>2000}(\texttt{Students})$$

Usually simple conditions occur in the form of selection predicates. They compare attribute values with constants or attribute values with one another. Selection predicates are in SQL syntactically represented by the condition after the `where` keyword. The simple structure of the selection predicates is therefore translated one-by-one.

The *projection* is used to select columns by specifying an attribute list. As an operator symbol π is used. The attribute is again shown lowered to the right of the operation symbol with attribute names separated by commas.

$$\pi_{LastName}(\texttt{Students})$$

Since relations are sets of tuples, the projection automatically removes duplicates. Therefore the `distinct` keyword is used after the `select` in the translation process later on to match this characteristic syntactically.

The *join* operation links tables via columns with the same name, by merging two tuples together if they have the same values. As an operator symbol the $\bowtie$ is used. This operation is also called the *natural join* because it appears to be very intuitive to join same-named columns because they usually have the same meaning.

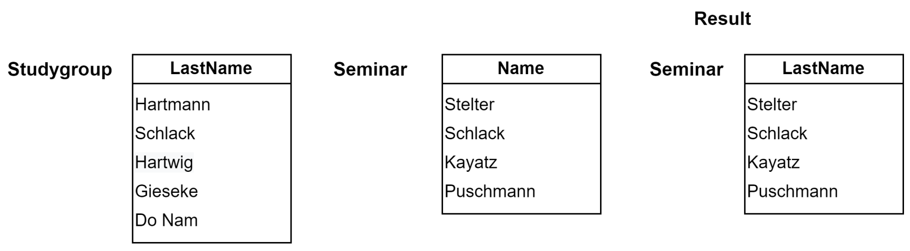
$$\texttt{Students} \bowtie \texttt{Courses}$$

To join both tables the column “Student.Id” is being used. If one of the columns is a key attribute of its table then every resulting tuple is unique. “Dangling tuples”, which are tuples that do not have a corresponding partner, disappear from the resulting table. The above mentioned operations can be combined to formulate more complex queries like:

$$\pi_{FirstName,LastName}(\sigma_{StudentId>2000}(\texttt{Students}) \bowtie \sigma_{Chair='DBSE'}(\texttt{Courses}))$$

To transform the ⋈ operator syntactically correct to SQL one has to insert a `natural join` between both operands or generally speaking, relations.

Sometimes the customization of attribute names comes in handy. Therefore the *rename* operation exists with β as its symbol. This operation is useful when both respected attributes in the relations depict the same meaning but are named differently. This could help in the example below to change the column name “Name” from `Seminar` to the new name “LastName”, so that further operating would be easier with `Studygroup`.



Studygroup

LastName
Hartmann
Schlack
Hartwig
Gieseke
Do Nam

Seminar

Name
Stelter
Schlack
Kayatz
Puschmann

Result

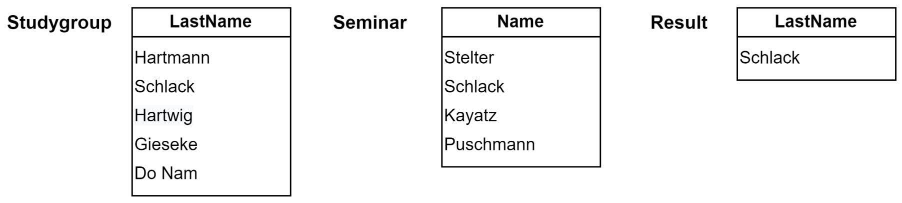
Seminar

LastName
Stelter
Schlack
Kayatz
Puschmann

$$\beta_{LastName \leftarrow Name}(\texttt{Seminar})$$

When translating this operation to SQL it is very natural to use the `as` keyword to change the respective name of the column name entered beforehand.

Set operations like union, intersection and difference are frequently used across relational algebra expressions.
The *union* $r_1 \cup r_2$ of two relation r_1 and r_2 results in the entirety of both tuple sets. It can only occur if the attribute sets of both relations are the same.



Studygroup

LastName
Hartmann
Schlack
Hartwig
Gieseke
Do Nam

Seminar

Name
Stelter
Schlack
Kayatz
Puschmann

Result

LastName
Schlack

$$\texttt{Studygroup} \cup \beta_{LastName \leftarrow Name}(\texttt{Seminar})$$

In order to translate this to SQL the *union* keyword unionizes both operands together.

The *intersection* $r_1 \cap r_2$ as the second set operation results in tuples that are included in both relations at the same time.

Studygroup

LastName
Hartmann
Schlack
Hartwig
Gieseke
Do Nam

Seminar

Name
Stelter
Schlack
Kayatz
Puschmann

Result

LastName
Hartmann
Hartwig
Gieseke
Do Nam
Stelter
Kayatz
Puschmann

$$\texttt{Studygroup} \cap \beta_{LastName \leftarrow Name}(\texttt{Seminar})$$

It is translated to `intersect` and is used in the same way as `union` syntactically in SQL.

Lastly the third set operation *difference* $r_1 - r_2$ eliminates tuples from the first relation, that also occur in the second one.

Studygroup

LastName
Hartmann
Schlack
Hartwig
Gieseke
Do Nam

Seminar

Name
Stelter
Schlack
Kayatz
Puschmann

Result

LastName
Hartmann
Hartwig
Gieseke
Do Nam

$$\texttt{Studygroup} - \beta_{LastName \leftarrow Name}(\texttt{Seminar})$$

Its SQL translation is used in the same way as the other set operations using the `except` keyword.

2.7 SQL Validator and its architecture

The SQLValidator from [OBH+21] is a webbased interactive tool to teach SQL and lets students practice on a live system. Students are able to solve SQL related tasks by sending their suggestion for a possible solution to a database. Then they will get instant feedback regarding errors in their query they need to fix. The SQLValidator uses a client-server architecture where user roles such as students, exercise instructors and administrators interact with the database system using a web interface via a PHP server. In addition to that core purpose of this environment, it also serves as a tool for teaching evaluation of according courses at the Otto-von-Guericke University Magdeburg via polls.

To evaluate the input from the students the SQLValidator implements a validation module that generates a relational table according to the natural representation of a database query and compares it to the solution disregarding to the different output every language feature generates. This procedure allows to evaluate all respective language features from the *DDL* (Data Definition Language), the *DML* (Data Manipulation Language) and the *QL* (Query language) which may be required by the question. Therefore the students can use every mentioned language feature hence the system works independently from them, assuming that students do not cheat by manually selecting the expected solution instead of the desired language construct.

There are four feedback levels the students input can get classified by after submitting an answer to the system: positive/negative feedback which is indicated by color codes (green, yellow, red), a hint, a reference solution and the students (partially) correct solution. Occurring errors are divided into four different error classes. First the SQL query is checked for syntax errors before the table name gets checked with a subsequent examination of column counts, order and name. After that different parts of the database associated components get checked [OBH+21, see Figure 2.6]. The **Foreign Keys** (with their name, reference label and reference column),the **table** (with respecting row count, their order and table content), the **schema** (with its column count CT, data type, isNull, isDefault and column Name errors) and **constraints** (with its constraint count, primary key, foreign key, general key and unique errors) represent the error classes thereby.

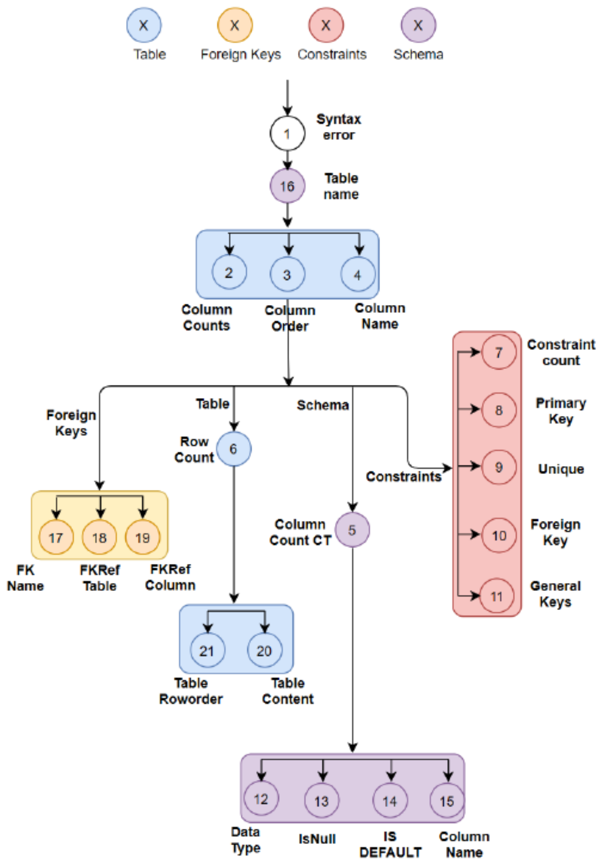


Figure 2.6: Error classes according to [OBH+21, p. 4]

3. Related Work

This chapter, turns the attention to related work. Here, the focus is on the one hand on various tools that implement relational algebra translators or interpreters, and on the other hand on studies that address the implementation of such tools.

3.1 Tools

To start this chapter off, three tools that pursue the same goal as this work will be addressed. In addition, the evolution of the "*RA: A Relational Algebra Interpreter*" approach to the *RADB* tool is taken into account.

3.1.1 Relational Algebra Interpreter

The *Relational Algebra Interpreter* [Raj22] from *Naveen Rajshekhar* is an open source project written in Java. Similar to the approach this thesis introduces, the relational algebra handling is done by a context-free grammar.

The translation is performed in three steps. First, the *lexical analysis* is executed together with the *parsing step*. In lexical analysis, rules are set up to describe the linguistic transfer meanwhile parsing the expression constructs a tree structure that stores the parsed relational algebra statement.
Phase two of this tool is the semantic check. Some steps are performed depending on the operation. For *atomically occurring tables*, it is checked whether they exist in the database or not. With *set operations*, on the other hand, the number of columns and data types of the tables are checked for equality. The attributes of the *projection* are verified whether they occur in the schema of the subquery. With the *rename*, it is examined again whether the attribute list has the same size as that of the schema of the subquery. Finally, the *select* checks if the datatypes of the left and right operands are the same and if the used table exists in the database. Furthermore all attribute references in the select condition must be present in the schema of the subquery.
The third and last phase represents the query translation with the respective execution on the database. For this purpose, a connection to the MySQL database is established and the generated SQL query is sent.

3.1.2 RelaX - relational algebra calculator

The *RelaX - relational algebra calculator* [Kes22] from *Johannes Kessler*, of the *University Innsbruck*, is a web based tool to test relational algebra statements on fixed or self added databases. It comes with a feature-rich intuitive interface whose user experience is further elevated by keyboard shortcuts. Tables and column names can be displayed in the left margin. After selecting them from the drop down menu via a mouse click, respective names get inserted conveniently to the cursor position. The output not only shows the relational algebra expression, but also prints a matching tree structure with associated table view. Additionally, one can download the entered request or view the history of the entered statements. All in all, it is a performant and well-rounded system to practice relational algebra online.

3.1.3 RA: A Relational Algebra Interpreter

RA: A Relation Algebra Interpreter [Yan22] is a simple relational algebra interpreter from *Jun Yang* written in Java, released 2014. It is built on top of a SQL-based relational database system. The system implements relational algebra queries by translating them into SQL queries and executing them on the underlying database system through JDBC. For this purpose it supports DB2, MySQL, PostgreSQL and SQLite database schemes. This system has some known limitations, which led to developing a new revised tool in Python called RA (radb).

Limitations were for example in the error message output, which had little to no meaning regarding the query, or in connection with some database schemes where the interpreter could not produce a result whenever identically named attributes occurred. Furthermore, *RA* may signal an error whenever it has trouble determining how to name the attributes of a result relation. This error behaviour is database type dependent.

3.1.4 RA (radb)

The new *RA (radb)* tool [Yan17] from *Jun Yang* is a simple relational algebra interpreter written in Python 3. This system is also built on a SQL-based relational database system. The main difference between this new tool to the RA: A Relational Algebra Interpreter is that the relational algebra query is again translated into SQL and checked on the database system, but this time with the help of the *SQLAlchemy* toolkit. It also relies solely on a SQLite system this time, which again allows database connections to custom databases.

By restricting to one database system type, some previously named limitations are bypassed. Thus identically named attributes are manageable and error signals are more controlled. Also, error messages themselves are more optimized and specific. Consequently, the biggest sources of errors of the previous version are actually eliminated or avoided.

3.2 Papers

The second part of this chapter is devoted to brief insights at a selection of scientific papers.

3.2.1 Implementation of Relational Algebra Interpreter using another query language

Ratnesh Litoriya and *Anshu Ranjan* designed in their paper “Implementation of Relational Algebra Interpreter using another query language” [LR10] from 2010 a system, written in Java, that compiles relational algebra to SQL to execute it on a relational database system. This tool gets a relational algebra statement as an input, performs lexical and syntactic parsing where in case of an error the user receives a notification.

The design process of the interpreter was split up into five consecutive phases. At the beginning, the already mentioned “*lexical analysis*” is carried out. To do this, the character stream that makes up the source program is read from left to right and is grouped into tokens, i.e. strings that have a common meaning. The second phase, the “*syntax analysis*”, involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to create the output. Afterwards, the “*code generation*” phase is started in which the target code is generated and can be executed directly on the machine. This code “*execution*” represents the next phase. Only statements that are lexically and syntactically correct get to be converted into the target language, in this case SQL. Finally, “*error handling* is performed, which detects and reports lexical or syntactic errors that have occurred.

With the help of JLex and JCup, the authors performed the lexical and syntactic analysis of the input query. JLex scans the particular relational algebra input, divides it into various lexemes, and checks whether the query matches the concerned regular expressions. JCup, on the other hand, analyses the input query but checks whether the query follows the context free language. After both of these routines specific errors get reported. If no errors are detected, the relational algebra gets translated to SQL which is then executed on the database as previously mentioned.

Overall, this work can be considered similar in terms of methodology and goal. However, the semantic checks with JCup are more thorough and the general control flow is more stringent regarding each check before the conversion is performed.

3.2.2 RAPT: Relational Algebra Parsing Tools

Olessia Karpova and her workgroup of the *Department of Computer Science of the University of Toronto*, designed **RAPT**: ***R**elational **A**lgebra **P**arsing **T**ool* and described their results in the associated paper[KDHP15] in 2015.

RAPT uses a syntactic and semantic understanding of relational algebra to transform input statements into a variety of outputs, including LATEX formatted queries, parse tree diagrams, and executable SQL statements. It provides two grammars for the parsing process:

- ▷ a *core* grammar with the five primitive operators $\sigma, \pi, \times, \cup$ and $-$,

▷ an *extended* grammar that adds $\bowtie, \theta$ and $\cap$

Additionally, RAPT supports the assignment operator, renaming of attributes and relations. To facilitate semantic analysis along the parsing process, the tool relies on a collection of decorated abstract syntax trees. This analysis is necessary for resolving names of relations and attributes. In order to identify and report common errors RAPT uses semantic information from a nodes children, its operator and a user-provided database schema.

The system was used for the exercise of a third year database class of over 350 students. It was primarily utilized as a debugging aid for relational algebra tasks students had to work on, which could visualize made mistakes, but also fully automated the grading process of the associated assignments. This paper is relevant because it uses a different approach to implement a similar system and has already tested it in a live environment.

3.2.3 Relational Algebra Interpreter

In their paper "Relational Algebra Interpreter" [AdV14], *Tamim Alkhalifah* and *Denise de Vries* presented a system that implements five different relational operators for relational algebra queries, translates these statements into SQL, and executes them on a database. The web application is written in Visual C#.NET while the technology to identify the terminals, non-terminals and the grammar is based on Irony.

This paper follows the same five phases of interpreter definition like Section 3.2.1. Once again a parse tree of some sort is used to split up the relational algebra expression into terminal and non-terminal symbols. Interior nodes represent the non-terminals, the children of each node represent the replacement of the associated non-terminal in one step of the derivation, meanwhile the leaf nodes themselves are labelled by terminals. In the underlying grammar, terminals are described as atomic elements with which the sentence of a language may be constructed. The five operators $\pi, \sigma, \cup, \cap$ and $-$ are therefore defined as terminals. Non-terminals are elements which are only used in the derivation of a sentence. They get to be replaced by more specific components, either terminals or other non-terminals. With these constructs in mind the grammar was gradually build.

Overall, this design approach is not a bad idea, but is inadequate due to some inconveniences. It lacks operations like *product* or *join*, only general errors are reported that do not fit the specific cause whatsoever and it just supports nesting of operations with one level of nesting. Those deficiencies reduce the general benefit immensely.

4. Concept

This chapter will go into detail about the concept of the relational algebra translation process to SQL. First, an overview over the requirements with the general workflow of a student solving SQL tasks, the adaptation of the translation system to the current SQLValidator state and formalism adaptation is given. Afterwards, the architecture of this extension is featured, the workflow of the relational algebra translation process is established before pointing out how the three requirements regarding error handling and query evaluation are conceptualized.

4.1 Requirements

In the following existing structures are featured to extrapolate the behaviour of the relational algebra extension. The idea is to provide interfaces that can be implemented to the already existing workflow and are easily adaptable.

4.1.1 Workflow of SQL tasks

In order to understand how the translation extension should be implemented similar to the SQLValidator environment, one should have a look at the current system beforehand. From a technical perspective the basic workflow of a student doing his tasks is simplified in Figure 4.1 in form of an activity diagram. It is divided into the two swim lanes *Frontend* and *Backend* to depict mechanisms that are happening while a student works on his SQL task. At the beginning a student enters his solution approach to the specific task he previously selected. After confirming the input, the backend behaviour gets invoked to check the solution the student provided. There are only two outcomes for that process: the suggested solution is *correct*, then the student gets a notification, that the solution was right and the task is solved, **or** the suggested solution is *wrong* which has a detected error with a matching class called in the backend. This will then send a notification with the remarks about the input to the student. With the previously suggested solution and the new information about the errors in it, the student works on a new approach to the tasks problem. This cycle repeats itself until the student solved the task successfully. There is also a real life case in which the student does not contribute a proper solution to the task within a certain time span at all which then result in not getting a voting point for this particular task. This case is not depicted in the activity diagram whatsoever. The mentioned backend activities are highly narrowed down. Normally there are a lot more of database, server and program interactions working together to result in the depicted activities.

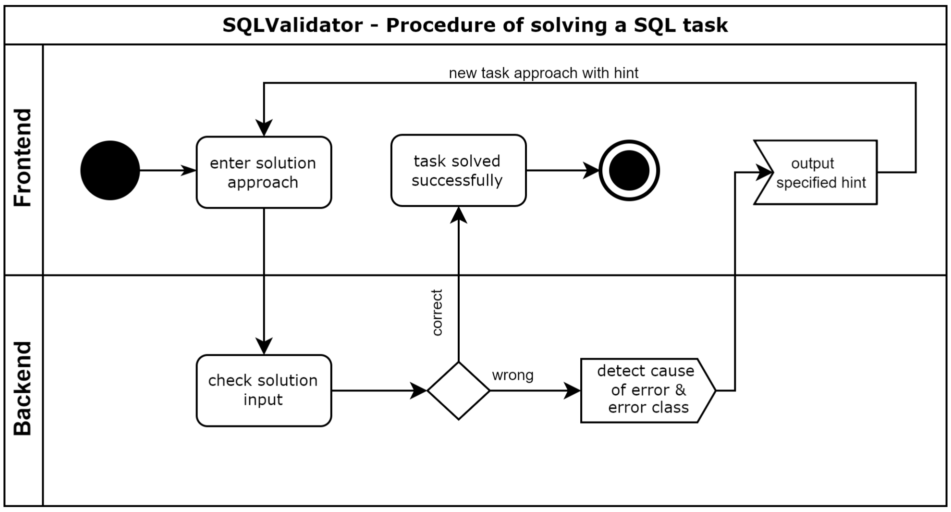
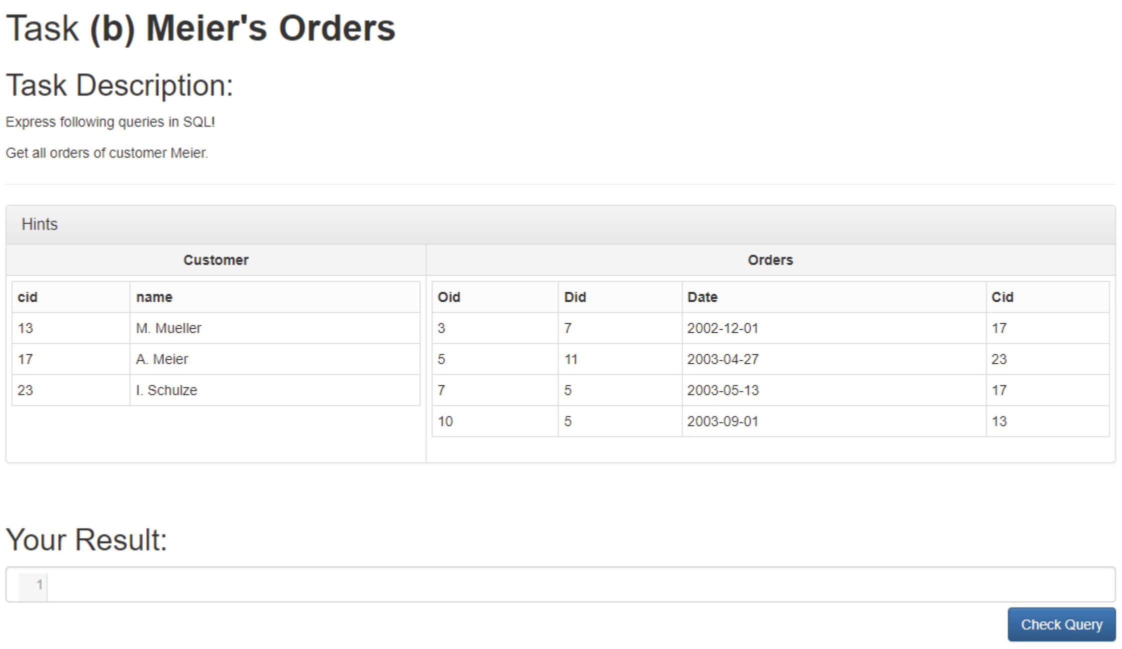


Figure 4.1: Activity diagram about the current SQL task workflow

4.1.2 Adaption to SQLValidator environment

The Figure 4.2 shows the user interface (UI) a student has to interact with in order to do his SQL tasks.



Task **(b) Meier's Orders**

Task Description:

Express following queries in SQL!

Get all orders of customer Meier.

Hints

Customer

cid	name
13	M. Mueller
17	A. Meier
23	I. Schulze

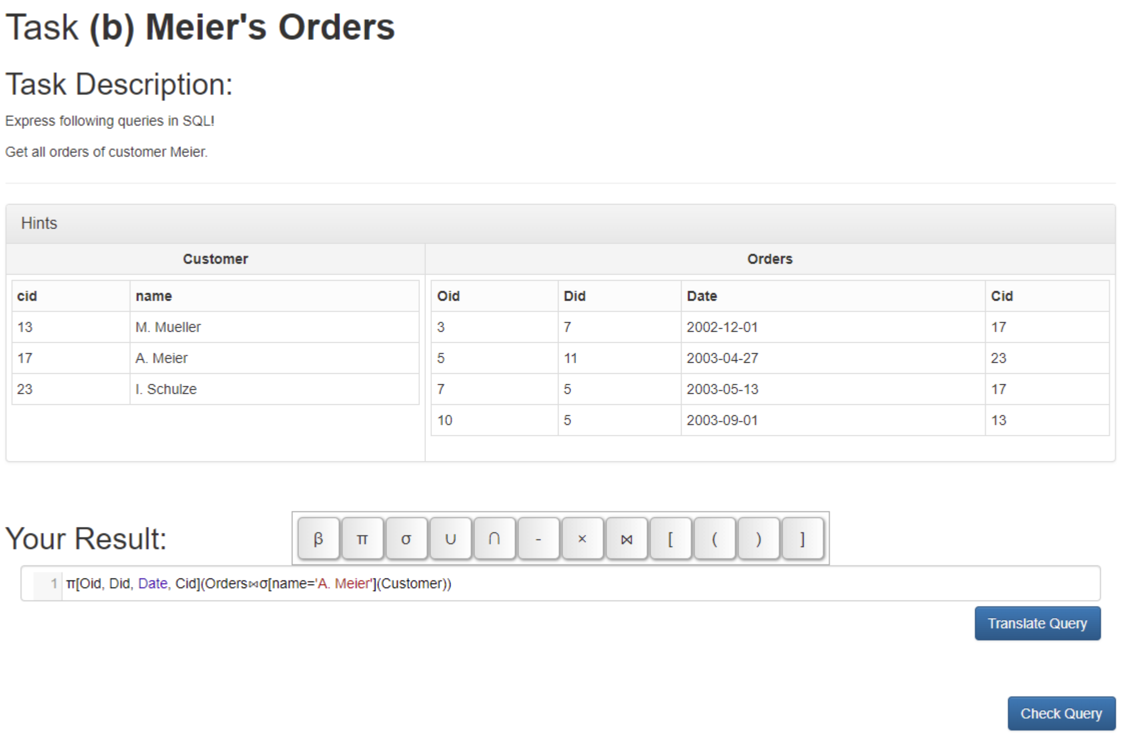
Orders

Oid	Did	Date	Cid
3	7	2002-12-01	17
5	11	2003-04-27	23
7	5	2003-05-13	17
10	5	2003-09-01	13

Your Result:

Figure 4.2: Snippet from the task

At the top you can see the task name "Meier's Orders" with its task description followed by a table the students have to perform the task on. Below that is an input field where the solution approach can be entered and checked with the "Check Query" button. It would be highly convenient if the relational algebra tasks would follow same patterns and UI behaviours in order to keep the user experience the same. In order to do that, additional behaviour was added to the *viewTask.tpl.php*, which defines this page in particular. The idea is that when a tutor or admin creates a task, that the type of task has to be selected. If the relational algebra option will be selected then the new behaviour and controls are added to the view in Figure 4.2. This control sequence will be part of the section Future work.
The relational algebra tasks can have two appearances that are pretty similar to the UI in Figure 4.2. They also have a task name and description but appear with or without a displayed table. In order to provide the necessary symbols a standard keyboard does not provide naturally, there is a virtual on-screen keyboard to fill in this gap (shown in Figure 4.3).



Task **(b) Meier's Orders**

Task Description:

Express following queries in SQL!

Get all orders of customer Meier.

Hints

Customer		Orders			
cid	name	Oid	Did	Date	Cid
13	M. Mueller	3	7	2002-12-01	17
17	A. Meier	5	11	2003-04-27	23
23	I. Schulze	7	5	2003-05-13	17
		10	5	2003-09-01	13

Your Result:

```
π[Oid, Did, Date, Cid](Orders⋈σ[name='A. Meier'](Customer))
```

Figure 4.3: Possible user interface of a relational algebra task including table view

In Figure 4.3 one can see an additional "Translate Query" button. This button provides the interface for future implementations and invokes the translation process of the relational algebra expression. In future implementations this functionality could be invoked by the check query button for relational algebra tasks making the button therefore obsolete. More improvements of this conceptualized idea are part of Future work. Further functions and behaviours are described in Debugging structures and provided interfaces.

4.1.3 Additional input symbols and deviations of notation

For practical purposes, complementing the translation framework and to avoid overloading of operators for precise statement expression, some operator behaviours are restructured with their notation. Instead of lowercasing the condition for the selection, projection and renaming operation after the associated operator symbol, square brackets are wrapped around the condition. Furthermore the use of underscores in names is forbidden because it defines an alternative operation that is not further implemented. The rename operation got an additional change on top. Instead of assigning a new column name with the $\leftarrow$ operator an argument list replaces it. To adjust to a classic list order it enlists the old Name before the new one. Thus allows to rename more columns at once by separating each new renaming condition with a comma. Accordingly, an even number of arguments is assumed.

$$\beta_{newName \leftarrow oldName}(TableName)$$

$$\beta[oldName, newName](TableName)$$

In order to simplify the input process for the users the virtual keyboard provides additional and common used symbols. These symbols will be pasted on the current cursor position in the input field and will be translated later on.

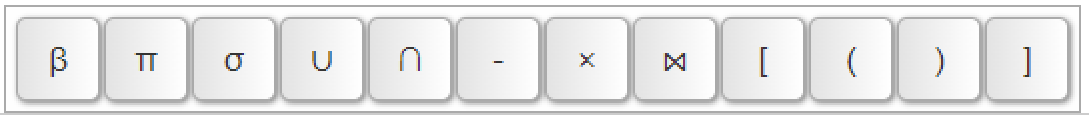


Figure 4.4: Virtual on-screen keyboard for faster and more convenient input

4.2 Architecture of relational algebra extension

The relational algebra extension consists basically of two parts: the JavaScript in the `viewTask.tpl.php` that combines the frameworks, performs some error detection while slightly changing framework mechanisms and the frameworks themselves. For now, the frameworks are focused before highlighting certain implementations of the JavaScript in Implementation.
As in Figure 4.5 depicted, the frameworks act independently from the base `viewTask.tpl.php` and are just connected over the added script, therefor forming the "*Relational algebra extension*" frame. The green highlighted abstract classes describe any visible interactable object on the frontend. Red containers illustrate therefore both used frameworks `virtual keyboard framework` [SGc22] and `ra-to-sql framework` [Jav22] with their most important components or derived features. The attribute lists for the abstract classes consist of four different groups to describe their behaviour to each other additionally: *provides*, *represents*, *allows* and *task end*. These control flow describing attributes mostly illustrate the use for the end-user. In order to grasp their different meanings one goes through the diagram beginning at the *CodeMirror input area* that was already provided from the SQLValidator system. It allows the student to enter any desired key on their

keyboard to do the common input. To perform a ***R**elational **A**lgebra statement input* that represents the solution to a task a student wants to formulate, additional symbols have to be added via the *virtual keyboard* that provides the mentioned additional relational algebra characters. After that the *parser* provides the translation process in combination with its dependant `sql_scope.js`, which provides direct SQL transformation rules, and the Backus-Naur form in `ra.jison`, which provides the transformation rules of the relational algebra staments. Both components of the parser will be featured in Implementation. To illustrate the overall control flow the parser allows two options: *on error* and *flawless*, therefor implying the status of the translation. "*On error*" the *hint output area* will provide the student with hints about their made mistakes meanwhile "*flawless*" recognizes a correct and error free translation to inform the student in the *solution message output*.

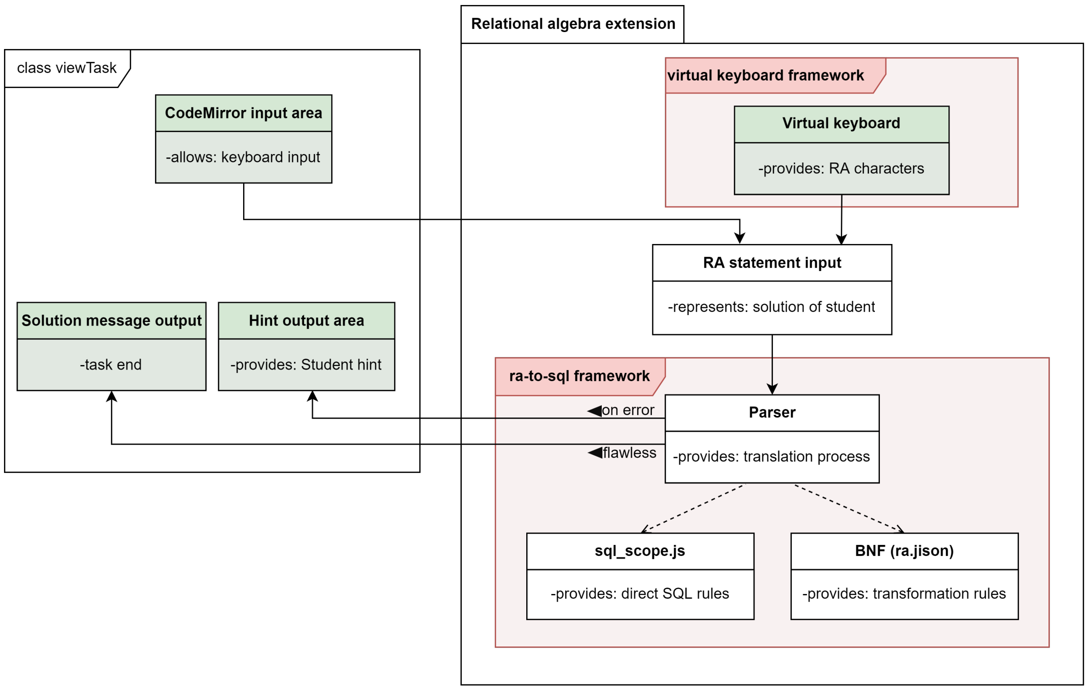


Figure 4.5: Adapted domain model diagram of most important components

4.3 Workflow of relational algebra tasks

Similar to the activity diagram that depicts the Workflow of SQL tasks, the adapted version for the relational algebra tasks will be highlighted in this section. Figure 4.6 illustrates the workflow on top of the SQL task workflow depicted at the bottom. To understand landing in the adjusted SQL part, one have to have a look at the new initial node first. Right after the initial node the fork node allows the activity to split up into two simultaneous actions: *enter keyboard character* and *enter relational algebra character*. In order to enter relational algebra character in the frontend of the application, the virtual keyboard "accept event" action fires and allows therewith special symbols like the attached note mentions.
After the forked actions in the frontend are joined into the *relational algebra solution approach*, the student should've entered the task related input, sending it to the

backend and waiting for a system result. The *relational algebra parser* in the backend will then try to translate the input statement. If there are errors regarding relational algebra statement integrity, the system will *detect* the *translation error type* and sends it to the *connector A*. *Connector A* results in an *output* of a *(relational algebra) specified hint* which then leads to a new task approach for the student with his new hint about his mistakes regarding relational algebra integrity. If, on the other hand, the *relational algebra parser* translates the statement correctly, *connector B* gets triggered.

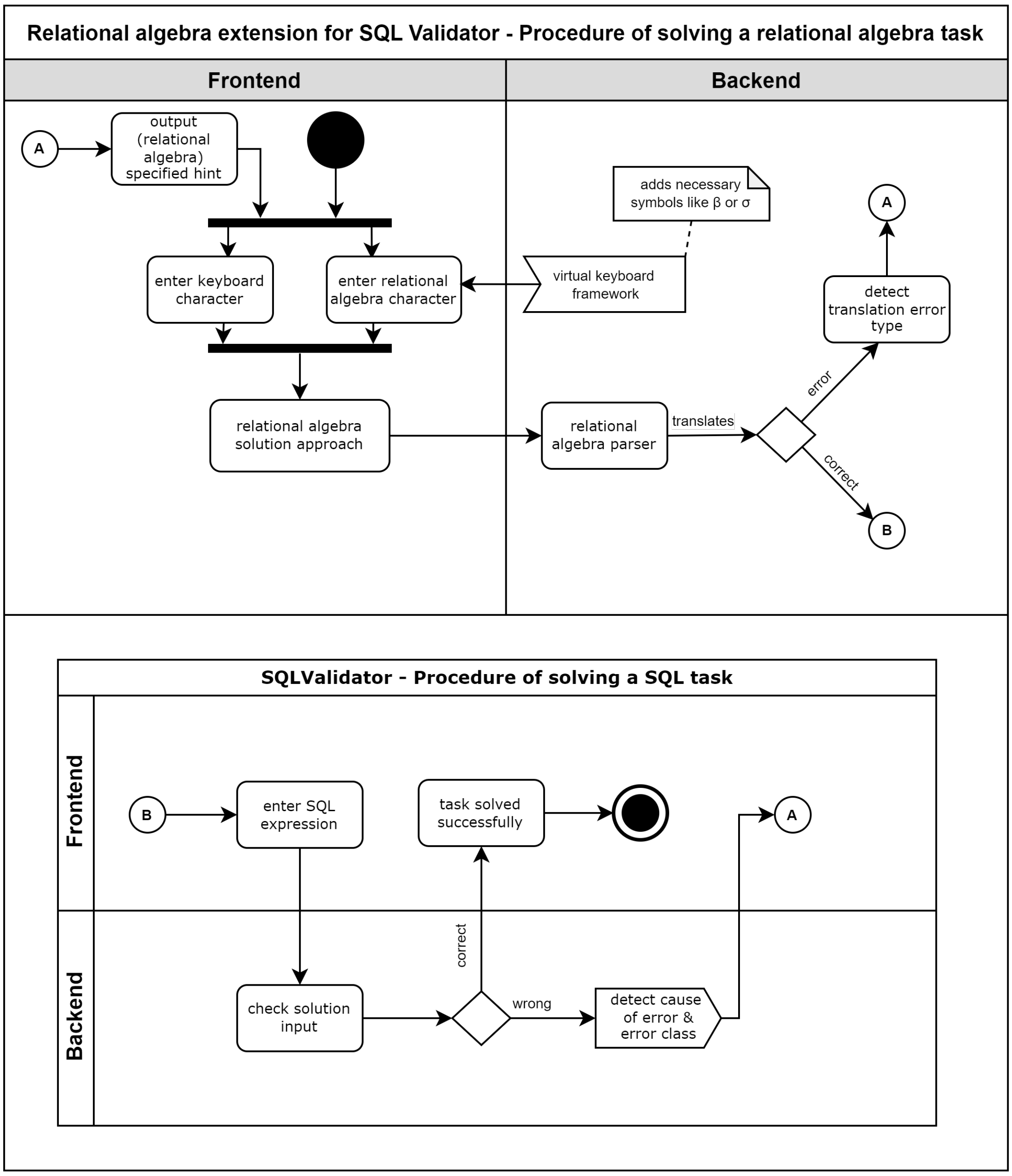

Relational algebra extension for SQL Validator - Procedure of solving a relational algebra task
Frontend
Backend
A
output (relational algebra) specified hint
enter keyboard character
enter relational algebra character
relational algebra solution approach
adds necessary symbols like β or σ
virtual keyboard framework
relational algebra parser
translates
error
correct
detect translation error type
A
B
SQLValidator - Procedure of solving a SQL task
Frontend
Backend
B
enter SQL expression
task solved successfully
A
check solution input
correct
wrong
detect cause of error & error class

Figure 4.6: Activity diagram about relational algebra task workflow

This connector will start a similar routine to the Figure 4.1. The translation will result in a SQL statement, that then will be entered in the "*enter SQL expression*" action. For visualization purposes this action is still in the frontend of the subdiagram but it will not be a part of it anymore. This control flow was not adapted in order to show the similarity of Figure 4.1 and Figure 4.6. Subsequently, the solution gets checked on the SQL level and can either generate a SQL related cause of error with according error type, which gets send to *connector A*, where the student can work on the new approach with the custom hint, **or** the student gets the output that the *task* is *successfully solved*. The bottom part of this figure has therefor pretty much the same interpretation but follows a slightly improved activity flow as the Activity diagram about the current SQL task workflow.

4.4 Error handling

Lastly, details about the way relational algebra errors are handled will be described. Therefore, this problem is divided into *syntax validation*, highlighting syntax level analysis, and *semantic verification*, describing type checking and verification of valid column references. Figure 4.6 already illustrated that the error detection is split up into the relational algebra part and the SQL part (both resulting in *connector A*). The relational algebra part will be handle by Syntax validation meanwhile the SQL part will be in the Semantic verification. *Query evaluation* is then performed by the same system as it was before and will not be mentioned any further.

4.4.1 Syntax validation

In order to check the correctness of the relational algebra input, two mechanisms intertwine. Whenever a syntax error of any sort occurs, the parser is unable to translate the relational algebra statement and throws a specific exception, thus defining the first mechanism. This exception needs to be classified and put into corellation with the students solution approach input, therefore defining the second mechanism. In order to generate fitting hints for the input, error classes were designed similar to the idea of [OBH+21] depicted in Figure 2.6. The Figure 4.7 illustrates three main categories the above mentioned relational algebra structure can be split up in: *operator*, *attribute* and *context*.

Assuming the structure of a relational algebra is $\mathit{Operator}[\mathit{Attribute}](\mathit{Context})$ every part can result in an error. Missing starting and closing parenthesis are mentioned in the according encapsulation blocks for *attribute* and *context*. On a higher level missing an *operator*, an *attribute* or *context* are depicted. The *context* can also be a relational algebra expression itself, therefore is subject to the same error classification. Set operations are deliberately omitted here from the syntax error analysis because they can not result in any technical problem regarding only the translation process to SQL. Their error potential is more of a semantic kind.

4.4.2 Semantic verification

As the Figure 4.6 suggests SQLValidator mechanisms are still used to perform the semantic verification. Just correctly translated relational algebra statements will be forwarded to the system, therefore excluding the occurrence of syntactical mistakes. Column references as well as type checking are already verified for SQL tasks (see Figure 2.6). In conclusion, the translation process results in a SQL statement so that the whole semantic analysis is exactly the same as for SQL tasks, therefor following existing behaviours extensively explained in [OBH+21].

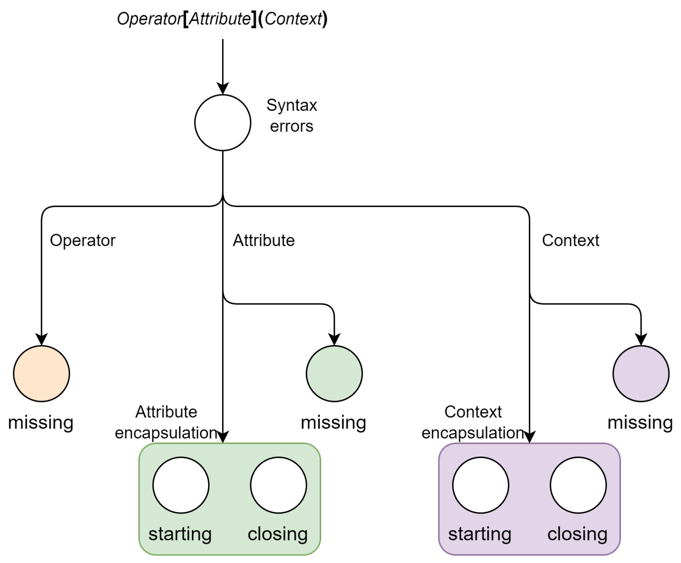


Figure 4.7: Derived error classes for parser error codes

4.5 Conclusion

In order to get a grasp of the whole relational algebra translation idea, small insights about already existing mechanisms and workflows of SQL tasks were provided before adapting the concept to it. Furthermore, the architecture and workflow of the translation extension were highlighted before discussing important key features regarding common mistakes students could run into when using the new relational algebra system. The next chapter will go into detail about the technical background and further implementations made to realize the explained concept.

5. Implementation

This chapter lays the focus on the technical realisation and problems that occur while implementing the above outlined Concept. At first, used technologies are presented before discussing the main frameworks `virtual keyboard` [SGc22] and `ra-to-sql` [Jav22] in greater detail. In the end the implementation progress of added code fragments is retrospectively discussed.

5.1 Technologies

To realise the concept, one has to have a look at used tools and technologies first. Furthermore special libraries with new file formats are presented.

5.1.1 Languages

The programming language used here has been chosen because it allows proper access to the used frameworks and therefore allow the extension of the SQLValidator project for a relational algebra translator. This addition runs solely on JavaScript for connecting the frameworks with one another, get the input from the DOM handler and performing the translation in the end. Additionally, a bit HTML was used for adding some control structures that simplify further implementations later on.

5.1.2 Development environments

Primarily `PHPStorm` 2021.1.2, a JetBrains IDE, was used because it supports the different languages used in the SQLValidator, like PHP or JavaScript, which is pretty neat when it comes to inspecting already existing mechanisms. It also provides tools like convenient syntax highlighting, error detection, code references, project wide searches as well as refactoring methods.
In order to compile the code fragments `XAMPP` [Apa22], a popular development environment for PHP by Apache, was used. `XAMPP` provides an easy to install Apache distribution that generates a local web server on which **A**pache, **M**ariaDB, **P**HP and **P**erl (=XAMPP, X stands for cross platform) are preconfigured for a comfortable development experience.

5.1.3 Libraries, essential frameworks and tools

Essentially all used frameworks operate through jQuery, a small JavaScript library. As already stated in Section 2.1, jQuery also provides an UI extension, which is also used in the `virtual keyboard` framework. Especially the event handling, HTML document traversal and manipulation are an essential part in this implementation. These functionalities are also essential in the `ra-to-sql` framework as well as in the `virtual keyboard` framework.
The `ra-to-sql` framework provides the needed tools to parse any relational algebra statement into a SQL statement excluding semantic checks on any database schema. This comes in handy because this step is already performed in the SQL-Validator environment, therefore representing a minimal but fitting solution for this implementation problem.
In order to provide additional needed characters for relational algebra statements the `virtual keyboard` framework is part of the implementation. Additionally, the framework grants convenient customizations and animations to level up the user experience when using it later on in the live SQLValidator environment.

5.2 Virtual on-screen keyboard

This section will focus on how the `virtual keyboard` framework, by Jeremy Satterfield and current maintainer Rob Garrison, was implemented rather than how it operates in general. Therefore some design choices and tweaks made along the implementation process are highlighted.

5.2.1 Used parameters

As the repository of this framework might suggests, there are a bunch of options to customize or specialize the virtual keyboard. Besides different design themes provided by jQuery UI, there are countless functionalities solely for the behaviour of the keyboard such as pop-up animations, language, displaying or input preferences. On top of that, extensions are implemented that support alternative keys, simulate typing, autocompletion and many more features.
In the following paragraph parameters, that were chosen for the keyboard, are inspected and the reason, why some of these were set, analysed (see Listing 5.1). After detecting the element and invoking the `.keyboard()` function on it, it starts with the "`layout: 'custom'`" setting which allows to use the below listed `customLayout`. Besides the `normal` configuration, shift and meta key sets would have been possible, which are not suitable for the purpose here. The disabled "`usePreview`" prevents displaying a preview window above the keyboard for the input process before inserting it into the desired input field. This behaviour would be confusing to use in combination with the normal keyboard the students use too. In order to deny minimizing the keyboard the "`alwaysOpen`" configuration is activated.
In general, the pressed keys are added to a string that represents the sequence of them. To set the order in which the pressed keys are added to the string the "`rtl`" parameter is set to false, therefore pasting the pressed key from left to right to the input string. This follows natural input behaviour. Some keys on the standard input devices of the user can trigger certain events on the virtual keyboard.

This mechanism can result in loosing the input focus, which harms the use significantly. Therefore this feature is deactivated by setting the parameters "`stayOpen`", "`autoAccept`" and "`autoAcceptOnEsc`" as they are. To reposition the keyboard on window resize the "`reposition`" parameter is set true.

```
$(function () {
   $('#codeMirrorInput').keyboard({
     layout: 'custom', //generate custom keyboard layout
     customLayout: {   //β is later transformed to ρ for ra-to-sql
         'normal': ['β π σ ∪ ∩ - × ⋈ [ ( ) ]']
     },
     usePreview: false,
     alwaysOpen: true,
     rtl: false,
     stayOpen: true,
     autoAccept: false,
     autoAcceptOnEsc: false,
     reposition: true,
     lockInput: true,      //prevents direct input, see 5.2.3
     preventPaste: false, //prevents direct input, see 5.2.3
     position: {
         of:  "#keyboardWrapper",
         my:  "center bottom",
         at:  "center bottom",
         at2: "center top",
         collision:  "fit"
     }

  //here is the entry point for the code listed in the listing 5.2.3

});
```

Listing 5.1: Settings of the implemented virtual on-screen keyboard

The SQLValidator uses another framework for optimizing the code input process called `CodeMirror`. This framework interferes with the `virtual keyboard` framework, which will be discussed in Section 5.2.2. For this purpose, the usage of the "`lockInput`" and "`preventPaste`" keyword will be highlighted later on.
In order to position the keyboard above the input field the "`position`" parameter group is modified. The "`of`" parameter attaches the keyboard to an according wrapper meanwhile every other configuration in this group changes the offsets towards it. There is also a code fragment, that reacts to scroll events to reposition the keyboard based on the position of the wrapper. This fragment is shown in Listing 5.2. Implementing this was necessary for updating the position when bigger tasks need more top space of the `viewTask` page or the size of the input field changes.

```
$(window).scroll(function () {
       $('#codeMirrorInput').getkeyboard().redraw();
});
```

Listing 5.2: Repositioning the keyboard on scrolling through window

5.2.2 Feature interaction with CodeMirror

As already mentioned in the previous chapter, a feature interaction of `CodeMirror` and the `virtual keyboard` framework was encountered during the implementation. This behaviour is shortly explained in advance. `CodeMirror` is an open-source JavaScript framework that provides any project with a versatile text editor popularly used for code inputs [Com22]. However, it pastes the configured and modern text area input field, which is used for the solution input, on top of the plain input field of the standard HTML document. Normally the `virtual keyboard` would paste its pressed keys in the preview window or, if this option is disabled like in this case, in the standard input field it is attached to.
So the main problem was, that there is no option to paste the input made by the keyboard into the overlying `CodeMirror` text area, therefore excluding the additional character set from contributing to the relational algebra statement input. In order to overcome this problem a solution was designed that avoids relying on general standard pasting of the virtual keyboard string by defining a certain callback method and avoiding the *acceptance routine* the `virtual keyboard` framework provides.

5.2.3 Workaround for feature interaction

The implementation idea is to use a provided callback method called *change* which performs the attached function whenever a change is happening to the keyboard or its string. Before considering this type of implementation one need to think of other events that can trigger the callback. That is the point where additional parameters in line fourteen and fifteen in the Listing 5.1 on the previous page were defined.

```
//------following code is part of the keyboard configuration------
 //this gets called on every user interaction with keyboard
 change: function (event, keyboard, el) {
     insertTextAtCursor(keyboard.$preview.val());
 }

 ///hidden field only for virtual keyboard & paste into CodeMirror
 function insertTextAtCursor(text) {
     const doc = window.editor.getDoc();
     const cursor = window.editor.getCursor();
     var lastInput = text.split("").reverse().join("").substr(-1);
     var position = {
         line: cursor.line,
         ch: cursor.ch
     }
     doc.replaceRange(lastInput, position);
 }
```

Listing 5.3: Code to work around the feature interaction

The parameter “`lockInput`” prevents any direct input into a preview window if it were configured otherwise, while the “`preventPaste`” keyword prevents any other content that is not virtual keyboard related from entering the input field. Defining the “`lockInput`” option was necessary because changes on the preview mode can be later implemented that would result in undesired behaviour for the *change* callback.

In the Listing 5.3 the *change* callback method calls `insertTextAtCursor` with the current value of the keyboard representing the input string. Whenever this string changes, the method gets called. The function `insertTextAtCursor` locates the `CodeMirror` editor with the respecting cursor position and pastes the desired character into the editor field, on pressing a virtual keyboard key. Note that the input is not pasted in the underlying element anymore, due to the parameter changes earlier in this section, but is rather redirected with the callback method to the cursor position of the `CodeMirror` editor.

5.3 Translating relational algebra via ra-to-sql

The following paragraphs will focus on the use of the `ra-to-sql` framework, some core components and adaptations that were made to transform relational algebra statements to SQL flawlessly. In order to understand this library one has to have a quick overview of the main components first. To shorten the mention of the semi-relevant mechanisms, they are reduced down in the following table:

File component	Function
src/grammar/ra.jison	contains the definition of the grammar
src/sql_scope.js	contains the transformation for converting to SQL
src/ra-to-sql.js	is the entry point of the library
src/ra.js	is the RA parser, which is built automatically by using the *Jison parser generator* [Con22]
dist/ra-to-sql.js	is the JavaScript library, which is bundled using the *Webpack bundler* [KEL+22]

Table 5.1: Main components (top) and important files of the build process (bottom)

Therefore generated components from the build process are pictured, because changing the Backus-Naur form or SQL translation patterns result in the need to build a new parser file with the *Jison parser generator*. Additionally, certain necessary adjustments regarding input adaptations will be featured in Section 5.4.

5.3.1 Backus-Naur form of contextfree grammar

In this framework a Backus-Naur form represents the foundation of the whole translation. The here used Backus-Naur form can be found in the Appendix - Backus-Naur form of *ra-to-sql framework*. When using the standard implementation of this grammar the same compatibility problem arises every time when nesting more than one statement.
After investigating the rule systems and the translation to SQL the problem could be narrowed down: some rules followed a mixed up order of commands. That let the suspicion arise that the grammar is mostly left linear instead of right linear which leads to the mix up of special character orders when following the "*Operator*[*Attribute*](*Context*)" structure. A right linear construct is needed for certain rules that follow the mentioned pattern because resolving them in the manner of $\mathcal{A} \Rightarrow x\mathcal{B}$ would fit the layered resolving process the best. $\mathcal{B}$ as a non-terminal

symbol represents thereby the modular "$[Attribute](Context)$" part whereas x as a terminal symbol is defined as an *Operator* and is therefore fixed. At first this should not be a problem because it still translates keywords separately. However, the translation order is important when having nested statements. In order to translate nested statements correctly one has to translate the inner statement first before translating the outer ones.

$$\sigma_{StudentId>2000}(\sigma_{LastName='Meier'}(Students)$$

To illustrate this specific solution concept, one considers the above example. For translating purposes one has to translate the inner part $\sigma_{LastName='Meier'}(Students)$ to `SELECT Students WHERE LastName='Meier'` first before adapting it to the context of $\sigma_{StudentId>2000}(Context)$ resulting in

```
SELECT DISTINCT * FROM (
  SELECT DISTINCT * FROM Students WHERE LastName='Meier')
AS SEL2 WHERE StudentId>2000
```

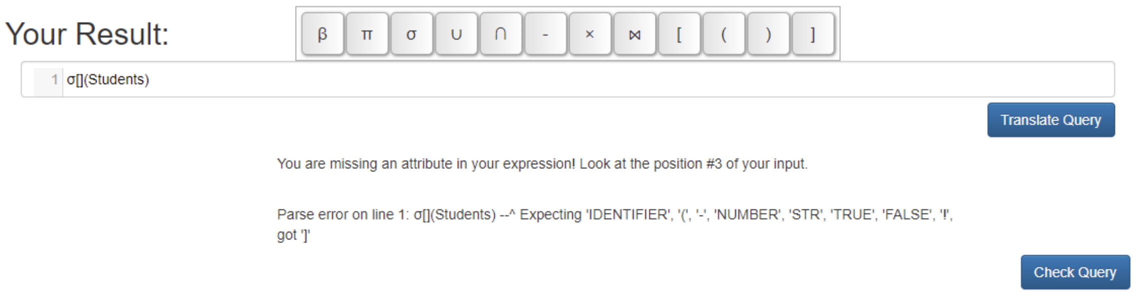
This pattern shares the same layering characteristic as the relational algebra statement, nesting the `LastName` selection in the `StudentId` selection. The effectiveness of this translation approach will be discussed in Evaluation.

5.3.2 Translation patterns for SQL in *sql_scope.js*

In order to provide correct translation structures, the parser follows the rule system of the grammar and transfers them to the SQL patterns. The most relevant SQL translation patterns can be found in the Appendix - Transformation patterns for SQL in *sql_scope.js*. If one has a closer look on these translation patters, a distinct code fragment stands out from all the different functions. The pattern `AS ${getNewId("`*CONTEXT*`")}` serves a very specific purpose when nesting queries. It helps formulating a specific query as a subquery before conjoining them with the outer queries, therefore avoiding errors by nested queries. However, this pattern is always executed which leads to a MariaDB error in the SQLValidator system for the outermost layer. This error is prevented by string processing, later discussed in Section 5.4.2.
In addition, the basic implementation of the *rename* operation was not implemented properly. To fix this issue a *helpRename* function was created as you can see in Listing A.7. This construct iterates through every attribute in the attribute list, alternates between "," and "`AS`" before applying the pattern to the respective column. However, it is assumed that the number of arguments is even and thus a unique assignment is made. If this is not the case, an *undefined* is included in the SQL output. This can also be filtered accordingly with string processing and is mentioned as a part in Future work.

5.3.3 Debugging structures and provided interfaces

Along the implementation process it was necessary to check different stages and results of the translation process. For this purpose three paragraph tags were added to the general structure of the `viewTask` page: *testing*, *userMistake* and *errorOutput*. They are used to output different strings along the process. The *testing* field outputs the translated SQL query if no mistakes were found in the relational algebra input.



```
1 σ[](Students)
```

You are missing an attribute in your expression! Look at the position #3 of your input.

Parse error on line 1: σ[](Students) --^ Expecting 'IDENTIFIER', '(', '-', 'NUMBER', 'STR', 'TRUE', 'FALSE', '!', got ']'

Figure 5.1: Snippet of debug messages when clicking on "*Translate Query*" button

As Figure 5.1 illustrates, *userMistake* displays the custom hint the student gets when finding flaws in the task input. It consists of the *custom hint message* and a *position* where the error occurred. In this example the *custom hint message* suggests the student to correct the attribute because it found none and to look on position three of his input as the *position* part proposes. `CodeMirror` offers different possibilities to mark or highlight text in the input area. Possible improvements of this type are discussed in Future work. The *errorOutput* holds the *parser error message* provided from the framework. Future implementations can dispense with these three displays and perform necessary forwarding of data types in the backend. Therefore, their implementation also represent interfaces at the same time.

5.4 String processing

This section focuses only on necessary adjustments of string processing of any sort. Most of the changes are made to the translated SQL statement and only a few are made to the relational algebra expression before being translated.

5.4.1 Input processing

Slight direct changes from the user input are necessary to make it compatible with the `ra-to-sql` framework. The framework itself uses ρ instead of β for the renaming operation. To adapt to this change, the following regular expression is implemented:

```
//change every β appearance to ρ globally
//ignore case sensitivity
userInput = userInput.replace(/β/gi, "ρ");
```

Note that the framework offers multiple operator appearances for some operators. They could be added manually in the input string but are not helpful in the learning process whatsoever. The abstraction level to the real appearance of relational algebra statements would be simply too enormous. Alternative operator names are illustrated in the Table 5.2 on the following page.

Operation	Intended use	Alternative
Rename	$\rho[a,b](A)$	Ren[a,b](A)
Projection	$\pi[a,b](A)$	Proj[a,b](A)
Selection	$\rho[Condition](A)$	Sel[Condition](A)
Cartesian Product	$A{\times}B$	A x B
Natural Join	$A{\bowtie}B$	A |x| B
Union	$A{\cup}B$	A U B
Intersection	$A{\cap}B$	A INT B

Table 5.2: Alternative operator names

More direct input processing is not necessary at this point.

5.4.2 Translated statement string processing

As already mentioned in Section 5.3.2 there is a MariaDB error in the SQLValidator system for the outermost layer when translating nested relational algebra statements. In order to get rid of this a routine, called *inputCleaning*, was implemented that counts the number of opening and closing parentheses with the help of the *countSymbol* function. If the last closing parenthesis is found the rest of the string is cut off.

```
//help function for counting symbol occurences
   function countSymbol(str, find){
       return (str.split(find)).length - 1;
   }

   //Crop the last known "AS  [operation name]" in string
   function inputCleaning(input){
       var parenthesisNum = countSymbol(input, "(");
       for(let i = 0; i < input.length; i++){
           if(input.charAt(i) == ")") {
               parenthesisNum = parenthesisNum - 1;
           }
           if (parenthesisNum == 0) {
               input = input.slice(0,i+1);
               break;
           }
       }
      return input;
   }
```

Listing 5.4: Solving the “AS” problem with MariaDB

With this pattern the parentheses, that are wrapped around the SQL statement, could also be removed but they do not harm the semantic meaning of the query whatsoever and are accepted by the SQLValidator “`Check Query`” functionality just fine.

There is also an additional case in which the SQL query needs further processing. After getting rid of the standard `SELECT DISTINCT * FROM` that was wrapped around every query, there was a case left that was still in need of this code snippet: the atomic table name. If this case would not be considered, table names without certain operations would not be translated to SQL whatsoever.

```
//whenever there is no operator symbol user --> add prefix
if (userInput.search(/(ρ|π|σ|∪|∩|-|×|⋈|\[|\(|\)|\])/g) == -1) {
        raExpression = "SELECT DISTINCT * FROM ".concat(raToSql.
            getSql(userInput));
    }
else {
        raExpression = raToSql.getSql(userInput);
    }
```

Listing 5.5: Regular Expression for detecting atomic tables

For this purpose the regular expression mechanism of JavaScript was used once again, illustrated in Listing 5.5. This code fragment searches for any operator symbol in the relational algebra input globally and whenever none of these appear in the whole string the "`SELECT DISTINCT * FROM`" snippet gets printed out in front of it.

5.4.3 Error handling

In the Syntax validation there was mentioned that the relational algebra translation extension will only handle syntactic errors by itself, leaving the semantic mistakes to the already existing SQLValidator routines. The seven described errors in Figure 4.7 are therefore listed as an attribute with the respective parser error in a dictionary.

Error class	Relevant parser error components
missingOperator	NEWLINE, ;, IDENTIFIER, (, PROJ, REN, SEL
startingAttribute	[
closingAttribute	),], -, COMMA, +, *, /, >, <, <=, >=, =, <>, OR, AND
missingAttribute	IDENTIFIER, (, -, NUMBER, STR
startingParenthesis	(
closingParenthesis	), UNION, INTERSECTION, PRODUCT, NATURAL, NATURAL1, -
missingRelation	IDENTIFIER, (, PROJ, REN, SEL

Table 5.3: Matching keywords to error classes within the error lookup dictionary

To get a simpler overview of the assignment of the error classes and their corresponding parser errors, Table 5.3 was created. It represents only the essential components of the parser error message, which are also recognized as missing in the most common problems. Note that only the most common mistakes are part of this error detection routine, which are derived as syntactic error classes. Total mixed up inputs, incorrect bracketing and gibberish answers are not part of that process. For further improvements and design ideas see Future work.

5.5 Conclusion

Besides various difficulties along the implementation process the result is still condensed to its bare functions. The overall implementation follows strictly the concept design ideas, be it regarding the facilitated input via the virtual keyboard, the simple relational algebra parser or the consideration of syntactic errors.

Attaching the script directly to the specific task page also matches the integration goal of fitting seamlessly into the SQLValidator environment. Provided interfaces also allow easier adaptations for future implementations and improvements. The decision to use plain JavaScript instead of more complicated structures in combination with `AJAX`, like the rest of the SQLValidator does, was made based on the versatility of JavaScript and `jQuery`. There was simply no need for `AJAX` because its main purpose revolves around asynchronous gradual updates to web pages which are not necessary when triggering an event by a button click and leaving the rest to the SQLValidator system. In order to evaluate the quality of the translation and possible inconveniences, following Chapter 6 will do into detail about it.

6. Evaluation

This chapter revolves around the quality of the translation by testing out sample tasks of a specific exercise sheet. Furthermore, translation patterns are examined for their differences from the standard solution. The quality of the syntax validation and semantic verification is also discussed.

6.1 Testing translated tasks on SQLValidators backend

In order to get a glimpse at the quality of the translation process, a Test on exercise sheet 9 - task 1 was performed on the system. The goal was to observe whether the solution developed for the tasks were semantically consistent with the translation from the parser or not. First of all, however, it should be taken into account that relational algebra keywords do not have a fixed translation in SQL. Moreover, the semantic meanings are important regardless of the composition or the keywords themselves in the statements over which they were generated.
The performed test on the relational algebra extensions is intended to show only the differences in possible answers from the set of all correct SQL translations and is not a representative sample. Nevertheless, the generated results show that different keywords still yield the semantically same meaning. For this purpose, a total of three specific examples is considered:

- ▷ task 1d) for the `UNION` keyword,
- ▷ task 1e) for the `NATURAL JOIN` keyword,
- ▷ task 1f) for the `EXCEPT` keyword with statement nesting.

As the Table A.4 shows, the results differ a lot from one another. The parser interprets the queries by themselves and joins them together afterwards with the `UNION` keyword representing the $\cup$ operation. On the other hand, only one binary operation is performed in the solution, which sets both attributes in relation to each other. To infer this solution, several steps are merged into each other. It is recognized that two selections, that refer to the same table, are to be united, which is also accomplished in the `WHERE` condition. So instead of uniting two selection queries, only the attributes of the queries are set in relation to each other.

The projection of the "*Date*" does not have to be wrapped around the union of both selections and can simply be placed in front of it.
Thus, both variants represent the same semantic sense, but the translation combines complete queries and the provided solution combines only the most important components of the queries. The standard solution to this task also has the semantic knowledge of the selections to use the same table. By leaving the semantic checking to the SQLValidator and not doing it separately in the relational algebra translator, such distinctions arise.

Table A.5 suggests a similar inconvenience. By translating the relational algebra statement with the provided extension tool, it results in cascaded `NATURAL JOIN`s from *Customer* to *Product*. The `NATURAL JOIN` keyword joins the tables together based on same named columns, but the extension does not know about the similarities between *Cid*, *Pid* and *Oid* the standard solution uses.
Accordingly, the example solution exploits semantic knowledge about the tables of the task again to achieve an optimal result. In this case, both solutions would only be semantically equivalent if corresponding columns, over which the composite is to be performed, are also named the same. Renaming respective columns with β or in the task would have the same effect.

Finally, the last example from Table A.6 highlights the semantic dependency of the standard solution compared to the nested translation of each component. Once more, there is a huge difference between the translation and the intended solution. The translation uses the `EXCEPT` keyword over nested queries representing the `NATURAL JOIN`, while the provided solution does not even perform a join. This is because it has been recognized that the *Customer* and *Orders* have the same relevant column, so you only need to query this commonality with `NOT IN`.

All in all, it can be stated that the translation schemes translate correctly, but without a direct semantic context they tend to be deeply convoluted and extraordinary long. Therefore making it a straight forward but not highly aware or intelligent algorithm. In addition, some edge cases tend to mess with the understanding of error causes.

6.2 Possible adaptations

The main problem of the parser is based on the pattern itself. In the current sample tasks, mentioned in Section 6.1 on the previous page, the standard solution approach is to interpret the query made by the relational algebra statement and create a matching SQL query. However, this procedure does not represent the approach from the relational algebra extension. The relational algebra translator will convert every query gradually and combines them with certain keywords for the respecting operation. This discrepancy arises from performing the semantic check just in the SQL stage. Would the same semantic mechanisms apply to relational algebra expression before translation, there would be no edge cases. To avoid edge cases, adaptations to the tasks where they occur, could also be an option. Further optimization suggestions are mentioned in the Section 7.2.

6.3 Evaluate core requirements

Finally, the translators behaviours regarding syntactic validation and semantic verification are discussed again. In particular, the extent to which these concepts have been taken into account and whether direct extensions or improvements can be imagined will be focused.

6.3.1 Syntax validation

The mechanism of looking up errors in an error class dictionary is an obvious solution. This way, the most common errors are covered, even if it is not possible to manually reproduce every possible error type that should be mentioned in the dictionary. That basically describes the biggest flaw in this implementation. An error can not be classified if there is no error class for it in the dictionary. Either one extends the dictionary in the future with the new error classes or one considers self-adapting data types, which generate error indications themselves. Spelling mistakes of database related names are checked from the SQL task backend and are therefore always detected.

6.3.2 Semantic verification

Semantic integrity of the relational algebra statements themselves is also checked through the error lookup dictionary to some sort. Mistakes like mixing up attribute and condition of a relational algebra statement are part of it. However, column references as well as semantic commonalities between tables with their respective columns are not considered in my implemented solution approach. They are checked later on, but are not a part during or previous to the translation, resulting in long, query-based translations to SQL. So an interpreter extension would take care of that. However, when avoiding edge cases or providing adapted tables the translator works just fine without direct semantic context understanding.

6.3.3 Workflow changes

The relational algebra extension made also couple workflow changes. To grasp those changes, the previous structure of the SQLValidator needs to be reviewed. Therefore, the focus is set exclusively on the user behaviour part.
The user submits a query to the *PHP Server* via the *Web Interface*. Afterwards, the *PHP Server* executes the query on the task-specific database copies of the master database. The result is then compared with the result from the master database. Error messages are then returned to the user via the *Web Interface*. This structure in Figure 6.1 on the following page represents the workflow for SQL tasks.

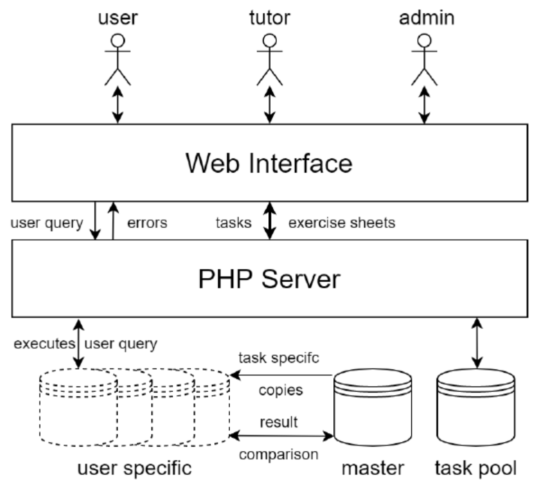


Figure 6.1: Architecture of SQLValidator according to [OBH+21]

In order to grasp the semantic error difference once more, an adjusted version of the diagram (Figure 6.2) for relational algebra tasks was created. The meaning for tutors and administrators is the same, so again only the user part will be discussed. Basically, the procedure is the same, but this time the *Relational Algebra Translator* is interposed. User queries are sent to the translator instead of directly to the *PHP Server* and are syntactically analyzed. Afterwards, the translated SQL query is forwarded to the server, as long as the provided interface has been linked. From there, semantic errors regarding the translated SQL expression are finally returned to the user.

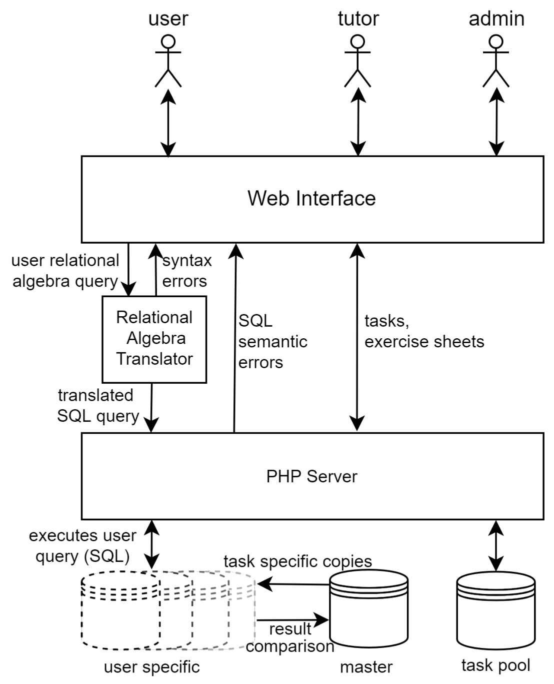


Figure 6.2: Adjusted architecture of SQLValidator with *Relational Algebra Translator*

These graphics show once more, that semantic context is not part of the translation process and is only performed on the translated SQL query on the *PHP Server*. It can also be observed that the path of the input from the user to the verification at the *PHP Server* becomes longer. However, the processing time should not be significantly higher, since the connecting JavaScript is kept minimal and does not perform any time-consuming tasks. All in all, it can be stated that the translation could be done quickly despite the additional components.

6.4 Conclusion

This chapter gave a hint about edge cases and how the translator performs compared to the standard solution approach. Furthermore, the limitations of the implementation were discussed while the differences between the relational algebra translator and an interpreter were presented. Therefore, the relational algebra extension is classified more as a translator due to its lack of semantic understanding.

7. Conclusion and future work

Finally, the chapters get briefly retrospectively reviewed and point out possible places for future work.

7.1 Conclusion

In this thesis, the conceptual design and implementation of a relational algebra translator for the SQLValidator is addressed. The SQLValidator is an online teaching and learning tool that is intended to support university students in learning SQL. However, it lacks relational algebra, that is already taught at the Otto-von-Guericke University but is not a part of the SQLValidator yet.

At the beginning essential backgrounds are explained to understand the used concepts. For this purpose languages like JavaScript or SQL were introduced together with more theoretical context like Backus-Naur form, regular expressions, relational database management, relational algebra and the SQLValidators architecture. The related works chapter presents similar tools and papers that had similar procedures in implementing such a translator. Subsequently, the conceptual design is presented in chapter 4. The initial focus was on the requirements for the system integration, before the general design of the extension and its error analysis were discussed. Thereupon, the implementation of the individual components is discussed in detail, before highlighting special processing of the input. Finally, the system was tested in the evaluation on some sample tasks and selected solutions were discussed with the respective weakness of the implementation.

7.2 Future work

The implemented solution approach leaves some interfaces open for improvement. So far, administrators and tutors cannot yet distinguish between relational algebra and SQL tasks. Therefore, such a distinction in the tutor menu would be very helpful. In addition, only the interfaces for the translator have been provided, i.e. the complete integration into the SQLValidator is still pending.

Furthermore, a separate error code can still be made for an odd number of attributes in the *Rename* operation. It is also conceivable to add an error class for missing apostrophes in specific attributes. Both last mentioned changes are optional.
Furthermore, an interface for the error position is given, which can be used for error highlighting in CodeMirror. In addition, one could adapt the error detection generally to a dynamic data type, or implement a similar system to the error detection of the SQL tasks. It should also be possible to functionally outsource the JavaScript or align it with different AJAX components.
All in all, the biggest implementation should be the addition of a relational algebra interpreter which, just like the SQL tasks, interprets the semantics of the tasks context and points out specific errors.

A. Appendix

A.1 Backus-Naur form of *ra-to-sql framework*

Click here to get back to where this context was mentioned.

```
ra_program
    : ra_sentences EOF
        { return $1 }
    | ra_sentences sentence_separators EOF
        { return $1 }
    | sentence_separators ra_sentences sentence_separators EOF
        { return $2 }
    | sentence_separators ra_sentences EOF
        { return $2 }
    ;

ra_sentences
    : ra_sentence
        { $$ = new Array($1); }
    | ra_sentences sentence_separators ra_sentence
        { $1.push($3); $$ = $1; }
    ;

sentence_separators
    : sentence_separator
    | sentence_separators sentence_separator
    ;

sentence_separator
    : NEWLINE
    | ';'
    ;
```

Listing A.1: Basic structure of relational algebra statement

```
ra_sentence
    : IDENTIFIER '<-' ra_expression
        { $$ = { type: 'identifier', value: { id: $1, expression:
            $3.value} }; }
    | IDENTIFIER '(' field_list ')' '<-' ra_expression
        { $$ = { type: 'identifier', value: { id: $1, expression:
            $6.value, fields: $3 } }; }
    | ra_expression
        { $$ = { type: 'expression', value: $1 }; }
    ;

ra_expression
    : '(' ra_expression ')' { $$ = {id: yy.getNewId('GROUP'), value
        : $2.value }; }
    | tableName { $$ = {id: yy.getNewId('ID'), value: $1 }; }
    | projection { $$ = {id: yy.getNewId('PROJ'), value: $1 }; }
    | selection { $$ = {id: yy.getNewId('PROJ'), value: $1 }; }
    | union { $$ = {id: yy.getNewId('UNION'), value: $1 }; }
    | intersection { $$ = {id: yy.getNewId('UNION'), value: $1 }; }
    | product { $$ = {id: yy.getNewId('PROD'), value: $1 }; }
    | natural { $$ = {id: yy.getNewId('PROD'), value: $1 }; }
    | theta { $$ = {id: yy.getNewId('PROD'), value: $1 }; }
    | subtraction { $$ = {id: yy.getNewId('SUBS'), value: $1 }; }
    | rename { $$ = {id: yy.getNewId('REN'), value: $1 }; }
    ;

tableName
    : IDENTIFIER
        { $$ = yy.getSingleTable($1); }
    ;

projection
    : PROJ '[' field_list ']' '(' ra_expression ')'
        { $$ = yy.getProjection($6.value, $6.id, $3); }
    ;

rename
    : REN '[' field_list ']' '(' ra_expression ')'
        { $$ = yy.getRename($6.value, $6.id, $3); }
    ;

selection
    : SEL '[' bool_expression ']' '(' ra_expression ')'
        { $$ = yy.getSelection($6.value, $6.id, $3); }
    ;
```

Listing A.2: Basic operations

```
union
    : ra_expression UNION ra_expression
        { $$ = yy.getUnion($1.value, $3.value); }
    ;

intersection
    : ra_expression INTERSECTION ra_expression
        { $$ = yy.getIntersection($1.value, $3.value); }
    ;

product
    : ra_expression PRODUCT ra_expression
        { $$ = yy.getProduct($1.value, $3.value); }
    ;

natural
    : ra_expression NATURAL ra_expression
        { $$ = yy.getNaturalJoin($1.value, $3.value); }
    ;

theta
    : ra_expression NATURAL1 '(' bool_expression ')' NATURAL2
       ra_expression
        { $$ = yy.getTheta($1.value, $7.value, $4); }
    ;

subtraction
    : ra_expression '-' ra_expression
        { $$ = yy.getSubtraction($1.value, $3.value); }
    ;
```

Listing A.3: Set operations

```
field_list
    : e {$$ = new Array($1)}
    | field_list COMMA e
        {
            $1.push($3);
            $$ = $1;
        }
    ;

e
    : e '+' e
        { $$ = $1 + '+' + $3; }
    | e '-' e
        { $$ = $1 + '-' + $3; }
    | e '*' e
        { $$ = $1 + '*' + $3; }
    | e '/' e
        { $$ = $1 + '/' + $3; }
    | '-' e %prec UMINUS
        { $$ = '-' + $2;}
    | '(' e ')'
        { $$ =  '(' + $2 + ')'; }
    | NUMBER
        { $$ = Number(yytext); }
    | IDENTIFIER
    | STR
        { $$ = "'" + $1 + "'" }
    ;

b_e
    : e bool_operator e
        { $$ = yy.getBooleanOperation($1, $2, $3); }
    ;

bool_operator
    : '>'
    | '<'
    | '<='
    | '>='
    | '='
    | '<>'
    ;

bool_expression
    : factor
    | factor bool_op bool_expression
        { $$ = yy.getBooleanExpression($1, $2, $3); }
    ;

bool_op
    : OR
    | AND
    ;
```

Listing A.4: Expression additions and boolean operations

```
term
    : factor
    | factor AND factor
        { $$ = yy.getAnd($1, $3); }
    ;

factor
    : TRUE
    | FALSE
    | b_e
    | '!' factor
        { $$ = yy.getNot($2); }
    | '(' bool_expression ')'
        { $$ =  '(' + $2 + ')'; }
    ;
```

Listing A.5: Term and factor extension

A.2 Transformation patterns for SQL in *sql_scope.js*

Click here to get back to where this context was mentioned.

```
 function getNot(input) {
    return "NOT " + input;
}

function getBooleanExpression(op1, b_e, op2) {
    return `${op1} ${b_e} ${op2}`;
}

function getBooleanOperation(op1, operation, op2) {
    return op1 + operation + op2;
}

//original versions of following commands are commented out
function getUnion(sentence1, sentence2) {
    //return `(${getTableFromSentence(sentence1)} UNION
    //${getTableFromSentence(sentence2)})`;
    return `(${getTableFromSentence(sentence1)} UNION ${
        getTableFromSentence(sentence2)}) AS ${getNewId("UNION")}`;
}

function getIntersection(sentence1, sentence2) {
    //return `(${getTableFromSentence(sentence1)} INTERSECT
    //${getTableFromSentence(sentence2)})`;
    return `(${getTableFromSentence(sentence1)} INTERSECT ${
        getTableFromSentence(sentence2)}) AS ${getNewId("INTERSECT")
        }`;
}

//Except is dialect, for more info see [Smi02]
function getSubtraction(sentence1, sentence2) {
    //return `(${getTableFromSentence(sentence1)} EXCEPT
    //${getTableFromSentence(sentence2)})`;
    return `(${getTableFromSentence(sentence1)} EXCEPT ${
        getTableFromSentence(sentence2)}) AS ${getNewId("SUB")}`;
}

function getProduct(sentence1, sentence2) {
    //return `(SELECT DISTINCT * FROM ${sentence1}, ${sentence2})`;
    return `(SELECT DISTINCT * FROM ${sentence1}, ${sentence2}) AS
        ${getNewId("PROD")}`;
}
```

Listing A.6: SQL patterns for set operations

```
function getNaturalJoin(sentence1, sentence2) {
  //return `(${getTableFromSentence(sentence1)} NATURAL JOIN
  //${sentence2})`;
  return `(${getTableFromSentence(sentence1)} NATURAL JOIN ${sentence2}) AS ${getNewId("NAT")}`;
}

function getTheta(sentence1, sentence2, condition) {
  //return `(${getTableFromSentence(sentence1)} JOIN
  //${sentence2} ON ${condition})`;
  return `(${getTableFromSentence(sentence1)} JOIN ${sentence2} ON ${condition}) AS ${getNewId("NAT")}`;
}

function getSelection(table, alias, condition){
  //return `(SELECT DISTINCT * FROM ${table} WHERE ${condition})`;
  return `(SELECT DISTINCT * FROM ${table} WHERE ${condition}) AS ${getNewId('SEL')}`;
}

function getProjection(table, alias, fieldList){
  //return `(SELECT ${fieldList} FROM ${table})`;
  return `(SELECT DISTINCT ${fieldList} FROM ${table}) AS ${getNewId('PROJ')}`;
}

//allows alternating , and AS in renaming process
function helpRename (fieldList) {
  var solutionString = "";
  if(fieldList.length %2 == 0) {
      for (let i = 0; i < fieldList.length; i++) {
          if (i % 2 == 0) {
              solutionString = solutionString.concat(fieldList[i], " AS ");
          } else
               solutionString = solutionString.concat(fieldList[i], ", ");
      }
      return solutionString.substr(0,solutionString.length-2);
   }
//if not even number of arguments returns undefined
//ToDo: error handling for undefined in SQL output
}

function getRename(table, alias, fieldList){
   return `(SELECT ${helpRename(fieldList)} FROM ${table}) AS ${getNewId('REN')}`;
   //return `(SELECT DISTINCT ${fieldList.map(x => `null as
   //${x}`).join(',')} WHERE 1=2 UNION SELECT DISTINCT * FROM
   //${table}) AS ${getNewId('REN')}`;
}
```

Listing A.7: SQL patterns for joins, selection, projection and rename

```
function getSingleTable(tableName) {
    return `##${tableName}##`;
}

function getTableFromSentence(sentence) {
    return `SELECT * FROM ${sentence}`;
}
```

Listing A.8: SQL patterns for atomic table handling

A.3 Test on exercise sheet 9 - task 1

Task:	$\pi_{oid}(Line\_item)$
Reformulation:	$\pi[oid](LineItem)$
Note:	underscores forbidden, better: CamelCase
SQL-Translation:	(SELECT DISTINCT oid FROM LineItem)
Solution:	SELECT DISTINCT oid FROM LineItem;

Table A.1: Exercise sheet 9 task 1 a)

Task:	$\pi_{Name}(Dealer \bowtie Orders)$
Reformulation:	$\pi[Name](Dealer \bowtie Orders)$
Note:	
SQL-Translation:	(SELECT Name FROM (SELECT * FROM Dealer NATURAL JOIN Orders) AS NAT1)

Solution:	SELECT Name FROM Dealer, Orders
	WHERE Dealer.DID = Orders.DID

Table A.2: Exercise sheet 9 task 1 b)

Task:	$\pi_{Did}(Dealer) - \pi_{Did}(Offers)$
Reformulation:	$\pi[Did](Dealer) - \pi[Did](Offers)$
Note:	”SELECT * FROM” to ”unite” both query results
SQL-Translation:	(SELECT * FROM (SELECT Did FROM Dealer) AS PROJ1 **EXCEPT** SELECT * FROM (SELECT Did FROM Offers) AS PROJ2)

Solution:	SELECT Did FROM Dealer WHERE Did
	NOT IN(SELECT Did FROM Offers);

Table A.3: Exercise sheet 9 task 1 c)

Task:	$\pi_{Date}((\sigma_{Date<01.03.2003}(Orders)) \cup (\sigma_{Date>01.05.2003}(Orders)))$
Reformulation:	$\pi[Date]((\sigma[Date < \text{‘01.03.2003‘}](Orders))\cup$
	$(\sigma[Date > \text{‘01.05.2003‘}](Orders)))$
Note:	apostrophes are necessary
	(SELECT Date FROM (
	SELECT * FROM (
	SELECT DISTINCT * FROM Orders
	WHERE Date<’01.03.2003’)
	AS SEL1
SQL-Translation:	**UNION**
	SELECT * FROM (
	SELECT DISTINCT * FROM Orders
	WHERE Date>’01.05.2003’)
	AS SEL2)
	AS UNION1)
Solution:	SELECT Date FROM Orders WHERE
	Date<’01.03.2003’ OR Date>’01.05.2003’;

Table A.4: Exercise sheet 9 task 1 d)

Task:	$Customer \bowtie Orders \bowtie Line\_item \bowtie Product$
Reformulation:	$Customer \bowtie Orders \bowtie LineItem \bowtie Product$
Note:	underscores forbidden, better: CamelCase
	(SELECT * FROM (
	SELECT * FROM (
	SELECT * FROM Customer
	NATURAL JOIN
SQL-Translation:	Orders) AS NAT1
	NATURAL JOIN
	lineItem) AS NAT2
	NATURAL JOIN
	Product)
	SELECT * FROM Customer, Orders, LineItem, Product
	WHERE Customer.CID = Orders.CID
Solution:	AND Orders.OID = LineItem.OID
	AND LineItem.PID = Product.PID;

Table A.5: Exercise sheet 9 task 1 e)

Task:	$Customer - \pi_{Cid,Name}(Customer \bowtie Orders)$
Reformulation:	$Customer - \pi[Cid, Name](Customer \bowtie Orders)$
Note:	

SQL-Translation:	(SELECT * FROM Customer
	  EXCEPT
	SELECT * FROM (
	    SELECT Cid,Name FROM (
	      SELECT * FROM Customer
	        NATURAL JOIN
	      Orders)
	    AS NAT2)
	AS PROJ3)

Solution:	SELECT * FROM Customer WHERE Cid
	NOT IN (SELECT Cid FROM Orders);

Table A.6: Exercise sheet 9 task 1 f)

Task:	$(Customer \times Product) - \pi_{Cid,Name,Pid,Label}(Customer \bowtie Orders \bowtie Line\_item \bowtie Product)$
Reformulation:	$(Customer \times Product) - \pi[Cid, Name, Pid, Label]$ $(Customer \bowtie Order \bowtie LineItem \bowtie Product)$
Note:	exceptionally long through multiple natural joins

SQL-Translation:	`(SELECT * FROM (`
	`   SELECT DISTINCT * FROM Customer, Product)`
	`AS PROD3`
	`   EXCEPT`
	`SELECT * FROM (`
	`   SELECT DISTINCT Cid,Name,Pid,Label FROM (`
	`      SELECT * FROM (`
	`         SELECT * FROM (`
	`            SELECT * FROM Customer`
	`               NATURAL JOIN`
	`            Order) AS NAT8`
	`         NATURAL JOIN`
	`         LineItem) AS NAT11`
	`      NATURAL JOIN`
	`      Product) AS NAT14`
	`   )`
	`AS PROJ16)`

Solution:	`SELECT * FROM Customer, Product`
	`WHERE (Cid, Name, Pid, Label)`
	`NOT IN (`
	`   SELECT Customer.Cid, Name, Product.Pid, Label`
	`   FROM Customer, Orders, LineItem, Product`
	`   WHERE Customer.Cid = Orders.Cid`
	`      AND Orders.Oid = LineItem.Oid`
	`      AND LineItem.Pid = Product.Pid`
	`);`

Table A.7: Exercise sheet 9 task 1 g)

Bibliography

[AdV14] Tamim Alkhalifah and Denise de Vries. Relational algebra interpreter. *International Conference on Advanced Information and Communication Technology for Education*, 2014. (cited on Page 20)

[Apa22] Apache. Xampp installers and downloads for apache friends, 16.05.2022. https://www.apachefriends.org/de/index.html. (cited on Page 29)

[Bac63] J. W. Backus. Revised report on the algorithmic language algol 60. *The Computer Journal*, 5(4):349–367, 1963. (cited on Page ix and 5)

[Cod07] E. F. Codd. Relational database: a practical foundation for productivity. In *ACM Turing Award Lectures*, page 1981. Association of Computing Machinery, New York, 2007. (cited on Page 8)

[Com22] CodeMirror Community. Codemirror versatile text editor, 30.04.2022. https://codemirror.net/. (cited on Page 32)

[Con22] GitHub Contributers. Github - zaach/jison: Bison in javascript, 17.05.2022. https://github.com/zaach/jison. (cited on Page 33)

[Fou22] JS Foundation. jquery api documentation, 25.04.2022. https://api.jquery.com/. (cited on Page 4)

[Jav22] Javier Rebagliatti. ra-to-sql - npm, 16.05.2022. https://www.npmjs.com/package/ra-to-sql. (cited on Page 24 and 29)

[KDHP15] Olessia Karpova, Noel D'Souza, Diane Horton, and Andrew Petersen. RAPT: Relational Algebra Parsing Tools. In Valentina Dagienė, Carsten Schulte, and Tatjana Jevsikova, editors, *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, page 334, New York, NY, USA, 2015. ACM. (cited on Page 19)

[KEL$^+$22] Tobias Koppers, Johannes Ewald, Sean Larkin, Kees Kluskens, and GitHub Contributers. Github - webpack/webpack: A bundler for javascript and friends. packs many modules into a few bundled assets. code splitting allows for loading parts of the application on demand. through "loaders", modules can be commonjs, amd, es6 modules, css, images, json, coffeescript, less, ... and your custom stuff, 17.05.2022. https://github.com/webpack/webpack. (cited on Page 33)

[Kes22] Johannes Kessler. Relax - relational algebra calculator, 31.03.2022. https://dbis-uibk.github.io/relax/landing. (cited on Page 18)

[Kle51] S. C. Kleene. Representation of events in nerve nets and finite automata. *Research Memorandum*, 15.12.1951. https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM704.pdf. (cited on Page 6)

[LR10] Ratnesh Litoriya and Anshu Ranjan. Implementation of relational algebra interpreter using another query language. In *2010 International Conference on Data Storage and Data Engineering*, pages 24–28. IEEE, 2010. (cited on Page 19)

[MDN22a] MDN contributors. Javascript | mdn, 23.04.2022. https://developer.mozilla.org/en-US/docs/Web/JavaScript. (cited on Page 3)

[MDN22b] MDN contributors. About javascript - javascript | mdn, 24.04.2022. https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript. (cited on Page 3 and 4)

[MDN22c] MDN contributors. First-class function - mdn web docs glossary: Definitions of web-related terms | mdn, 24.04.2022. https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function. (cited on Page 3)

[MDN22d] MDN contributors. Regular expressions - javascript | mdn, 24.04.2022. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions. (cited on Page ix, 6, and 7)

[OBH$^+$21] Victor Obionwu, David Broneske, Anja Hawlitschek, Veit Köppen, and Gunter Saake. Sqlvalidator – an online student playground to learn sql. *Datenbank-Spektrum*, 21(2):73–81, 2021. (cited on Page ix, 14, 15, 27, and 42)

[Raj22] Naveen Rajshekhar. Relational-algebra — bitbucket, 22.05.2022. https://bitbucket.org/naveenraj16/relational-algebra/src/master/. (cited on Page 17)

[Saa18] Gunter Saake. *Datenbanken: Konzepte und Sprachen*. mitp Verlags, Frechen, 6. auflage edition, 2018. https://learning.oreilly.com/library/view/-/9783958457782/?ar. (cited on Page ix, 9, 11, and 12)

[SGc22] Jeremy Satterfiel, Rob Garrisson, and other contributors. Github - mottie/keyboard: Virtual keyboard using jquery, 16.05.2022. https://github.com/Mottie/Keyboard. (cited on Page 24 and 29)

[Smi02] Ian Smith. New except, intersect and minus operators. *Rdb Journal - technical corner*, May 2002. https://download.oracle.com/otndocs/products/rdb/pdf/tech_archive/except_intersect_minus_ops.pdf. (cited on Page 52)

[Yan17] Jun Yang. Radb 3.0.4 documentation, 2017. https://users.cs.duke.edu/~junyang/radb/. (cited on Page 18)

[Yan22] Jun Yang. Ra: A relational algebra interpreter, 22.05.2022. https://users.cs.duke.edu/~junyang/ra2/. (cited on Page 18)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 25.05.2022

[illegible]