Faculty of Informatics, Masaryk University

Relational Algebra Expression Evaluation

Bachelor's Thesis

Lucie Molková

Brno, spring 2009

Declaration

Thereby I declare that this thesis is my original work, which I have created on my own. All sources and literature used in writing the thesis, as well as any quoted material, are properly cited, including full reference to its source.

Advisor: RNDr. Vlastislav Dohnal, Ph.D.

Acknowledgements

I would like to thank my advisor RNDr. Vlastislav Dohnal, Ph.D. for the help and motivation he provided throughout my work on this thesis and to Loren K. Rhodes, Ph.D. for valuable inputs. Special thanks go to Tomáš Janoušek and Bc. Petr Ročkai for support and ideas that saved me a lot of time.

Abstract

Relational algebra is a query language that is being used to explain basic relational operations and their principles. Many books and articles are concerned with the theory of relational algebra, however there is no practical use for it. One of the reasons is that no software exists that would allow a real use of relational algebra. Most of the currently used relational database management systems work with SQL queries. This work therefore describes and implemets a tool that transforms a relational algebra expressions into SQL queries, allowing the expressions to be evaluated in standard databases. This work also defines a new, applicable syntax for relational algebra and describes its grammar in order to ensure the correctness of the expressions.

Keywords

Relationa Algebra, SQL, Transformation, Perl, Parsing, Parse::Yapp, Syntax Analysis, Semantic Analysis

Contents

Chapter 1

Introduction

In 1970, E.F. Codd proposed a new model for database systems named the relational model. The relational model, built on a mathematical basis, provided the foundation for current database systems. It not only had an impact on the theory, but also on the development of database systems.

A query language is a language in which a user requests information from the database. Query languages are usually higher-level languages than standard programming languages. In essence, we can categorize query languages as being either procedural or non-procedural [15]. The difference lies in their approach to obtain the result. When a user wants to obtain a result using a procedural language, he or she needs to instruct the system to perform a specific sequence of tasks on the database. In a nonprocedural language, the user only needs to describe the desired information without giving the system a specific procedure.

Relational algebra is a good example of procedural language, while relational calculus is representative of non-procedural languages [15]. However, most query languages used in current relational database systems combine elements of both the procedural and non-procedural approaches.

1.1 Objectives

Relational algebra is a language that is being used to explain basic relational operations and principles. When one opens up a book covering the topic of relational model and databases, there is a high probability that the book will contain a chapter dedicated to relational algebra. The relational algebra is usually explained and then forgotten, once the SQL is introduced. One of the reasons that relational algebra is not being used in practical life is that there is not a tool that would allow the evaluation of such expression. The objective of this work is to create a program that will accomplish this task.

The relations are stored in a database and the results from a database can be obtained by using database queries. In the field of relational databases, SQL became a standard that is now being used worldwide. This work is not an exception and

uses SQL as well. To evaluate a relational algebra expression, it is necessary to transform it into an SQL query first. It is surprising that such a tool, that would do the transformation, does not exist yet. The goal of this work is to explore the possibilities of the transformation.

To accomplish this, the relational algebra operations and rules needs to be explained first. After providing the theoratical basis of relational algebra, a new applicable syntax is defined. This work is concerned with a program tool to enforce the syntax and possibly ensure that even the semantic values of the expression are used properly.

The main objective of the work remains. It is to find a way to transform relational algebra to SQL. Then, the transformation process needs to be explained and described. Once the transformation is done, the program needs to be able to connect to the database and run the query. The last part of this work is obtaining the results and displaying them.

1.2 Relational Model

To be able to define a relation, we first need to provide definitions for domain and attribute. Let D_1, D_2, ..., D_n ($n < 0$) be n domains, not necessarily distinct. We can refer to a **domain** as a set of values of similar type. The domain is called simple if all of its values are atomic. For each tuple component, we define its domain and its distinct name, which is called an **attribute**.

The Cartesian product is the set of all n-tuples $\langle t_1, t_2, ..., t_n \rangle$ such that $t_i \in D_i$ for all i. A **relation** on R is defined on these n domains if it is a subset of this Cartesian product. A relation of this nature is said to be of degree n [5].

In a relational model of databases, we can imagine a relation as a table. More formally, the relation is a set of tuples, where each tuple needs to have the same set of attributes. If the domains are simple, the tabular representation of the relation has the following properties [5]:

1. There is no duplication in rows (tuples).
2. Row order is insignificant.
3. Column (attribute) order is insignificant.
4. All table entries are atomic values.

Chapter 2

Relational Algebra

Relational algebra is a procedural query language with five **fundamental operations** [15]. These operations include project, select, Cartesian product, union, and set difference. Besides, several **additional operations** like intersection, theta join, natural join, outer joins and division can be defined. Sometimes rename is also mentioned as an auxiliary operator. In this work, rename will be handled as a fundamental operation, the reason for this will be explained later.

All of relational algebra operators operate on operands that are relations. These operators produce a new relation as their result. Sequences of operators can be used to express complex queries. In one of his definitions of relational model, E.F. Codd emphasized that the algebraic operators are just as much a part of the model as are the structures [5].

For operations in relational algebra, a relation is viewed as a set of n-tuples. The operations of relational algebra are, therefore, the operations of set theory with additional operators which take into account the specific nature of relations [12].

2.1 Relational Algebra Fundamental Operations

2.1.1 Projection

Definition

Projection is a unary operation denoted by the Greek letter pi (Π). Intuitively, a projection of a relation onto a subset of its attributes is a relation formed by omitting some of the columns. More formally, the project operation copies its argument relation, but certain columns are left out. We list the attributes we want to appear in the result as a subscript to Π. The argument relation follows Π in parentheses.

$$\Pi_{\text{comma separated list of attributes}}(relation)$$

Note, that if the projection produces two identical rows, the duplicate rows must be removed since the relation is a set and it is not allowed to contain identical tuples.

Example

Suppose, we have a binary relation *student* containing two attributes *first_name* and *last_name*. This relation contains six tuples.

first_name	last_name
John	Smith
Sarah	Miller
Joe	Johnson
David	Wolfe
Amanda	Bennett
Joel	Smith

Table 2.1: Relation *student*

Now, we apply the project operator to the relation above as follows:

$$\Pi_{last\_name}(student)$$

The attribute *first_name* is not listed in Π subscript, so it is ignored. As a result, we obtain a unary relation with only a *last_name* attribute. Note that there are two students with the last name Smith in the original relation. The only piece of information that tells us that these are distinct students is their first name. When we ignore the first name, we need to remove the duplicate tuple and therefore the resulting relation only has five tuples.

last_name
Smith
Miller
Johnson
Wolfe
Bennett

Table 2.2: Result of $\Pi_{last\_name}(student)$

2.1.2 Selection

Definition

Selection is a unary operation that selects tuples that satisfy a given predicate. Just as the projection selects only a subset of attributes, the selection selects a subset of tuples. The lowercase Greek letter sigma (σ) is used to denote selection. The selection condition appears as a subscript to σ. The argument relation is given in parenthesis following the σ.

$$\sigma_{\text{selection condition}}(relation)$$

The selection condition or selection criterion can be any legally formed expression that involves [14]:

1. Constants (i.e., members of any attribute domain)
2. Attribute names
3. Arithmetic comparisons ($=, \neq, <, \leq, >, \geq$)
4. Logical operators (and, or, not)

Example

Let us have the student relation from the previous example.

first_name	last_name
John	Smith
Sarah	Miller
Joe	Johnson
David	Wolfe
Amanda	Bennett
Joel	Smith

Table 2.3: Relation *student*

Now, we apply the select operator to the relation above as follows:

$$\sigma_{last\_name=\text{`}Smith\text{'}}(student)$$

The result is a binary relation listing all the students with the last name Smith. Note that the first name is not ignored in this case, therefore, we can distinguish between two students in the result and the final relation contains two tuples.

first_name	last_name
John	Smith
Joel	Smith

Table 2.4: Result of $\sigma_{last\_name=\text{`}Smith\text{'}}(student)$

Suppose we have *student* relation with the following scheme:
student = (*first_name*, *last_name*, *age*, *teacher_name*)
Then the expression $\sigma_{age>15}(student)$ selects all tuples in which the student is older than 15 years.

The expression $\sigma_{last\_name=\text{`}Smith\text{'} \wedge age>15}(student)$ returns tuples pertaining to students older than 15 years with the last name Smith.

The last example $\sigma_{last\_name=teacher\_name}(student)$ results in all students who have the same name as their teacher.

2.1.3 Cartesian Product

Definition

We already used the **Cartesian product** to define a relation. Let us have a closer look at this operation. So far, we have not been able to combine information from several relations. The Cartesian product is the first binary operation and it is denoted by a cross ($\times$). The Cartesian product of two relations, r_1 and r_2, is written in infix notation as $r_1 \times r_2$.

To define the final relation scheme, we need to use **fully qualified attribute names**. Practically, it means we attach the name of the original relation in front of the attribute. This way we can distinguish r_1.A from r_2.A. If $r_1(A_1, ..., A_n)$ and $r_2(A_1, ..., A_n)$ are relations, then the Cartesian product $r_1 \times r_2$ is a relation with a scheme containing all fully qualified attribute names from r_1 and r_2: ($r_1.A_1$, ..., $r_1.A_n$,$r_2.A_1$, ..., $r_2.A_n$).

The tuples of Cartesian product are formed by combining each possible pair of tuples: one from the r_1 relation and one from the r_2 relation. If there are n_1 tuples in r_1 and n_2 tuples in r_2, then there are $n_1 n_2$ tuples in their Cartesian product.

Example

Assume we have two relations, *student* and *subject*, as follows:

first_name	last_name
John	Smith
Sarah	Miller

name
Math
IT
English

Table 2.5: Relations *student* and *class*

Now we are going to use Cartesian product operator on those relations to obtain our result.

$$student \times\ class$$

This expression results a relation whose scheme is a concatenation of student scheme and subject scheme. In this case, there are no identical attribute names. For that reason, we do not need to use the fully qualified attribute names. *Student* relation contained 2 tuples, *subject* relation 4, therefore, the result has 8 tuples (2 times 4).

Note that the Cartesian product contains no more information than its components contain together, however, the Cartesian product consumes much more memory than the two original relations consume together. These are two good reasons why Cartesian product should be de-emphasized, and used primarily for explanatory or conceptual purposes [4]. In practice, it can usually be replaced by the natural join operator which will be described later in this chapter.

first_name	last_name	name
John	Smith	Math
John	Smith	IT
John	Smith	English
Sarah	Miller	Math
Sarah	Miller	IT
Sarah	Miller	English

Table 2.6: Result of $student \times\ class$

2.1.4 Union

Definition

The binary operation **union** is denoted, as in set theory, by $\cup$. Union is intended to bring together all of the facts from its arguments, however, the relational union operator is intentionally not as general as the union operator in mathematics [4]. We cannot allow for an example that shows union of a binary and a ternary relation, because the result of such union is not a relation.

Formally, we must ensure that union is applied to two **union compatible** relations. Therefore, for a union operation $r_1 \cup r_2$ to be legal, we require that two conditions are held [15]:

1. The relations r_1 and r_2 are of the same arity. Which means, they have the same numbers of attributes.
2. The domains of the ith attribute of r_1 and the ith attribute of r_2 are the same.

Example

In the following example we have two relations, *student* and *teacher*. They are both of the same arity and their attributes are from the same domain. Therefore, these relations are union compatible and we can apply the union operation to them.

first_name	last_name
John	Smith
Sarah	Miller
Joe	Johnson
David	Wolfe
Amanda	Bennett
Joel	Smith

first_name	last_name
Rebekah	Wolfe
Ashley	Rhodes
John	Smith
Joe	Johnson

Table 2.7: Relations *student* and *teacher*

Following table shows the result of the union operation:

$$student \cup\ teacher$$

The result of the query are all the people (students and teachers) appearing in either or both of the two relations. Again, since relations are sets, duplicate values are dropped.

first_name	last_name
John	Smith
Sarah	Miller
Joe	Johnson
David	Wolfe
Amanda	Bennett
Joel	Smith
Rebekah	Wolfe
Ashley	Rhodes

Table 2.8: Result of $student \cup teacher$

2.1.5 Set Difference

Definition

The last fundamental operation we need to introduce is **set difference**. The set difference, denoted by –, is a binary operator. To apply this operator to two relations, it is required for them to be union compatible. The result of the expression $r_1 - r_2$, is a relation obtained by including all tuples from r_1 that do not appear in r_2. Of course, the resulting relation contains no duplicate tuples.

Example

Let us have the same relations, *student* and *teacher*, from the previous example. We need to remind ourselves that these relations are union compatible, so applying set difference to them is legal.

first_name	last_name
John	Smith
Sarah	Miller
Joe	Johnson
David	Wolfe
Amanda	Bennett
Joel	Smith

first_name	last_name
Rebekah	Wolfe
Ashley	Rhodes
John	Smith
Joe	Johnson

Table 2.9: Relations *student* and *teacher*

Now, we apply the set difference operator to them as follows:

$$student - teacher$$

As a result, we obtain a relation that contains tuples with students that are not also teachers.

first_name	last_name
Sarah	Miller
David	Wolfe
Amanda	Bennett
Joel	Smith

Table 2.10: Result of $student - teacher$

2.1.6 Rename

Definition

In the list of fundamental operators, **rename** is sometimes omitted or it is referred to as an auxiliary operator. For purposes of this work, rename is considered to be a fundamental operator. It is cruicial to create some expressions such as self joins.

Rename is a unary operation denoted by lowercase Greek letter rho (ρ). The result of applying the rename operator to a relation is a relation identical to the original except that the relation and its attributes are given new names. The new names for a relation and its attributes appear as a subscript to ρ, where the new relation name is listed first followed by comma separeted list of new attribute names in parentheses. Let us have a relation r_1 with attributes a_1, ..., a_n, then we can apply the rename operator as follows:

$$\rho_{r_2(b_1,\ldots,b_n)}(r_1)$$

Note, that the number of attributes listed in the subscript must be the same as the number of attributes in the original relation.

Rename operator can be used in a simplified version that allows to rename only the relation, while the attribute names stay the same. The notation for this rename is similar to the previous one. The only difference is omitting the attribute names as follows:

$$\rho_{r_2}(r_1)$$

Example

We are now going to use the relation, *student* from previous examples.

We will now apply a rename operator to the *student* relation. We try to rename the relation to *new_student* with attributes *fname* and *lname*. To do this, we will use the next expression.

$$\rho_{new\_student(fname,lname)}(student)$$

first_name	last_name
John	Smith
Sarah	Miller
Joe	Johnson
David	Wolfe
Amanda	Bennett
Joel	Smith

Table 2.11: Relation *student*

It is important that the rename operation does not affect the tuples. The content of the relation also remains unchanged. The result is a relation called *new_student* that looks as follows:

fname	lname
John	Smith
Sarah	Miller
Joe	Johnson
David	Wolfe
Amanda	Bennett
Joel	Smith

Table 2.12: Result of $\rho_{new\_student(fname,lname)}(student)$

2.1.7 Construction of Expressions

The five basic operators give relational algebra the power to formulate complex queries. All the operators can be considered as functions mapping one or two relation(s) into another relation. The following rules define the relational algebra [1]:

1. If r is a relation, then it is an algebraic expression.
2. If E_1 and E_2 are algebraic expressions, then so are $E_1 \cup E_2$, $E_1 - E_2$ and $E_1 \times E_2$. For union and set difference to apply, E_1 and E_2 must be union compatible.
3. If E is an algebraic expression, then so are $\Pi_L(E)$, $\sigma_P(E)$ and $\rho_R(E)$, where L is a list of attribute names contained in the scheme of E, P is a predicate on the attributes on E and R is a list of new names.
4. If E is an expression, so is (E).

2.2 Relational Algebra Additional Operations

We have now introduced the six fundamental operations of the relational algebra: σ, Π, $\times$, $\cup$, $-$ and ρ. With these six operators we have the power to express any

relational algebra query. However, some commonly used queries become very long when we are restricted to these operations. This is why we are going to define some additional operations in this section.

Note that additional operators do not increase the power of relational algebra. Anything that can be expressed using additional operators can be expressed by fundamental operators as well. The only reason we are introducing them is to simplify common queries.

All the following additional operators are defined in the terms of fundamental operations.

2.2.1 Intersection

Definition

The first operation we add to relational algebra is **intersection**, which is a binary operation denoted by $\cap$. Intersection is not considered a fundamental operation because it can be easily expressed using a pair of set difference operators. Therefore, we require the input relations to be union compatible. Then we can rewrite the intersection as follows:

$$r_1 \cap r_2 = r_1 - (r_1 - r_2)$$

After applying the intersection operator, we obtain a relation containing only those tuples from r_1 which also appear as tuples in r_2. We do not need to eliminate duplicate rows because the resulting relation cannot contain any (since neither of the operands contain any).

2.2.2 Theta-Join

Definition

The second additional operation we are going to define is the **theta-join**. Again, it is a binary operation and it is denoted by $\bowtie_\Theta$, where $\bowtie$ is the join symbol and the Θ symbol in the subscript is the Greek letter theta, which is replaced by the selection predicate. Theta-join is especially useful to simplify Cartesian product expression. Usually, a query with a Cartesian product includes a selection that is applied to the result of the Cartesian product.

Theta-join allows us to combine the selection and Cartesian product into one operation. It forms the Cartesian product of its two arguments and then performs a selection using the predicate Θ [15].

$$r_1 \bowtie_\Theta r_2 = \sigma_\Theta(r_1 \times r_2)$$

The resulting relation scheme is a concatenation of the original schemes. Note that the schemes might have some attribute names in common. In that case we need to use fully qualified attribute names.

2.2.3 Natural Join

Definition

Another additional operation we are going to introduce is the **natural join**. It is a binary operation, denoted by $\bowtie$. Natural join allows us to combine relations that have some common attributes without specifying any detail. It is based on the equality of all common attributes.

Consider two relations $r_1(R_1)$ and $r_2(R_2)$. The natural join of r_1 and r_2 is a relation on scheme $R_1 \cup R_2$. It is the projection onto $R_1 \cup R_2$ of a theta join where the predicate requires r_1.A = r_2.A for each attribute A in $R_1 \cap R_2$ [15]. Formally,

$$r_1 \bowtie r_2 = \Pi_{R_1 \cup R_2}(r_1 \bowtie_{r_1.A_1 = r_2.A_1 \wedge \ldots \wedge r_1.A_n = r_2.A_n} r_2)$$

2.2.4 Outer Joins

Definition

The last additional operation we need to define is the binary operation **outer join**. An outer join is similar to formerly introduced natural join. The only difference is that outer join retains the information that would have been lost from the relations by replacing missing data with nulls.

There are two types of outer joins, left and right. Left outer join is denoted by $\supset\!\!\bowtie$. The result of the left outer join applied to relations r_1 and r_2 in this order is the set of all combinations of tuples in r_1 and r_2 that are equal on their common attribute names, in addition to tuples in r_1 that have no matching tuples in r_2. {(null, ..., null)} is a singleton relation on the attributes that are unique to the relation r_2. Formally,

$$r_1 \supset\!\!\bowtie r_2 = (r_1 \bowtie r_2) \cup ((r_1 - \Pi_{r_1.A_1,\ldots,r_1.A_n}(r_1 \bowtie r_2)) \times \{(null, ..., null)\})$$

The right outer join is denoted by $\bowtie\!\!\sqsubset$ and it is analogous to left outer join, except it keeps data from the right-hand relation. Formally,

$$r_1 \bowtie\!\!\sqsubset r_2 = (r_1 \bowtie r_2) \cup ((r_2 - \Pi_{r_2.A_1,\ldots,r_2.A_n}(r_1 \bowtie r_2)) \times \{(null, ..., null)\})$$

Chapter 3

Expression Analysis

The goal of this chapter is to introduce techniques for relational algebra expression analysis. When we are given an expression, we need to decide whether it is correct, i.e. if it follows the rules defined for the language. Our language is made of all correctly formed relational algebra expressions. One expression can be considered as a sentence of the language.

To describe the relational algebra language, we use a **formal grammar** (or simply a grammar) which describes all sentences that are syntactically correct within given language. Once we define a grammar, we can use it as a basis for a recognizer. The recognizer can determine whether a given sentence is part of the language or not. The process of recognizing a part of language using a grammar is called **parsing**. The actual parser implementation in this work is done in Perl, a high-level, dynamic programming language.

3.1 Formal Grammars

In order to define a grammar, we first of all need to define a couple of terms. **Terminals** are elements with which the sentence of a language may be constructed. It means that terminals cannot be made any more specific. On the other hand, **non-terminals** are elements which are only used in the derivation of a sentence; they never occur as such in the sentences of language [13]. In practice it means they have to be replaced by even more specific components, either terminals or other non-terminals.

Production rules are ordered 2-tuples of strings. They have a non-empty set of terminals and non-terminals on the left side followed by arbitrary set of non-terminal and terminal elements on the right side [13]. The meaning is following; the symbols from the left side of the rule can be replaced by the symbols on the right side. The **start symbol** is from the set of non-terminals and represents the point at which sentence generation will begin.

A grammar is then a system that consists of a finite number of terminals, a finite number of non-terminals, and a finite number of production (grammar) rules [9].

Furthermore, a start symbol must always be specified in order for a grammar to be fully defined.

Grammars can be further classified. A scheme for the classification of grammars that is now in general use is the **Chomsky hierarchy** [13]. For purposes of this work, we need to introduce context-free grammars. **Context-free grammars**, also called Type 2 grammars, have the restriction of being able to have only a single non-terminal on their left side [9]. The significance of using a grammar in this format is that since it is context-free, each non-terminal is able to operate independently of its neighbor. Therefore, the context-free grammars can be implemented in computing using a non-deterministic pushdown automaton [13].

3.2 Relational Algebra Grammar

This section introduces relational algebra grammar. The grammar as shown below is a context-free grammar, thus it can be used in our parser. In Perl notation, the semicolon (:) stands for "can be replaced by". In this particular notation, tokens that are lowercase are non-terminals, and tokens that are uppercase are terminals. Several rules can have the same non-terminal on the left side, the vertical bar (|) is a shorthand for these and it means there are more possibilities for the same non-terminal. The production rule is terminated by a semicolon (;). Since the start symbol is not specified, Perl considers the first non-terminal in the grammar a starting point.

```
exp :
     exp ‘INTERSECTION’ exp
  |  exp ‘UNION’ exp
  |  exp ‘SET_DIFFERENCE’ exp
  |  exp ‘NATURAL_JOIN’ exp
  |  exp ‘LEFT_JOIN’ exp
  |  exp ‘RIGHT_JOIN’ exp
  |  exp ‘CARTESIAN_PRODUCT’ exp
  |  ‘SELECTION’‘[’condlist‘]’‘(’exp‘)’
  |  ‘PROJECTION’‘[‘attrlist’]’‘(’exp‘)’
  |  ‘RENAME’‘[’name‘(’attrlist‘)’‘]’‘(’exp‘)’
  |  ‘RENAME’‘[’name‘]’‘(’exp‘)’
  |  ‘(’exp‘)’
  |  relation
;

attrlist :     attribute
           |  attribute ‘,’ attrlist
;

condlist :     condlist ‘OR’ condlist
```

```
            |  condlist 'AND' condlist
            |  'NOT' condlist
            |  '('condlist')'
            |  compared comp compared
;

comp :          '<>' | '!=' | '=' | '=='
            |   '>=' | '<=' | '<' | '>';
compared :      attribute | data;
relation :      NAME;
attribute :     NAME | fullname;
fullname :      NAME '.' NAME;
data :          NUMBER | STRING_VALUE;
```

Note that this grammar describes relational algebra expressions using a syntax different from the standard syntax introduced earlier. It will be described in the upcoming section that it was necessary to use non-standard syntax in order to simplify typing the expressions.

Since an empty expression makes no sense, it is not allowed by the grammar. Therefore, the simplest expression you can write is just a name of a relation. In that case, the *exp* non-terminal is replaced by *relation* non-terminal, which is then replaced by a particular relation name. The result of such an expression would contain all the tuples from the relation.

The grammar allows the use of different relational algebra operators. The typical infix notation is used for binary operators, where the operator is written in between its operands. For unary operators a prefix notation is used, where the operator precedes the operand. The grammar next defines all possible comparison symbols and describes how the attribute list for projection and the condition list for selection should look like.

The grammar also allows to put parenthesis around any expression. This can be used to increase the readability of the expression or to change the default precedence of some operators. The default precedence of relational algebra will be discussed later in this work as well as detailed syntax of relational algebra expressions as it is accepted by this grammar.

3.3 Parsing Fundamentals

3.3.1 Parsers

When performing parsing tasks, we start with the resultant sentence and a grammar. The parser is then responsible for generating a production graph. The production graph represents how the sentence could have been originally generated. This process is called semantic analysis [11].

Remember that we are using a context-free grammar. This yields the benefit that the production graph produced will always be in a tree form [9]. Hence, it can be referred to as a **production tree**.

If the parser constructs the production tree, we can say that the sentence is proper, since the production tree is only constructible for sentences that follow the rules given in the grammar. Let us now have a closer look at how the parsers function.

For context-free grammars, there are two main approaches and therefore two main categories of parsers: bottom-up and top-down. **Bottom-up** parsers start with terminals and they try to work backward to obtain the most abstract non-terminal [9]. Bottom-up parsers are commonly used in compilers. They are normally constructed using parser generators. One of them, Yapp [6], is introduced in one of the following sections.

3.3.2 Lexers

For a better understanding, the whole parsing process can be divided into two smaller tasks. Before we can decide how the words in a sentence were put together, we need to recognize these words. We start with a sequence of characters that needs to observed and analyzed. In that sequence, the tokens have to be found. The **tokens** are the smallest parts of an input that have a definite meaning [7].

The process of dividing the characters into tokens is known as a **lexical analysis** or just **lexing**. Therefore, the program or part of a program that does it is called a **lexer**. The lexer generates a sequence of tokens to be consumed by some later part of the parsing [11]. It is responsible for identifying tokens and making some form of logical sense for them.

Using some kind of iterator is a natural way to represent the lexing process. The iterator keeps a buffer with the characters seen so far. Each new character is appended to the buffer until there is a whole terminator string. The buffer is then emptied and its contents are returned to the caller.

3.4 Using Yapp

Yapp stands for Yet Another Perl Parser and it is a Perl port of the common Unix parsing utility Yacc (which stands for Yet Another Compiler Compiler) [6].

When we decide to use the Yapp parser generator, we must first create a grammar file that serves as the basis for parser generation. The typical Yapp grammar file consists of three sections, separated by %%:

```
header section
%%
rules section
%%
footer section
```

The header section can optionally contain precedence declarations specifying associativity, followed by the list of tokens or literals having the same precedence and associativity.

The rule section contains the grammar rules. The notation and the grammar used for relational algebra were introduced formerly. In addition, every right hand side symbol can be followed by an optional **semantic action** code block. Semantic actions are performed every time the production rule is used. They can contain any valid Perl code and they are handled as subroutines. The tokens from the right hand side rule are passed to them as the arguments and their semantic value become their return value.

The footer section may contain any valid Perl code and will be appended at the very end of the parser module. Since Yapp does not provide integrated lexing ability, it becomes necessary to supply a lexing routine. The footer section is the place where the lexing subroutine can be put together with error report subroutines and anything relevant to the parser.

3.5 Applicable Syntax For Relational Algebra

It was indicated earlier in the work that the standard relational algebra syntax is very impractical. It is hard to type Greek letters, subscripts and other special symbols on a standard keyboard. Therefore, we need to define a new syntax in order to simply type expressions using a default English keyboard. The newly introduced syntax needs to keep expressions simple to read plus it must be easy to remember and to type.

The notation basics are not changed at all. The binary operators apply infix notation, where the operator is put in between its operands. The prefix notation is used for unary operators which means the operator goes first, followed by its operand. Anything that normally appears in subscript is now to be placed in square brackets. Parentheses stay a way to nest expressions.

We can find a similarity between some relational algebra operations and standard mathematical operations. Union is analogical to addition (+), since it puts together its arguments. We can find the same analogy between set difference and subtraction (−). Cartesian product has some parallels in multiplication (*).

Sigma is replaced by a dollar sign ($). The selection condition folows the dollar sign in brackets and it can contain any non-empty logical expression. The logical expression is formed by comparisons that are connected by logical operators **or** and **and**. The **not** operator can be used as a negation. The notation established in several programming languages is used for logical operators. One or two ampersands (& or &&) act as and, one or two horizontal lines (| or ||) act as or, exclamation mark (!) is used for negation. The syntax allows comparison between any attributes, numbers or string values. Strings have to be enclosed in single or double quotes. Traditional comparison symbols are used, the syntax allows anything from this list: $=, ==, <, >, >=, <=, <>, != $. First two symbols have exactly the same meaning

as well as last two symbols.

Pi is replaced by the hash symbol that is sometimes referred to as a pound symbol (#). It is followed by a non-empty list of comma separated attributes enclosed in brackets. The percent sign makes up for rho. In the case of relational rename, the new name for relation comes after, enclosed in brackets. If we want to rename both the relation and its attributes, we can put a list of attributes right after the new relation name. The number of attributes in the list has to be exactly the same as the number of actual attributes in the relation.

Intersection is accomplished by using carrot sign (^) between two relations. For a natural join we use an at symbol (@). Left and right joins are achieved by placing the tilde ($\sim$) on the proper side of at sign.

The applicable syntax for relational algebra is summarized in the following table:

Operator	RA	Proposed Syntax
Union	$\cup$	$r_1 + r_2$
Intersection	$\cap$	r_1 ^ r_2
Set difference	$-$	$r_1 - r_2$
Natural join	$\bowtie$	r_1 @ r_2
Left join	$\rhd\!\bowtie$	r_1 $\sim$ @ r_2
Right join	$\bowtie\!\lhd$	r_1 @ $\sim$ r_2
Cartesian product	$\times$	$r_1 * r_2$
Selection	σ	\$ [condition] (r_1)
Projection	π	# [list of attributes] (r_1)
Relation Rename	ρ	% [r_2] (r_1)
Relation and Attributes Rename	ρ	% [r_2 (list of attributes)] (r_1)
And	&	$cond_1$ & $cond_2$ or $cond_1$ && $cond_2$
Or	|	$cond_1$ | $cond_2$ or $cond_1$ || $cond_2$
Not	!	!$cond_1$

Table 3.1: Applicable syntax for relational algebra

3.5.1 Precedence

In addition to the sytax, we need to be familiar with the precedence. The precedence rules tell us how an expression with multiple operators is evaluated, i.e. which operators are evaluated first. Let us consider following expression.

$$table_1 \cap table_2 \bowtie table_3$$

Precedence is a simple ordering, based on either the importance or sequence. In relational algebra, the unary operators Π, σ and ρ have the highest precedence. Next highest is the Cartesian product operator $\times$ followed by the join operators $\bowtie$, $\rhd\!\bowtie$ and $\bowtie\!\lhd$. The set difference operator $-$ is evaluated next. Finally are the operators $\cup$ and $\cap$. This is the default precedence.

It means that in our expression, the $\bowtie$ operator will be first applied to tables $table_2$ and $table_3$. Then the result will be intersected with $table_1$. If we want the intersection applied first, we need to change the basic precedence by putting parentheses around it. Whatever is in parentheses will be then evaluated first. Our expression now looks like follows.

$$(table_1 \cap table_2) \bowtie table_3$$

The following table shows the default precedence for relational algebra. The operators at the top of this list are evaluated first. Operators within a group have the same precedence. All operators have left-to-right associativity.

The highest precedence	Projection, Selection, Rename
	Cartesian product
	Natural join, Left join, Right join
	Set difference
The lowest precedence	Union, Intersection

Table 3.2: Table showing precedence of algebraic operators

Besides relational algebra operators precedence, we need to provide precedence for logical operators. Logical operators can be used in selection condition where they connect different comparisons. The precedence for logical operators is demonstrated in the following table. Again, all operators have left-to-right associativity.

The highest precedence	Not
	And
The lowest precedence	Or

Table 3.3: Table showing precedence of logical operators

3.6 Syntax Analysis

Once we have relational algebra syntax defined, we need to come up with a tool that can check it. The previous sections indicate that we are going to use Perl parsing module Yapp. In the header section, the precedences are defined that correspond to introduced principles. The rules section contains the relational algebra grammar.

In addition, the semantic actions are added. Whenever the production rule for a relational algebra operator is used, the semantic action block creates a new node in a parsing tree. The node contains information about the operator. For the intersection operator, the node type 'intersection' is stored and two child nodes are created, each of them including the relational algebra expression that the intersection is being applied to. The $_[n] refference in the sample code listed below refers to

the n-th element of the production rule. When the first element of the production rule is a particular relation, a node with type 'relation' and name of the relation is created and appended. The relation node has no more child nodes. The leaf node of the parsing tree can be either a relation, or information that would normally appear in an operator subscript.

```
exp 'INTERSECTION' exp {
    my %node;
    $node{type} = "intersection";
    $node{left} = $_[1];
    $node{right} = $_[3];
    return \%node;
}
```

When the parser gets a syntactically incorrect expression as an input, it throws an error. The unrecognizable part of the expression is returned as part of the error message, indicating where the error occured and where the mistake in the expression is.

3.6.1 Parsing Tree Example

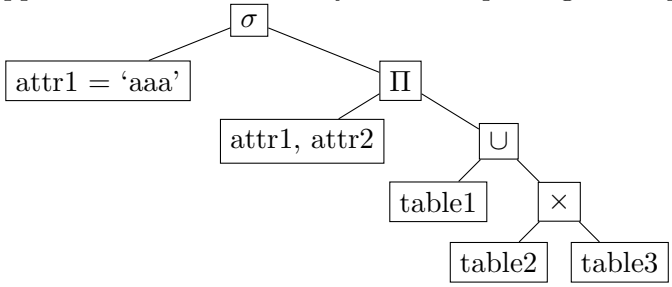
Let us now have a closer look at what the parsing tree looks like. We are going to examine a generic relational algebra expression:

$$\sigma_{attr1='aaa'}(\Pi_{attr1,attr2}(table1 \cup table2 \times table3))$$

To run the expression through the parser, we need to use the newly defined syntax that makes the expression look like the following:

$$\$[attr1 = 'aaa'](\#[attr1, attr2](table1 + table2 * table3))$$

We know that the selection is applied first, followed by the projection. Next, the Cartesian product is applied because it has higher precedence than the union which is applied last. That is exactly what the parsing tree represents.



Note that the selection condition and the projection list appear as child nodes of selection, respectively projection. These leaves are going to be discussed further in the upcoming section, where we are going to perform semantic analysis on them.

To demonstrate the error subroutine ability, we can try to omit the closing bracket of the projection list. Parsing such expression ends with a syntax error.

$$\$[attr1 = \text{`}aaa\text{'}](\#[attr1, attr2(table1 + table2 * table3))$$

```
Syntax error. Data:
table1+table2*table3))
```

3.7 Semantic Analysis

Inseparable part of expression analysis is the semantic analysis. The syntax analysis guarantees that the expression is formed correctly, i.e. the brackets and parentheses are correctly nested and the operators are correctly applied to their operands. But we also need to make sure that all the relations and attributes stated in the expression really exist.

This is when the underlying relational structure comes into play. We need to know what relations are available and we need to know their schemes. We are going to transform the expression into SQL query and run it on a database. Therefore, we are interested in the database scheme. In next chapter we are going to introduce PostgreSQL database [8]. We can obtain table names and attributes in a particular table from the database by running following select queries.

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema
  NOT IN ('pg_catalog', 'information_schema');

SELECT column_name
FROM information_schema.columns
WHERE table_name = 'table_name';
```

The semantic control is a recursive funcion. It walks through the parsing tree starting in the leaves that have type of relation. For each relation it checks whether the relation of a given name exists in a database. If the relation is not found, an error occurs.

```
Semantic Error: Relation 'table_name' does not exist.
```

When the relation exists, another query is sent to the database looking for the attribute names. These are stored internally for further use. As the subroutine walks through the parsing tree, it stores the current scheme. In every node, it knows which relations and attributes are available for use. Whenever a selection or projection node is encountered, a check subroutine is invoked. The check subroutine gets an attribute list or a selection condition as a parameter. It finds all the attributes and

runs a check. For each attribute it checks whether it can be used in a given context. If the full name is used for the attribute, it also checks whether the relation exists.

The semantic control is simplified and does not work correctly in all cases. When a projection is applied, reducing the count of attributes, those attributes are not deleted from current scheme, because the selection can still refer to those attributes. When another projection is then applied, listing some attributes that were already left out, the semantic check does not end with an error.

It is important to know why we can afford recognizing only some semantic mistakes. The expression is going to be transformed into SQL query and sent to a database. If there is something wrong with the query, including semantic errors, the database will still tell us. The described semantic analysis attempts to reveal some of these errors, but not all of them. It correctly uncovers the cases when the relation does not exist in the database, however it does not always work correctly with the attributes for the reasons explained above. One might object that the semantic analysis can be omitted from the evaluation process and the responsibility for finding errors should go directly to the database. However, it is a good habit to perform a semantic analysis, and this work attemps to cover all areas of the transforming process.

Chapter 4

Transformation into SQL

4.1 SQL Basics

SQL is a tool for organizing, managing, and retrieving data stored by a relational database. Originally, SQL was an acronym for **Structured Query Language**. SQL defines many database operations including data definition, data retreival, data manipulation and data integrity [10]. Relational algebra is basically a subset of SQL. Therefore, we only need a small part of SQL to express relational algebra queries. The most fundamental concept of SQL is called the query block. In fact, all relational algebra expressions can be transformed into a select block. Its basic form is:

```
SELECT <list of attributes>
FROM <list of tables>
WHERE <qualification expression>
```

The result of the execution of a query block is a table whose structure and contents are determined by that block [2]. The list of attributes specifies attributes that are selected from the tables in the list of tables. The first two clauses (SELECT and FROM) in the query block are therefore analogical to the operation of projection. The qualification expression in the WHERE clause is a logical expression. It can contain attributes of the tables listed in the FROM clause and determines what records of those tables qualify. This operation is analogical to the algebraic operation of projection [2]. Only the records of those tables for which the qualification expression is true will appear in the result of the query block.

4.1.1 Database Management Systems

With the origin of SQL, many **relational database management systems** (RDBMS) were formed based on it. One of many current RDBMS is PostgreSQL [8]. It is an open-source RDBMS and it very closely follows the industry standard for query languages, SQL92. One of its many features are subselects [8]. As we will

find out later, subselects are very important for transforming relational algebra into SQL.

4.2 Using Perl DB Interface

Perl::DBI is a database interface module that provides a consistent database interface. It is independent of the actual database being used and, among many other features, allows to establish the connection to the database, run a query and obtain the result [3].

After we define the database name, host and port, we can then connect to the database using particular username and password. We can then run any queries we want to. We have already introduced some queries related to the semantic control. Those queries were supposed to figure out what the tables in the database are and what the attributes in those tables are.

The following snippet of code shows how the connection information can be set and how the connection to the PostgreSQL is etablished using module Pg, the DBI extension for Postgres.

```
my $dbname = "School";
my $dbhost = "localhost";
my $dbport = "5432";
my $dbuser = "postgres";
my $dbpass = "password";

my $dbh = DBI->connect("dbi:Pg:dbname=$dbname;host=$dbhost;
              port=$dbport","$dbuser", "$dbpass",
              {AutoCommit => 0, PrintError => 1, RaiseError => 0})
   || die "Error: Cannot connect to the database.";
```

4.3 Tree Traversal

We have explained that since relational algebra is a subset of SQL, it is possible to transform any relational algebra expression into an SQL query. The last question that remains unaswered is what the transformation looks like.

First, we need to know how the simplest relational algebra expressions should be translated to the SQL. The simple relational algebra operations and their SQL equivalents are shown below.

Remember that in relational algebra, the relations are sets of tuples, therefore all the tuples are unique. In a result of an SQL query identical rows can appear when we use some operations. However, we are evaluating a relational algebra expression, so we need to ensure that the identical rows are eliminated. That is achieved by using SELECT DISTINCT when doing a projection. Note, that DISTINCT does

Union: r $\cup$ s, r + s

```
SELECT * FROM r
UNION
SELECT * FROM s
```

Intersection: r $\cap$ s, r $\wedge$ s

```
SELECT * FROM r
INTERSECT
SELECT * FROM s
```

Set difference: r $-$ s, r $-$ s

```
SELECT * FROM r
EXCEPT
SELECT * FROM s
```

Natural join: r $\bowtie$ s, r @ s

```
SELECT * FROM
r NATURAL JOIN s
```

Left join: r $\exists\bowtie$ s, r $\sim$@ s

```
SELECT * FROM
r NATURAL LEFT JOIN s
```

Right join: r $\bowtie\sqsubset$ s, r @$\sim$ s

```
SELECT * FROM
r NATURAL RIGHT JOIN s
```

Projection: $\pi_{list\ of\ attributes}$(r),
[list of attributes] (r)

```
SELECT DISTINCT
list of attributes
FROM r
```

Selection: $\sigma_{condition}$(r),
$ [condition] (r)

```
SELECT * FROM r
WHERE condition
```

Cartesian product: r $\times$ s, r $*$ s

```
SELECT * FROM
r, s
```

Relation Rename: ρ_s(r), % [s] (r)

```
SELECT * FROM r AS s
```

Relation and Attributes Rename:
$\rho_{s(list\ of\ attributes)}$(r),
% [s(list of attributes)] (r)

```
SELECT * FROM
r AS s(list of attributes)
```

not need to be used with INTERSECT, UNION and EXCEPT operators, because these operators eliminate identical rows by default.

As the examples show, it is easy to transform a relational algebra expression with a single operator. We now need to demonstrate what happens when the expression contains nested operators. It was implied that PostgreSQL allows for the creation of nested select statements. It makes sense to take advantage of this feature, replace each simple relational operation with a select statement and put all of them together.

To create a nested select statement, we need to traverse the parsing tree in a way that starts with the easiest expressions, i.e. relations, and then wraps those with increasingly more complicated selects. This is similar to the tree traversal explained in the semantic analysis section. Therefore, we just need to adjust that algorithm. For each node of the tree we need to rewrite the operation from relational algebra to SQL; doing this recursively ensures the nesting.

4.4 Potential Problems

Nesting the select expressions seems to be easy; however we have to be careful, because some rules and constraints apply to it. PostgreSQL requires some subselects to be renamed. This is important. Let us now show an example in which the renaming is necessary.

First, we can show an example that works and does not require any additional renaming. It is a simple union of two tables, $r + s$. When it is transformed to an SQL query, it looks like the following:

```
(SELECT * FROM r)
 UNION
(SELECT * FROM s)
```

We can now add a projection operator to the expression and display only the first attribute of the union, $\#[attribute](r + s)$. The natural way to transform this expression to SQL would be wrapping the previous example in another select statement. However, this is one of the cases in which PostgreSQL requires renaming. Before the projection is made from the union statement, the original statement needs to be put into parentheses and provided with an alias.

```
SELECT DISTINCT attribute FROM
( (SELECT * FROM r)
   UNION
  (SELECT * FROM s)
) AS foo
```

Note that the original relational algebra expression is correct and does not need any aliases. However, the transformed expresion without an alias is not correct in our database.

It is easy to generate an alias whenever necessary; we just need to be careful about the consequences. If we now decide to refer to the attribute using its full name, it would need to be *foo.attribute*. The aliasing is done without the user, who has no idea that his relation needed to be renamed. Therefore, some correct expressions inputted by user fail.

This work does not provide a clear solution to this problem. There is only a recommendation for anyone using the program, which is to try to avoid full names. Whenever there is an attribute ambiguity, renaming one of the attributes is better than using full names. Another option might be to put the selection or projection operator close to the relation inside, so that it does not need to use the full name.

4.5 Optimization

The transformation subroutine tries to do at least some basic optimization tasks. The section above discussed the problem with the necessary rename that is required by PostgreSQL. Another option for users would be doing the necessary rename by themselves. The transformation process is optimized, so that it does not add an alias to a relation that has been renamed by the user.

It is quite common, that selection and projection appear following each other. In this case, there is no need to create two nested select statements. The transformation subroutine instead creates one select statement, placing the projected attributes at the beginning and the selection condition at the end of it.

More ways of how the transformation could be optimized exist. These are not implemented in this work, however some of the options are further discussed in the final chapter of this work.

Chapter 5

Simple User Interface

In this chapter we are going to explain how the abilities of our program can be expanded by providing a simple user interface. The interface is based on Perl cgi scripts and can be run in any regular web browser. The main goal is to allow one person, potentially a teacher, to formulate definitions for relational algebra queries. Other users, students, can then try to create defined queries. The responsibility of the interface is to display appropriate responses, informing the user about the correctness or incorrectness of his or her relational algebra expression.

In addition to performing syntax and semantic analysis, we need to decide whether the result of the given expression is correct. When we are looking for all subjects with more than four credits, we need to be able to say whether the result complies with the assignment or not. This can be achieved easily by providing the set of questions with some sort of correct answer. In a database we can create a simple table that contains questions, scores for each question and correct answers. Each question needs to have a unique id. It was said before that SQL is a standard that is commonly used, so the easiest way to store the correct answer is entering the correct SQL query.

It is important to realize that the correct SQL query does not necessarily need to look like the query created from relational algebra expression. We can use the ability of the program to transform relational algebra expression into SQL query, run it on a PostgreSQL database and then compare the results.

We are about to introduce a simple interface that has two modes, practicing and testing. Their characteristics are described later in this chapter. To demonstrate their ability, we will need to show some examples of relational algebra expressions. All the provided examples are based on the following relational scheme. The attributes that form the primary keys of the relations are underlined.

subject = (*subject_code*, *description*, *credits*)
enrollment= (*student_id*, *subject_code*)
sem_group = (*subject_code*, *group_no*, *description*, *capacity*)
student = (*student_id*, *first_name*, *last_name*, *address*)

5.1 Practicing Mode

5.1.1 Error Checking

In the practicing mode, one question at a time is pulled from the database. This question is displayed in a browser with an input field beneath. This input field is designated for the answer. When the user enters his relational algebra expression, he can submit it. The expression is then processed and the results are diplayed on a new page.

In this mode, errors can occur in three different levels. First, the syntax is checked by the parser module. If an error occurs, it is displayed to the user with the data that the parser was unable to process. Imagine we want to select all students whose first name is Joe, but we forget to type the sign of equation and we type in the expression:

$$\$[first\_name`Joe'](student)$$

The program will respond with a syntax error and output the part of the expression that cannot be parsed. In this particular case, the output will look like follows, indicating that the error is inside the brackets:

```
Syntax error. Data: ](student)
```

When the syntax check runs smoothly and the parsing tree is built, the semantic analysis is performed. Two different types of errors can appear within semantic analysis. Let us type almost correct query with the relation *student* mistyped as a *sttudent*:

$$\$[first\_name = `Joe'](sttudent)$$

The semantic control fails in finding a table *sttudent* in the database and displays an error. The evaluating process ends at this point and the query is considered wrong.

```
Semantic Error: Relation sttudent does not exist.
```

It was implied earlier in this work that semantic check for attributes does not work correctly in all cases. Therefore just a warning is displayed when an incorrect attribute is typed in. Here the attribute *first* is used in the expression instead of *first_name*:

$$\$[first = `Joe'](student)$$

The semantic control suspects the mistake, but since it is not correct all the time, only a warning is displayed instead of an error. The warning implies that the attribute might not exist:

```
WARNING: Attribute first might not exist in this context!
```

Note that in this case the evaluation process continues. The relational algebra expression entered by user is transformed to an SQL query and then send to the database. It is now the responsibility of database to determine any errors. We know that the attribute *first* does not really exist and therefore we get an error:

```
Transformed query:
SELECT DISTINCT * FROM student WHERE first = 'Joe'

Database Error: DBD::PgPP::db selectall_arrayref failed:
ERROR: column "first" does not exist at character 38
```

5.1.2 Result Comparison

So far we have shown what happens if the user expression cannot be evaluated at all. Let us now have a closer look at what happens when the expression is transformed into SQL query that can be processed by the database. The database returns a relation that needs to be further examined. This is when the original question and answer definition comes into play.

We know that the correct answer is stored for each question in a form of a select query. All we need to do to obtain correct result is run this query on a database. Then the correct result has to be compared with the result of user query. The results from the database are obtained as two-dimensional arrays. These two arrays, one with correct result and one with user result, are transformed to strings that can be later easily compared.

If the strings are not equal, a short message is displayed saying that the query is wrong. The user is given an option to try the same query again. The example below shows how the output for a correct relational algebra expression looks like. The user has now an option to try next expression.

```
Evaluating the following expression:
$[first_name = 'Joe'](student)
Syntax seems to be OK.
Semantics seems to be OK.

Transformed query:
SELECT * FROM student WHERE first_name = 'Joe'
Database run seems OK.

Congratulation! Your query is correct.
```

5.2 Testing Mode

The testing mode of the program works in a similar manner as the practicing mode. Instead of showing one question at a time, all the questions from the database are pulled and displayed on the same page. An input field for user answer is placed below each question. Once the user submits the test, the result is displayed showing the score. The score is a summation of points obtained for individual questions. No answer or wrong answer are both worth zero points. In this mode, there is no option to take the the test again; the result is final.

Note that in this mode, there is no information displayed implying whether there was syntax error, semantic error or a database error. It is not even said which questions were answered correctly and which were wrong. We are going to show a sample test that contains three questions:

1. Select all students whose first name is Joe (5 points)
2. Select subject code and description of all subjects that have at least one group (7 points)
3. Select subject code and description of all subjects that have at least one group and have an enrollment (10 points)

First expression was correctly answered in the practicing mode, however we are going to leave it empty.

To answer the second question correctly, we need to realize that the *description* attribute appears in both relations *subject* and *sem_group*. If we want to use a natural join and we want it to work correctly, we have to eliminate the *description* attribute in the *sem_group*. To do that, we can project only the *subject_code* from the *sem_group*.

$$\#[subject\_code, description]\ (subject\ @\ \#[subject\_code]\ (sem\_group))$$

Trying this expression in a practicing mode would show us the SQL query and it would also tell us that this is the correct answer.

```
SELECT DISTINCT subject_code, description FROM
(
  SELECT * FROM
    subject
    NATURAL JOIN
    (SELECT DISTINCT subject_code FROM sem_group) AS foo1
) AS foo2
```

Another approach to this expression would be using a rename operator to change the *description* attribute in the *sem_group* to something else. We need to keep in mind that the number of attributes in rename has to be the same as the number of attributes in the relation. We want to keep the *subject_code* attribute, because the natural join uses this one, but we can rename everything else.

$$\begin{aligned}&\#[subject\_code, description]\ (subject\ @\\ &\quad \%[sem\_group\ (subject\_code, a, b, c)]\ (sem\_group))\end{aligned}$$

Again, trying the query in the practicing mode would show us the SQL query and it would also tell us that it is the correct answer.

```
SELECT DISTINCT subject_code, description FROM
(
  SELECT * FROM
    subject
    NATURAL JOIN
    sem_group AS sem_group(subject_code, a, b, c)
) AS foo1
```

The third question in our sample test is similar to the second one, except we need to join more than two tables. We want to obtain all subjects that have at least one group and have an enrollment. Using the knowledge from previous examples, we can only add the join operator and the *enrollment* relation.

$$\#[subject\_code, description]\ (subject\ @ \\ \#[subject\_code]\ (sem\_group)\ @\ enrollment)$$

This would give us slightly more complicated, but still correct query.

```
SELECT DISTINCT subject_code, description FROM
(
  SELECT * FROM
  (
    SELECT * FROM
      subject
      NATURAL JOIN
      (SELECT DISTINCT subject_code FROM sem_group) AS foo1
  ) AS foo2
  NATURAL JOIN
  enrollment
) AS foo3
```

Leaving the first answer empty and submitting the test with the correct answers for the second and third question would show us the following result:

```
Your score is 17.
```

Chapter 6

Conclusions

This work proved that is possible to transform relational algebra to SQL and it showed that the transformation can be done in a few steps. The first step is to define a formal grammar for relational algebra that describes its syntax. Then, the basic parsing tasks need to performed. If the parser manages to create a parsing tree, the inputted expression is syntactically correct and semantic check follows. At this point queries are run that obtain table names and attributes for each table from the database. The parsing tree is then compared to those results which ensures that all the relations and attributes included in the relational algebra expression exist in the database.

To traverse the parsing tree, a recursion is used. The algorithm walks through the tree and transforms each operation into a select query. As a result, it returns one main select query with a couple other subqueries that are nested. It is important to know that our evaluation process is using PostgreSQL; therefore the use of nested selected queries is allowed. There are some constraints on nesting the queries. The main constraint, which is the need for aliases for nested queries, is solved by generating aliases in the transformation phase.

Once the relational algebra expression is transformed into SQL query, the database connection can be established and the database can be queried for the results. The database interface is provided by Perl database interface module. The reasons were given why an error can still occur at this point. When the database returns an error, evaluation of the relational algebra expression fails. When the query is correct, the result is returned by the database. Processing the result is then simple, it is compared to the result of correct SQL query that is defined in the database.

6.1 Optimizations

The algorithm is now able to generate nested selects that accomplish the task set forth, but there is still a lot of work that could be done to optimize this algorithm. There are ways to preprocess the parsing tree to simplify it. For example when

more projections in a row appear, they can be consolidated to one. The same thing is true for selections. Selections and projections have the same effect, no matter in what order they are applied. Therefore, even two projections that have a selection in between can be reduced to one and vice versa. There are other rules that can be applied when simplifing the relational algebra tree. The tree preprocessing would result in simpler SQL statements.

The expressions could be even normalized. Normalization would consist of eliminating needless operations and moving projections and selections closer to their source relations. The selection condition would be normalized by ordering the conditions and eliminating the useless conditions and the projection list would be ordered alphabetically.

There was a need to define a whole new syntax in order to be able to type the expressions. Another option would be to design a graphical user interface that would allow a user to insert all the necessary special symbols into the expression.

One possible way to do that would be a simple toolbar that would list all the special symbols; clicking the symbol would add it to the place where the cursor points. It should also allow for the insertion of symbols by using shortcuts. A part of the interface would be a mechanism to create and correctly display subscripts, that appear in almost every expression multiple times.

Bibliography

[1] Serge Abiteboul, Patrick C. Fischer, and H.-J. Schek. *Nested Relations and Complex Objects in Databases.* Springer, 1989.

[2] Suad Alagi. *Relational Database Technology.* Springer, 1986.

[3] Tim Bunce. DBI. `http://search.cpan.org/~timb/DBI/DBI.pm`, accessed May 2009.

[4] E. F. Codd. *The Relational Model for Database Management: Version 2.* Addison-Wesley, 1990.

[5] F.E. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4:397–434, December 1979.

[6] Francois Desarmenien. Parse::Yapp. `http://search.cpan.org/~fdesar/Parse-Yapp-1.05/lib/Parse/Yapp.pm`, accessed May 2009.

[7] Mark Jason Dominus. *Higher-order Perl: a guide to program transformation.* Elsevier, 2005.

[8] Susan Douglas. *PostgreSQL: a comprehensive guide to building, programming, and administering PostgreSQL databases.* Sams Publishing, 2003.

[9] Christopher M. Frenz. *Pro Perl Parsing.* Apress, 2005.

[10] James R. Groff and Paul N. Weinberg. *SQL, the complete reference.* McGraw-Hill Professional, 2002.

[11] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide.* Springer, 2007.

[12] S. Krishna. *Introduction to Database and Knowledge-based Systems.* World Scientific, 1992.

[13] W. J. M. Levelt. *An Introduction to the Theory of Formal Languages and Automata.* John Benjamins Publishing Company, 2008.

[14] Steven Roman. *Access Database Design and Programming.* O'Reilly, 2002.

[15] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, 1997.