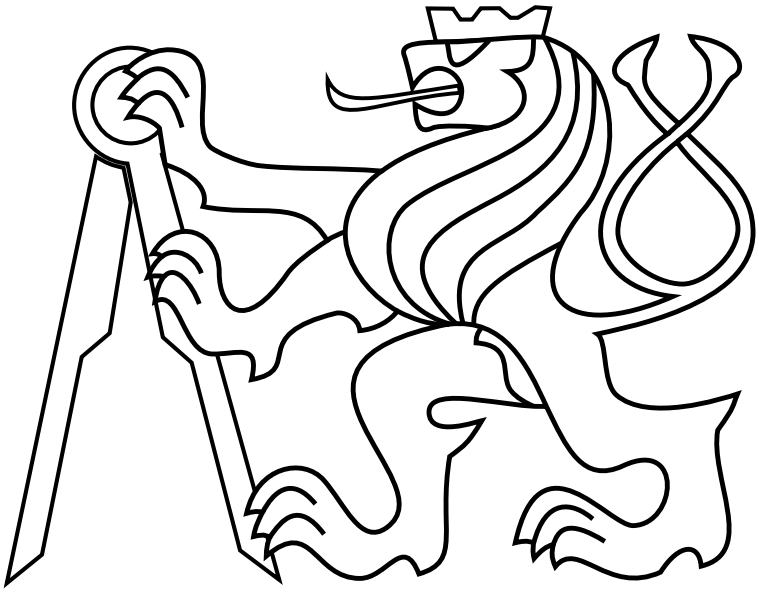
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR'S THESIS



Relational Algebra and Relational Calculus Converter

Tomáš Hauser

Thesis supervisor: **RNDr. Ingrid Nagyová, PhD.**
Department of Computer Science

May 2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Hauser** Jméno: **Tomáš** Osobní číslo: **492159**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Převodník notací relační algebry a relačního kalkulu

Název bakalářské práce anglicky:

Relational algebra and relational calculus converter

Pokyny pro vypracování:

The aim of this work is to analyze the possibilities of conversion of relational algebra (RA) and relational calculus (RC) queries written in different notations, and to design and implement an automatic notation converter.
1. Analyze the operations of relational algebra and the possibilities of their different notation.
2. Familiarize with relational calculus and try to define a converter from or to RA.
3. Familiarize with the possibilities of entering RA (alternatively RC) queries in various online tools and design your own way of entering queries.
4. Implement an input field for entering RA and RC queries. According to the results of the analysis, also implement query converters for selected notations.
5. Define a set of RA and RC queries in different notations and check the correctness of the conversion using the created notation converter. Try to automate the inspection process.

Seznam doporučené literatury:

1. Codd, E. F. A relational model of data for large shared data banks. Communication of the ACM 13, 6, June 1970, pp. 377-387. doi: 10.1145/362384.362685
2. Pichler, R. Database Theory – Codd's theorem. Institute of Logic and Computation, TU Wien, 2018.
3. Svoboda, M. Database systems – Relation algebra. FEL, ČVUT, 2021.
4. de Almeida Cruz, J.J. and de Azevedo Silva, K.K. (2017). Relational Algebra Teaching Support Tool. Journal of Information Systems Engineering & Management, 2(2), 8. doi: 10.20897/jisem.201708
5. RelaX - relational algebra calculator. Universität Innsbruck. https://dbis-uibk.github.io/relax/landing
6. Ranta, A. Query coverter. 2018. https://www.grammaticalframework.org/qconv/

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Ingrid Nagyová, Ph.D. kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **07.02.2022** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **30.09.2023**

RNDr. Ingrid Nagyová, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....

Acknowledgements

I would like to thank my adviser RNDr. Ingrid Nagyová, Ph.D. for her guidance, knowledge, and assistance during the research. I also thank prof. RNDr. Jaroslav Pokorný, CSc. for taking the time and kindly answering all the questions I had.

Abstract

Relational algebra is a formal procedural query language commonly taught in introductory courses into database systems. Due to the complexity of its notation, some schools use a simplified version which is easier for students to understand. The notations are described in a form of a conversion table, and the advantages of the simplified notation are discussed thereupon.
A way of inputting queries implemented by some of the tools that work with relational algebra is analysed.
After constructing formal grammars and describing the parsing and conversion processes, an online application for conversion between the two notations is implemented. The way of inputting queries is based on the mentioned analysis.
Additionally, a conversion from relational algebra into tuple relational calculus, which is a declarative counterpart of relational algebra, is defined and implemented into the application.

Keywords: relational algebra, relational calculus, tuple relational calculus, converter, notation

Abstrakt

Relační algebra je formální procedurální dotazovací jazyk, jenž se běžně učí v úvodních kursech do databázových systémů. Kvůli komplexnosti její notace používají některé školy zjednodušenou verzi, která je pro studenty srozumitelnější. Obě notace jsou popsané formou převodní tabulky, načež navazuje krátká diskuse o výhodách zjednodušené notace.
Dále je analyzován způsob, jakým různé nástroje, které také pracují s relační algebrou, umožňují vkládání dotazů.
Poté, co jsou zkonstruovány formální gramatiky a vysvětlen parsovací a převodní proces, je implementována online aplikace, která dokáže převádět mezi zmíněnými dvěma notacemi. Způsob, jakým aplikace řeší vkládání dotazů je založen na zmíněné analýze.
Dodatečně je také definován a implementován převod z relační algebry do n-ticového relačního kalkulu, což je deklarativní protějšek relační algebry.

Klíčová slova: relační algebra, relační kalkul, n-ticový relační kalkul, převodník, notace

Contents

List of abbreviations

In Table 1 are listed abbreviations used in this thesis.

Abbreviation	Meaning
RA	relational algebra
TRC	tuple relational calculus
AST	abstract syntax tree

Table 1: List of abbreviations

Chapter 1

Introduction

Contents

1.1 Relational Algebra and Relational Model

Relational algebra (RA) is a formal query language that formed the basis of the widely used SQL language. [1] For the purposes of this work, a basic understanding of the SQL language and its operations is assumed.

Just like SQL, RA defines a set of operations that are combined into queries, which are then used to retrieve data from the database. While SQL works with relational databases, RA is a formal language; therefore, it works with a mathematical model of relational databases. This mathematical model is called *relational model*.

The relational model aims to formalize the concept of tables by using set theory. Tables are represented by *relations* which are ordered pairs that contain a set of rows (*tuples*) and an ordered list of column names (*attributes*). The tuples contain the data as ordered pairs of values with the corresponding attributes. [2] For simplicity's sake, we can think of relations as regular database tables with a distinction that there are no identical rows because the tuples are organized in a set. To be terminologically correct, the tables, columns, and rows will henceforth be referred to with the names of their counterparts from the relational model.

1.2 Goals

The main goal of this thesis is to construct an online tool for a conversion between two RA notations. For the purposes of this work, we will call them the *standard* and *simplified* notation. The standard notation is the original notation used in the paper [3] where RA was first introduced. It uses Greek letters and subscripts, which makes it difficult to type queries on a keyboard. As an alternative to that, a simplified notation has been introduced by prof. RNDr. Jaroslav Pokorný, CSc. It uses different syntax and common symbols, thereby making it easier to learn and work with.

The tool will help with a better understanding of the advantages of simplified notation and will also serve as a communication bridge between two parties using different notations. Students who only know the simplified notation can use it to convert their queries into the standard notation. They can then use the result in other tools or in a thesis avoiding having to learn the standard notation and simultaneously getting all the advantages thereof.

The second goal is to look at other RA tools and analyse the way they use to input queries. The implementation of the conversion tool should then be based on this analysis.

Thirdly, after the implementation is finished, it should be tested to ensure the correctness of the conversion.

The last goal is to briefly introduce tuple relational calculus (TRC) and to explore the possibilities of conversion from or into RA.

Chapter 2

Relational Algebra Notations

Contents

2.1 Operations

The table 2.1 showcases RA operations in two notations with a brief explanation of what they do. [4] The first six operations - projection, selection, renaming, Cartesian product, union, and difference are so-called *atomic operations*. [4] Just by using those, we can infer any other operation in the table.

Although most of the operations are relatively simple and have an equivalent in SQL, a division stands out. Let us look at an example. Having a relation of programming languages and a relation of users with programming languages they know, the query 2.1 returns users who know all the languages in the right relation.

user	**language**
john21	Java
carl04	C
peter2	Java
john21	Python
peter2	Python
carl04	PHP
peter2	SQL

$\div$

language
Java
SQL

$=$

user
peter2

(2.1)

Finally, the table 2.2 contains the requirements that some operations have for the input relations. [4]

Name	Standard notation	Simplified notation	Meaning
Projection	$\pi_{a,b}(R)$	$R[a, b]$	Leaves only selected attributes
Selection	$\sigma_{x=y}(R)$	$R(x = y)$	Leaves only tuples satisfying the condition
Renaming	$\rho_{b/a}(R)$	$R\langle a \to b\rangle$	Changes attribute names
Cartesian product	$R \times S$	$R \times S$	Yields all combinations of tuples from R and S
Union	$R \cup S$	$R \cup S$	Performs set union
Difference	$R \setminus S$	$R \setminus S$	Performs set difference
Intersection	$R \cap S$	$R \cap S$	Performs set intersection
Natural join	$R \bowtie S$	$R * S$	Joins R and S based on common attributes
Theta join	$R \bowtie_{x=y} S$	$R[x = y]S$	Joins R and S based on a given condition
Left/right semijoin	$R \ltimes S$ $R \rtimes S$	$R <* S$ $R *> S$	Leaves tuples from the left/right relation that can be naturally joined with the other
Theta semijoin	$R \ltimes_{x=y} S$ $R \rtimes_{x=y} S$	$R\langle x = y]S$ $R[x = y\rangle S$	Semijoin with a condition for the tuples
Left/right antijoin	$R \triangleright S$ $R \triangleleft S$	$R \triangleright S$ $R \triangleleft S$	Leaves tuples from the left/right relation that can't be naturally joined with the other
Left/right/full outer join	$R \rtimes\!\bowtie S$ $R \bowtie\!\ltimes S$ $R \rtimes\!\bowtie\!\ltimes S$	$R *_L S$ $R *_R S$ $R *_F S$	Natural join that fills the tuples from the first/second/both relations that can't be joined with null values
Division	$R \div S$	$R \div S$	Explained above

Table 2.1: Standard and simplified notation with a brief explanation

Operation	Constraint
Projection	The list of attributes is not empty and all of them are present in the relation.
Renaming	No duplicate attributes in the result.
Union, intersection, difference	Both relations have the same number of attributes with the same names.
Cartesian product, theta join and semijoin	Both relations have disjoint attributes.
Division	The left relation is a proper superset of the right one.

Table 2.2: Constraints of RA operations

2.2 Characteristics of the Simplified Notation

As we can see from the table 2.1, the simplified notation mostly uses common symbols that are easy to find on a keyboard. Another advantage is that it uses a postfix notation when selecting, projecting, and renaming which may make reading and creating a query easier.

To illustrate, when reading the following expression

$$\rho_{b/a,d/c}(\pi_{a,c}(\sigma_{a=4}((R \times S) \bowtie_{x=y} (P \cup Q)))), \tag{2.2}$$

we first get to know that we are renaming some attributes, projecting and then selecting without actually knowing what's inside. We have to first evaluate what's inside the parentheses and then we have to come back to these three operations. On the contrary, when reading the same query in the simplified notation

$$((R \times S)[x = y](P \cup Q))(a = 4))[a, c])\langle a \to b, c \to d\rangle, \tag{2.3}$$

the order of operations is closer to the evaluation order.

Nevertheless, one small disadvantage is that the theta join and projection both use the same brackets, so an expression

$$\pi_x(R \bowtie_{x=y} S) \tag{2.4}$$

has an equivalent

$$R[x][x = y]S, \tag{2.5}$$

which may raise concerns whether the simplified notation introduces ambiguity. There is, fortunately, nothing to worry about as theta conditions cannot contain commas and a single attribute name is not a valid theta condition, so they can't be mistaken.

Chapter 3

Tuple Relational Calculus

Contents

3.1 Introduction

TRC is a declarative language that was invented by E. Codd. - the same scientist who introduced RA. [5] Unlike RA, which is a procedural language, TRC is declarative. In other words, in RA we say how to obtain the result, while in the TRC we specify what we want to retrieve without explicitly saying how to do it.

TRC uses the concept of a set builder. This means that instead of listing all elements of a set, we specify a condition that an element has to meet to become a member. [6] This condition is what TRC is about - making a statement that is true only for the tuples we want in the resulting relation. Such a set has a form

$$\{\, t \mid \varphi(t) \,\}, \tag{3.1}$$

where $\varphi(t)$ is the condition to be satisfied. This condition has a form similar to a first-order predicate logic formula. It specifies which tuples we want to retrieve from the schema. The t is called a *tuple variable*. Despite the fact that Codd's paper [5] allows for multiple tuple variables, the vast majority of lecture notes were found to use only one. We will therefore focus on TRC with a single tuple variable, the expressiveness of which, as we will see, is preserved. Moreover, the mental model presented in the following section would not work with multiple tuple variables - it is much easier to only work with one.

A TRC formula can contain predicates in these three forms:

1. Unary predicate $P(t)$ says that t belongs to a relation P.
2. $p.a = q.a$ says that an attribute a of a tuple variable p equals to the attribute a of a tuple variable q.
3. $p.a = x$ says that an attribute a of a tuple variable p is equal to x (a number or a string literal).

We can then join them with logical connectives ($t.a = p.a \wedge t.a > 4$) and quantify them ($\exists z(R(z) \wedge t.a = z.a)$) just like in predicate logic.

3.2 Examples

Let us have a relation called *Elements*:

element	**protons**	**electrons**
neon	10	10
copper	29	35
gold	79	118
oxygen	8	8
carbon	6	6

(3.2)

If we wanted to select all elements with the same number of protons and electrons, we would write

$$\{\, t \mid \text{Elements}(t) \wedge t.\text{protons} = t.\text{electrons} \,\}. \tag{3.3}$$

If the belonging of a tuple variable t is not specified, then only the attributes mentioned with equality to t get into the result. We use this fact to retrieve only a part of a relation - if we only wanted the *element* attribute from the previous query, we would write

$$\{\, t \mid \exists z(\text{Elements}(z) \wedge z.\text{protons} = z.\text{electrons} \wedge t.\text{element} = z.\text{element} \,\}. \tag{3.4}$$

As a mental model, we can imagine taking the condition in the query and going through the whole schema asking at each tuple whether it holds true. If so, we include it in the result.

3.3 Equivalence Between RA and TRC

In the paper [5], Codd proved that TRC is at least as expressive as RA. That, however, does not mean that any TRC query can be converted into RA. Consider a query

$$\{ t \mid \neg R(t) \}. \tag{3.5}$$

If we wanted to write an equivalent query in RA, we would need to somehow involve relations that were not named in the query. And even if we managed to do that, the query would only make sense in terms of that one specific database. Such queries are called *domain-dependent*. Because there is no way of expressing domain-dependent queries in RA, Codd restricted himself to domain-independent TRC. In contrast, all RA queries are implicitly domain-independent; thereafter, the algorithm merely proved that RA is a subset of TRC. [7]

Implementing the conversion algorithm would be of little utility for learning purposes because no attention was paid towards optimization - the resulting queries can become very complicated. [8] In addition to that, deciding whether a query is domain-independent is an undecidable problem[1] which makes sanitizing the input queries impossible. [2]

[1] A problem with a yes or no answer is undecidable if no algorithm that always terminates with a correct answer exists. [9]

Chapter 4

Conversion from RA to TRC

Contents

4.1 Introduction

We will not be focusing on a conversion from TRC into RA as it is already well described in the literature. [3, 8] Instead, we will try to define rules for the opposite conversion for which very few sources were found.

As was established in chapter 3, we will focus on a TRC with a single tuple variable. In addition to the mentioned reasons, there appears to be no actual source that specifies the conversion apart from lecture notes [10] which use multiple tuple variables.

We will omit the conversion of renaming and outer joins as our model would need a special extension for those and we will be working with the most basic one.

4.2 Conversion of Atomic Operations

We will work with expressions $E_R = \{ t \mid \varphi(t) \}$ and $E_S = \{ t \mid \psi(t) \}$. Relations represented by $\varphi(t)$ and $\psi(t)$ have attributes $r_1, \ldots, r_m$ and $s_1, \ldots, s_n$, respectively. Note that they may not be actual relations from the schema - they may have gone through projections, natural joins, etc.

4.2.1 Relation

Firstly, we need to convert individual relations. A relation Q can be converted into TRC as [10]

$$Q \equiv \{\, t \mid Q(t) \,\}, \tag{4.1}$$

meaning that we want to select all tuples t such that t belongs to Q.

4.2.2 Projection

As we discussed in the chapter 3.4, we can pick a subset of attributes only by not specifying the belonging of t.

Let us have $\alpha_1, \ldots, \alpha_k$ from $\varphi(t)$. Then the projection on those attributes is [10]

$$\pi_{\alpha_1,\ldots,\alpha_k}(\{\, t \mid \varphi(t) \,\}) \equiv \{\, t \mid \exists z \left(\varphi(z) \wedge \bigwedge_{i=1}^{k} t.\alpha_i = z.\alpha_i \right) \}. \tag{4.2}$$

4.2.3 Selection

Let us have a predicate $P(r_1, \ldots, r_n)$. Conversion formula for selection is [10]

$$\sigma_{P(r_1,\ldots,r_n)}(\{\, t \mid \varphi(t) \,\}) \equiv \{\, t \mid \varphi(t) \wedge P(t) \,\}, \tag{4.3}$$

where $P(t)$ is the same as $P(r_1, \ldots, r_n)$ but each r_i is replaced by $t.r_i$. We are saying that we want to find all t's such that $\varphi(t)$ is true with the addition that $P(t)$ is also true.

4.2.4 Union

When evaluating union, the resulting set contains all elements that belong to the first set or (non-exclusively) to the second set. This notion translates well into predicate logic as [10]

$$\{\, t \mid \varphi(t) \,\} \cup \{\, t \mid \psi(t) \,\} \equiv \{\, t \mid \varphi(t) \vee \psi(t) \,\}. \tag{4.4}$$

4.2.5 Difference

Difference, similarly to union, has an intuitive translation. When placed between two sets, we are looking for all elements that belong to the first set and simultaneously are not in the second one. By translating that logic, we get [10]

$$\{\, t \mid \varphi(t) \,\} \setminus \{\, t \mid \psi(t) \,\} \equiv \{\, t \mid \varphi(t) \wedge \neg\psi(t) \,\}. \tag{4.5}$$

4.2.6 Cartesian Product

Suppose that $\varphi(t)$ and $\psi(t)$ do not share any attributes. The Cartesian product can be expressed in two ways. An easier way of looking at the problem is to use the actual definition of Cartesian product from set theory [11]

$$\{\, t \mid \varphi(t) \,\} \times \{\, t \mid \psi(t) \,\} \equiv \{\, t, v \mid \varphi(t) \wedge \psi(v) \,\}; \tag{4.6}$$

however, we only have one tuple variable available. Fortunately, there is an alternative way of defining Cartesian product. [12] We need to introduce two bound variables p and q, one for each of the relations, and then use the concept of not specifying the belonging of t to assign the attributes to it, resulting in

$$\begin{gathered} \{\, t \mid \varphi(t) \,\} \times \{\, t \mid \psi(t) \,\} \equiv \\ \{\, t \mid \exists p\, \exists q \left(\varphi(p) \wedge \psi(q) \wedge \bigwedge_{i=1}^{m} t.r_i = p.r_i \wedge \bigwedge_{i=1}^{n} t.s_i = q.s_i \right) \}. \end{gathered} \tag{4.7}$$

4.3 Conversion of Derived Operations

Now that all atomic operations are defined, we can just expand any RA expression and convert it notwithstanding a possible loss of compactness of the resulting expression. We will take a different route instead and explore the ways in which we can optimize the results of derived operations.

4.3.1 Intersection

Set intersection can be expressed in terms of set difference as

$$A \cap B = A \setminus (A \setminus B). \tag{4.8}$$

And by using 4.5 we receive

$$\begin{gathered} \{\, t \mid \varphi(t) \,\} \cap \{\, t \mid \psi(t) \,\} \equiv \\ \{\, t \mid \varphi(t) \,\} \setminus (\{\, t \mid \varphi(t) \,\} \setminus \{\, t \mid \psi(t) \,\}) \equiv \\ \{\, t \mid \varphi(t) \wedge \neg(\varphi(t) \wedge \neg\psi(t) \,\} = \\ \{\, t \mid \varphi(t) \wedge (\neg\varphi(t) \vee \psi(t)) \,\} = \\ \{\, t \mid (\varphi(t) \wedge \neg\varphi(t)) \vee (\varphi(t) \wedge \psi(t)) \,\} = \\ \{\, t \mid \varphi(t) \wedge \psi(t)) \,\}. \end{gathered} \tag{4.9}$$

If we were applying the rules as a computer, we would not go beyond the third row even though the expression can be nicely simplified.

4.3.2 Division

Let $\varphi(t)$ and $\psi(t)$ have attributes $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l$ and $\beta_1, \ldots, \beta_l$ respectively. Division between R and S is defined as [4]

$$R \div S = \pi_{\alpha_1,\ldots,\alpha_k}(R) \setminus \pi_{\alpha_1,\ldots,\alpha_k}\left((\pi_{\alpha_1,\ldots,\alpha_k}(R) \times S) \setminus R\right). \tag{4.10}$$

We could try to derive the TRC formula by using 4.10, however, the resulting formula would be quite complicated and it is not clear how to simplify it by means of algebraic manipulation.

Recall the example 2.1. We wanted to find a user whose languages all appear in the right relation. Programmatically, we need to find tuples from the user attribute for which all occurrences of a language from the right relation can be found somewhere next to them in the left relation. This can be generalized as [13, 14, 15]

$$\begin{gathered}\{\, t \mid \varphi(t) \,\} \div \{\, t \mid \psi(t) \,\} \equiv \\ \{\, t \mid \forall q \left(\psi(q) \Rightarrow \exists p \left(\varphi(p) \wedge \bigwedge_{i=1}^{l} p.\beta_i = q.\beta_i \wedge \bigwedge_{i=1}^{k} t.\alpha_i = p.\alpha_i \right) \right) \,\}.\end{gathered} \tag{4.11}$$

According to [15], it is important to use implication instead of logical and. The reason is that if there were no languages in the right relation, we would get an empty relation instead of a relation of all users[1]. The difference is that $\psi(q) \Rightarrow \varphi(q)$ is true when $\psi(q)$ is false.

Following the fact that

$$\psi(q) \Rightarrow \varphi(q) \equiv \neg\psi(q) \vee \varphi(q), \tag{4.12}$$

we can eliminate the implication by rewriting the formula 4.11 to

$$\{\, t \mid \forall q \left(\neg\psi(q) \vee \exists p \left(\varphi(p) \wedge \bigwedge_{i=1}^{l} p.\beta_i = q.\beta_i \wedge \bigwedge_{i=1}^{k} t.\alpha_i = p.\alpha_i \right) \right) \,\}. \tag{4.13}$$

[1] If there are no languages in the right relation, all users satisfy the condition that they know all of them.

4.3.3 Natural Join

Let us split the attributes of R and S into four sets:

1. $C = \{c_1, \ldots, c_o\}$ are the common attributes between R and S.
2. $D = \{d_1, \ldots, d_o\}$ are dummy attribute names that are not in R nor S.
3. $R' = \{r'_1, \ldots, r'_k\}$ are the attributes from R without the common ones.
4. $S' = \{s'_1, \ldots, s'_l\}$ are the attributes from S without the common ones.

Natural join is defined as [4]

$$\pi_{r'_1,\ldots,r'_k,c_1,\ldots,c_o,s'_1\ldots s_l}(\sigma_{\bigwedge_{i=1}^{o} d_i=c_i}(\rho_{d_1/c_1,\ldots,d_o/c_o}(R) \times S)). \tag{4.14}$$

Notice that we cannot even evaluate the part $\rho_{d_1/c_1,\ldots,d_o/c_o}(R) \times S$ since we do not have renaming in TRC and we cannot do a Cartesian product between relations with non-disjoint attributes - we have to arrive to the same result using a different method.

Let us have an example of natural join evaluation with tables *CountrySymbols* and *CountryAbrreviations*.

ISO	**symbol**
CAN	maple leaf
CAN	beaver
CZE	lion
UKR	trident
UKR	nightingale
ARG	Sun of May

$\bowtie$

ISO	**country**
CAN	Canada
CZE	Czech Republic
POL	Poland
UKR	Ukraine
HRV	Croatia

$=$

ISO	**country**	**symbol**
CAN	Canada	maple leaf
CAN	Canada	beaver
CZE	Czech Republic	lion
UKR	Ukraine	trident
UKR	Ukraine	nightingale

. (4.15)

We first need to define a variable for each of the relations. We will call them p and q respectively and in the end, we will assign the tuples that we want in the resulting relation indirectly.

First, we restrict ourselves only to tuples from the common attributes that are in both relations. Here, we are looking at the tuples from the *ISO* attribute. We do that by writing $p.\text{ISO} = q.\text{ISO}$ or, generally, $p.c_i = q.c_i$.

This gives us *CAN*, *CZE* and *UKR* - these are the shared tuples which we assign to the resulting t by $t.\text{ISO} = p.\text{ISO}$, or we can choose the other variable and write $t.\text{ISO} = q.\text{ISO}$ - they both have the *ISO*. Generalized version of that is $t.c_i = p.c_i$.

Now we need to add the *symbol* and the *country* to that.

To add the *symbol*, we write $t.\text{symbol} = p.\text{symbol}$ or $t.r'_i = p.r'_i$ in general because the *country* lies in the set R'.

With the exact same reasoning, we add $t.\text{country} = q.\text{country}$ or $t.s'_i = p.s'_i$.

A TRC expression for 4.15 is

$$\begin{aligned}\{\, t \mid \exists p\, \exists q(&\text{CountrySymbols}(p) \wedge \text{CountryAbbreviations}(q) \wedge \\ &p.\text{ISO} = q.\text{ISO} \wedge t.\text{symbol} = p.\text{symbol} \wedge t.\text{country} = q.\text{country})\,\}.\end{aligned} \tag{4.16}$$

We can generalize that into

$$\begin{aligned}\{\, t \mid \varphi(t)\,\} \bowtie \{\, t \mid \psi(t)\,\} \equiv \{\, t \mid \exists p\, \exists q \\ \left(\varphi(p) \wedge \psi(q) \wedge \bigwedge_{i=1}^{o} p.c_i = q.c_i \wedge \bigwedge_{i=1}^{o} t.c_i = p.c_i \wedge \bigwedge_{i=1}^{k} t.r'_i = p.r'_i \wedge \bigwedge_{i=1}^{l} t.s'_i = q.s'_i\right)\,\}.\end{aligned} \tag{4.17}$$

4.3.4 Left and Right Semijoin

Semijoins can be reduced to atomic operations as [4]

$$E_R \ltimes E_S \equiv \pi_{r_1,\ldots,r_m}(E_R \bowtie E_S), \tag{4.18}$$

$$E_R \rtimes E_S \equiv \pi_{s_1,\ldots,s_n}(E_R \bowtie E_S). \tag{4.19}$$

We see that it is the same thing as natural join (4.17) but we do not want the resulting t to have all the attributes but just the left or the right ones.

We follow the same path as in the natural join example to the point where we are adding the tuples from the left and from the right relation and add only those on the left for the left semijoin and those on the right for the right semijoin.

$$\begin{aligned}\{\, t \mid \varphi(t)\,\} \ltimes \{\, t \mid \psi(t)\,\} \equiv \\ \{\, t \mid \exists p\, \exists q\, \left(\varphi(p) \wedge \psi(q) \wedge \bigwedge_{i=1}^{o} p.c_i = q.c_i \wedge \bigwedge_{i=1}^{m} t.r_i = p.r_i\right)\,\},\end{aligned} \tag{4.20}$$

$$\begin{aligned}\{\, t \mid \varphi(t)\,\} \rtimes \{\, t \mid \psi(t)\,\} \equiv \\ \{\, t \mid \exists p\, \exists q\, \left(\varphi(p) \wedge \psi(q) \wedge \bigwedge_{i=1}^{o} p.c_i = q.c_i \wedge \bigwedge_{i=1}^{n} t.s_i = q.s_i\right)\,\}.\end{aligned} \tag{4.21}$$

A significant number of terms was saved by completely avoiding the projection variable.

4.3.5 Antijoin

Antijoin join can be reduced to [4]

$$E_R \triangleright E_S \equiv E_R \setminus (E_R \ltimes E_S), \tag{4.22}$$
$$E_R \triangleleft E_S \equiv E_S \setminus (E_R \rtimes E_S). \tag{4.23}$$

There is, however, nothing we can do to simplify the resulting query.

4.3.6 Theta join

Theta join can be reduced to [4]

$$E_R \bowtie_{P(r_1,\ldots,r_m,s_1,\ldots s_n)} E_S \equiv \sigma_{P(r_1,\ldots,r_m,s_1,\ldots s_n)}(E_R \times E_S). \tag{4.24}$$

Albeit we clearly will not be able to simplify it, we will use the resulting conversion later.

Using 4.7 and 4.3, we get

$$\begin{gathered}\{\, t \mid \varphi(t) \,\} \bowtie_{P(r_1,\ldots,r_m,s_1,\ldots s_n)} \{\, t \mid \psi(t) \,\} \equiv \\ \{\, t \mid \exists p\, \exists q \left(\varphi(p) \wedge \psi(q) \wedge \bigwedge_{i=1}^{m} t.r_i = p.r_i \wedge \bigwedge_{i=1}^{n} t.s_i = q.s_i \right) \wedge P(t) \,\},\end{gathered} \tag{4.25}$$

4.3.7 Theta semijoin

Theta semijoin is reduced to [4]

$$\begin{aligned} E_R \ltimes_{P(r_1,\ldots,r_m,s_1,\ldots s_n)} E_S &\equiv \pi_{r_1,\ldots,r_m}(E_R \bowtie_{P(r_1,\ldots,r_m,s_1,\ldots s_n)} E_S) \\ E_R \rtimes_{P(r_1,\ldots,r_m,s_1,\ldots s_n)} E_S &\equiv \pi_{s_1,\ldots,s_n}(E_R \bowtie_{P(r_1,\ldots,r_m,s_1,\ldots s_n)} E_S). \end{aligned} \tag{4.26}$$

We will just use the same expression as 4.25 and cut the attributes we are not projecting onto out of the Cartesian product part:

$$\begin{gathered}\{\, t \mid \varphi(t) \,\} \ltimes_{P(r_1,\ldots,r_m,s_1,\ldots s_n)} \{\, t \mid \psi(t) \,\} \equiv \\ \{\, t \mid \exists p\, \exists q \left(\varphi(p) \wedge \psi(q) \wedge \bigwedge_{i=1}^{m} t.r_i = p.r_i \right) \wedge P(t) \,\}, \\ \{\, t \mid \varphi(t) \,\} \rtimes_{P(r_1,\ldots,r_m,s_1,\ldots s_n)} \{\, t \mid \psi(t) \,\} \equiv \\ \{\, t \mid \exists p\, \exists q \left(\varphi(p) \wedge \psi(q) \wedge \bigwedge_{i=1}^{n} t.s_i = q.s_i \right) \wedge P(t) \,\}.\end{gathered} \tag{4.27}$$

Chapter 5

Inputting Queries

Contents

Because the standard notation in RA uses subscripts and Greek letters, it is not a trivial task to design an easy-to-use input field with an intuitive input grammar. We will take a look at how other RA calculators attempt to tackle this problem[1] and then, based on this analysis, we will design our own solution in the chapter 6.

5.1 Other Tools

5.1.1 RelaX

ReLax [16] is an online tool for evaluating RA queries. The input field is accompanied by a toolbar containing symbols that are difficult to type on a keyboard. Without reading a manual, it is not clear how to input queries such as selection or projection. The subscript part is simply solved by separating the inner expression by spaces, so the queries look like this:

$$\begin{gathered} \pi \ \mathrm{r1}, \mathrm{r2} \ \mathrm{R}, \\ \sigma \ \mathrm{r1} = 4 \wedge \mathrm{r2} = 6 \ \mathrm{R}, \\ \mathrm{R} \bowtie \mathrm{r1} = 4 \ \mathrm{S}, \end{gathered} \tag{5.1}$$

with the numbers and operators highlighted for better readability.

[1] Note that the statements and views expressed herein are those of the author.

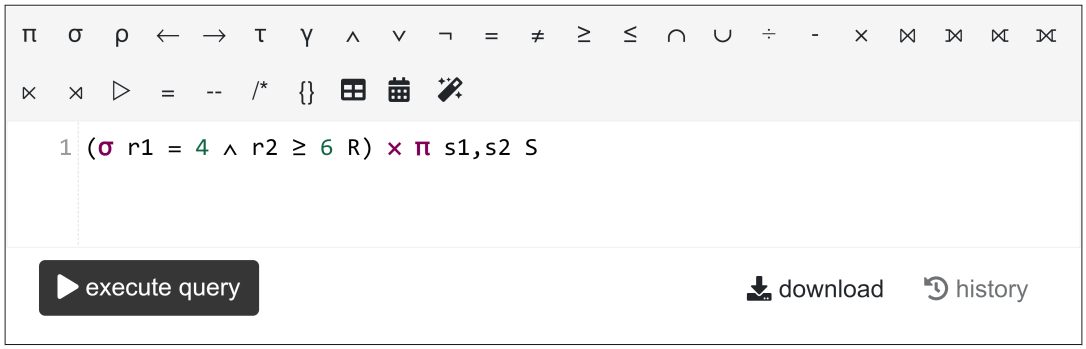


Figure 5.1: RelaX input field

There is also an option to use words for Greek letters (sigma, pi, etc.) and logical symbols (and, or) which can then be all replaced by their Unicode counterparts by clicking a button. The benefit of that is that we can comfortably write a query without using the toolbar.

Even though there are examples of usage when hovering over toolbar buttons, it can be argued that an additional effort has to be made in order to understand the semantics of the input field.

5.1.2 Rachel

Rachel (Relational Algebra CHecker and EvaLuator) [17] has been developed by a FEL CTU student Lukáš Kotlík as an alternative to RelaX. Because it only works with simplified notation, it uses a simple default input field with a toolbar below it.

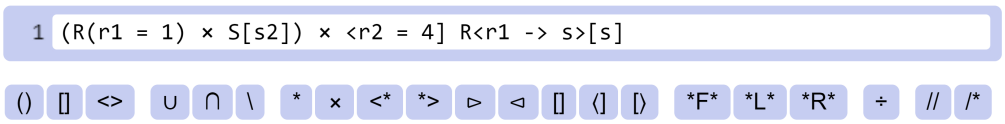


Figure 5.2: Rachel input field

One thing to note is that when we click on the renaming, it automatically outputs the first arrow and moves the cursor to the left of it. That is a great way of guiding the user without explicitly telling them what to do.

5.1.3 RAT

RAT (Relational Algebra Translator) [18] is a desktop application for converting RA queries into SQL.

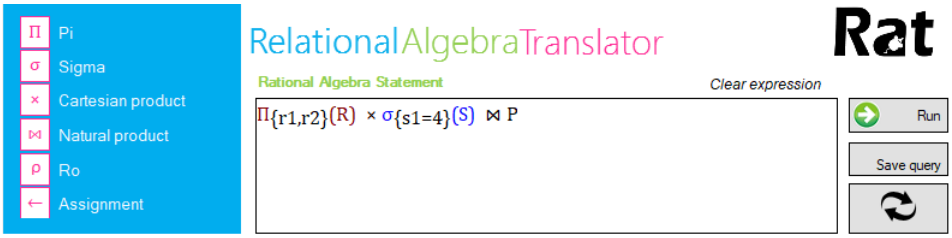


Figure 5.3: RAT input field

Operations can be chosen from the toolbar on the left side. Once we choose an operation such as projection, we automatically get a subscript block for the columns, and parentheses for the inner expression. The application also supports keyboard shortcuts such as `CTRL + S` for selection, which makes inputting the expressions faster.

Overall, this application offers a very intuitive way of entering queries; mainly because of its support for visual subscript.

5.2 Conclusion

Table 5.1 summarizes the discussed features.

Feature	RelaX	Rachel	RAT
Toolbar with unusual symbols	✓	✓	✓
Automatic completion		✓	
Keyboard shortcuts			✓
Example of usage on operator hover	✓		
Operator highlighting	✓		✓
Visual subscript			✓

Table 5.1: Overview of features of analysed tools

While RAT has the most of the discussed features available, it is a desktop application which may have made the implementation easier. Although Rachel only has two of the features ticked off, it has a great user manual and only works with simplified notation, which makes the visual subscript superfluous.

Most users would be clueless without a toolbar - it is a must.

The automatic completion makes it a little easier when long relation names are used, although it could be argued that it does not make much of a difference for the user as the schema can always be visible below the input field.

When writing very long queries, the keyboard shortcuts might seem to be useful; however, the difference between clicking on a toolbar item and using the shortcut is negligible. Most users will not write such long queries anyway.

The examples of usage have proven to be a great practice, especially when the tool does not offer the visual subscript, the absence of which was very confusing when using RelaX for the first time.

Operator highlighting is a small detail that enhances readability and is easy to implement. It is, for sure, worth adding into the tool.

The visual subscript is very appealing because it makes the queries look exactly the same as in the textbooks right when typing them, although it might be difficult to implement.

Chapter 6

Application

Contents

This chapter contains all the information about the application. The first section shows the architecture and explains the conversion process in terms of the communication between the browser and the server. After that, there is a use-case diagram with a brief explanation of the available functionalities. The parsing section then shows how the parsing was implemented, what grammars were used and what role visitors play in the application. Since both grammars and visitors had been explained at that point, a brief explanation of the conversion process is presented. The rest of the implementation is explained in the classes section, where we take a look at a class diagram with an explanation of some of the classes. The frontend section contains a screenshot of the application GUI with a description of the controls. Finally, there is an explanation of the testing process and an installation manual with instructions for downloading and deploying the application.

6.1 Architecture

The application consists of two parts - frontend in JavaScript framework called React which communicates with Rest API on the backend. The Rest controller and the service layer are under a Java framework called Spring. The diagram 6.1 depicts communication between the user (Client browser) and the server (Web server).

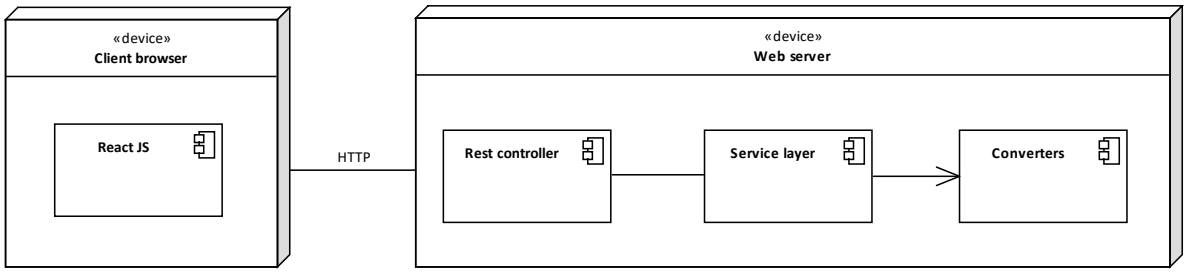


Figure 6.1: Deployment diagram of the application

Conversion of an input query has the following steps:

1. User enters the website with a GUI through which they enter an expression and choose options for the conversion.
2. After clicking on a conversion button, the input is wrapped into JSON format and sent through HTTP to a dedicated Rest API endpoint.
3. Upon receiving the request, the Rest controller unpacks the JSON, converts it to Java context and sends it to the service layer.
4. Service layer then takes care of the conversion logic. It calls the parser, converters and other helping classes and it either produces a result or an error message. When it finishes, it wraps the result back into JSON and sends it to the user.
5. User sees a result or an error message in the output field.

The decision to make a web app instead of a desktop app was made because it is the easiest way to offer the service to everyone regardless of the platform they are using. It is easy to access, simple, mobile-friendly, and does not require any installation. Furthermore, the API can communicate with any other website, not just this one.

6.2 Use Cases

The diagram 6.2 depicts the actions a user can take in the application.

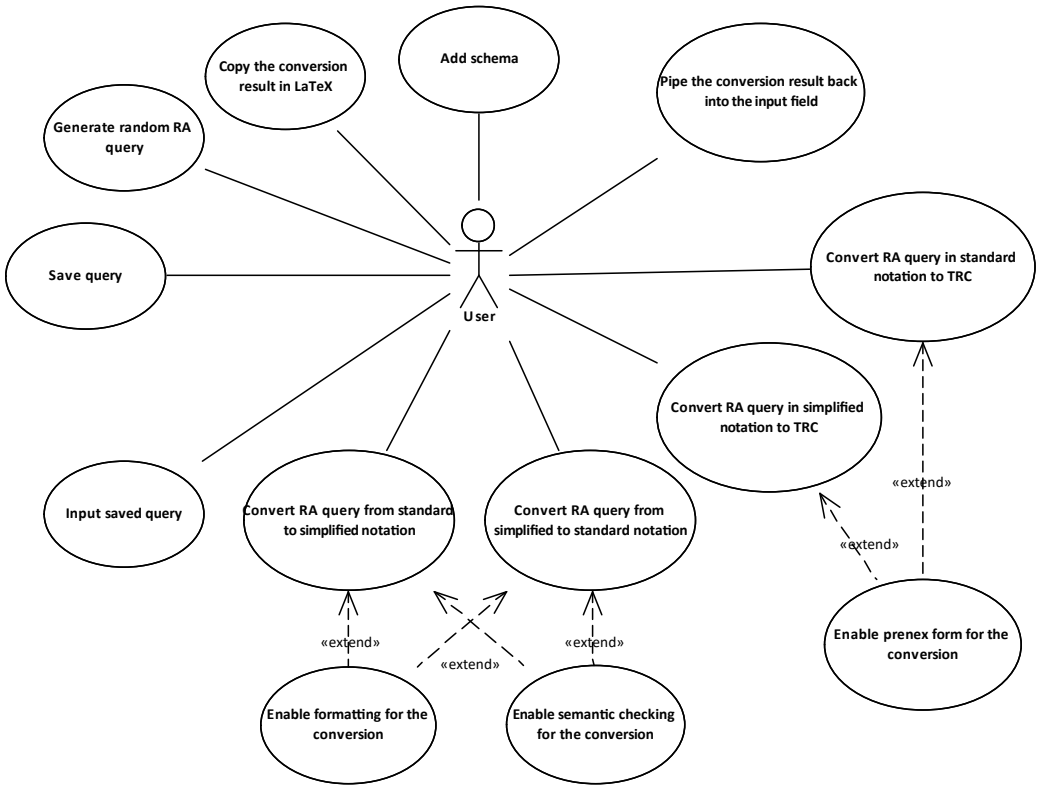


Figure 6.2: Use case diagram

On the bottom right, there is an option to enable prenex normal form for TRC expressions. It means that all the quantifiers will get grouped without negations before the inner formula. It was added as a demonstration that the conversion is not solely string-based and that we have a syntax tree available.

Saving and restoring queries has initially been made as a tool for easier testing, but it seemed to be a useful utility, so it was left there as a feature.

Putting the result back into the input field is useful for demonstrating that the notation converters are actually equivalent - when we convert any valid RA expression into the other notation, put the result back into the input field, switch notations and click on the conversion again, we always get the initial expression.

Since the TRC converter only works with a valid expression under a given schema, it was easy to offer the functionality of checking the semantic correctness of the RA query also during the conversion between notations.

6.3 Parsing

In order to be able to manipulate with user queries, we need to parse them into an abstract syntax tree (AST). For that, we will use a tool called ANTLR.

ANTLR is a powerful program in the category of compiler generators[1]. It is widely used not only in the industry but also in academia and its purpose is to take a formal language description and produce a code that recognizes strings from the given grammar. There is an API through which we can intercept the parsing process and save the input into our own syntax tree. [20]

6.3.1 Grammars

Formal grammar is a description of some language that consists of *terminals*, *non-terminals*, starting symbol and a set of *production rules*. The production rules specify how to form strings from a language's alphabet. Terminals are just symbols from the alphabet and they cannot be further substituted for, while non-terminals serve as variables on which we apply the rules. [21]

When we look at the table 2.1, we can see that the notations only differ in the way RA operators are written, while the logical conditions in selections and theta joins (together called theta conditions) are the same. For that reason and because ANTLR supports importing grammars into other grammars, we can define a separate grammar for theta conditions and import it into the grammars for the individual notations. Listing 6.1 shows the theta condition grammar in ANTLR notation.

The words written in capital letters are the terminals. They are specified by string literals or regular expressions. The regular expression for the `IDENTIFIER` says that it must be a sequence of alphanumeric characters that does not start with a number. The `STRING_VALUE` allows for any sequence of alphanumeric characters as long as they are surrounded with quotes. The `WS` is a directive that tells ANTLR to skip spaces when parsing.

Non-terminals are written in camel-case and they are defined by the production rules specified after the colon. When there are multiple productions rules for one non-terminal, a pipe symbol `|` is used to separate them.

The names on the right side starting with the hashtag are specifying which visitor methods should be generated. This visitor will allow us to access a parsing context and insert the values into our AST. We will see that in the next section.

As we can see, only $\wedge$, $\vee$, and $\neg$ are allowed in the theta condition. They were chosen because the set $\{\wedge, \vee, \neg\}$ is functionally complete [22], meaning that we can infer any other logical operation by only using those three.

[1] Compiler generator or a compiler-compiler is a tool that generates a code for a parser, interpreter or compiler [19]

```
grammar ThetaCondition;

NUMBER : ('0' .. '9') + ('.' ('0' .. '9') +)?
       ;

STRING_VALUE : '\'' [a-zA-Z0-9_-]+ '\''
             ;

IDENTIFIER : [a-zA-Z][a-zA-Z0-9_-]*
           ;

WS : [ \r\n\t]+ -> skip
   ;

COMPARISON_OP : '>' | '<' | '=' | '\\lneq' | '\\leq' | '\\geq'
              ;

BINARY_CONNECTIVE : '\\land' | '\\lor'
                  ;

UNARY_CONNECTIVE : '\\lnot'
                 ;

formula : columnSpec COMPARISON_OP columnSpec         # Predicate
        | columnSpec COMPARISON_OP term               # Predicate
        | term       COMPARISON_OP columnSpec         # Predicate
        | '(' formula ')'                             # FormulaParentheses
        | UNARY_CONNECTIVE formula                    # NotOperation
        | formula BINARY_CONNECTIVE formula           # BinaryLogicalOperation
        | formula BINARY_CONNECTIVE formula           # BinaryLogicalOperation
        ;

columnSpec : IDENTIFIER                               # ColumnSpecification
           ;

term : STRING_VALUE                                   # StringTerm
     | NUMBER                                         # NumberTerm
     ;
```

Listing 6.1: Theta condition grammar

Listing 6.2 contains a standard RA grammar with the theta condition imported from 6.1. The definitions we have already seen are omitted.

```
1  grammar RAStandard;
2
3  import ThetaCondition;
4
5  BINARY_OPERATION : '\\cap' | '\\cup' | '\\times' | '\\setminus' | '\\div'
6                   ;
7
8  REGULAR_JOIN : '\\triangleleft' | '\\triangleright' | '\\leftouterjoin' |
9                 '\\rightouterjoin' | '\\fullouterjoin'
10              ;
11
12 THETA_JOIN : '\\bowtie' | '\\ltimes' | '\\rtimes'
13            ;
14
15 root : expr EOF
16      ;
17
18 expr : '\\pi' '_{' columnList '}' '(' expr ')'        # Projection
19      | '\\sigma' thetaCondition '(' expr ')'          # Selection
20      | '\\rho' '_{' renameList '}' '(' expr ')'       # Rename
21      | expr BINARY_OPERATION expr                     # BinaryOperation
22      | expr REGULAR_JOIN  expr                        # JoinOperation
23      | expr THETA_JOIN thetaCondition expr            # JoinOperation
24      | '(' expr ')'                                   # Parentheses
25      | IDENTIFIER                                     # Relation
26      ;
27
28 thetaCondition : ('_{' (formula)? '}')?
29                ;
30
31 columnList : IDENTIFIER ( ',' IDENTIFIER )*
32            ;
33
34 renameList : IDENTIFIER '/' IDENTIFIER ( ',' IDENTIFIER '/' IDENTIFIER )*
35            ;
```

Listing 6.2: Standard RA grammar

The `root` is a starting rule here. ANTLR's default behaviour is that once it accepts a string, it also accepts everything that comes after it. That is bypassed by specifying the `EOF` after an expression.

Regular expression operators can also be used in the production rules as we can see in the `columnList` where the $*$ is used to signify zero or more allowed occurrences.

Finally, the listing 6.3 contains a grammar for the simplified notation.

```
grammar RASimplified;

import ThetaCondition;

BINARY_OPERATION_SYMBOL : '\\cap' | '\\cup' | '\\times' | '\\setminus' | '\\div'
                        ;

JOIN_OPERATION_SYMBOL : '*' | '<*' | '*>' | '*L' | '*_L' | '*_{L}' | '*R' |
                        '*_R' | '*_{R}' | '*F' | '*_F' | '*_{F}'|
                        '\\triangleleft' | '\\triangleright'
                      ;

root : expr EOF
     ;

expr : expr '[' columnList ']'                          # Projection
     | expr '(' thetaCondition ')'                      # Selection
     | expr '\\langle' renameList '\\rangle'            # Rename
     | expr BINARY_OPERATION_SYMBOL expr                # BinaryOperation
     | expr JOIN_OPERATION_SYMBOL expr                  # JoinOperation
     | expr '[' thetaCondition ']' expr                 # JoinOperation
     | expr '\\langle' thetaCondition ']' expr          # JoinOperation
     | expr '[' thetaCondition '\\rangle' expr          # JoinOperation
     | '(' expr ')'                                     # Parentheses
     | IDENTIFIER                                       # Relation
     ;

thetaCondition : (formula)?
               ;

columnList : IDENTIFIER ( ',' IDENTIFIER )*
           ;

renameList : IDENTIFIER '\\rightarrow' IDENTIFIER ( ',' IDENTIFIER '\\rightarrow
    ' IDENTIFIER )*
           ;
```

Listing 6.3: Simplified RA grammar

In the `JOIN_OPERATION_SYMBOL`, there are also variants of outer join symbols without subscript to avoid confusing users who expect the simplified notation to be subscript free.

Now that we have the grammars ready, we need an AST. We will use a hierarchy with an abstract class `Expression` which will be extended by everything else.

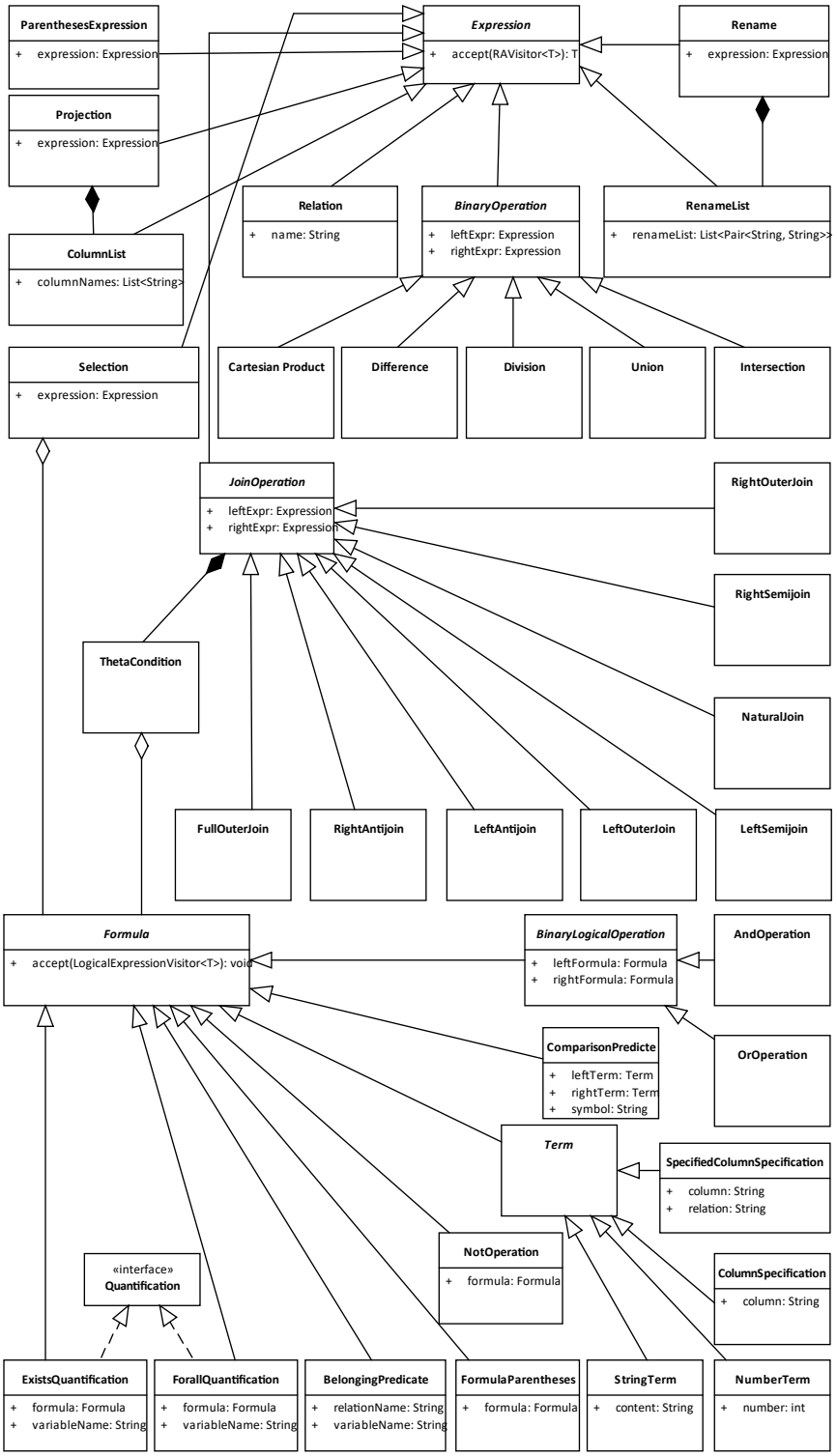


Figure 6.3: Abstract syntax tree for RA and TRC

Because we also need to store TRC expressions, we can make use of the similarities between theta condition and TRC formula. The difference is that TRC formula allows quantifiers and uses different terminals. For that reason, classes `BelongingPredicate`, `Quantification`, and `SpecifiedColumnSpecification` were added.

6.3.2 Visitors

Visitors play an integral part in the application. That is not surprising because the most common use case for the use of visitors is to define operations on AST.

Visitors allow us to add an operation to the AST without having to change the signature of its classes. [23] There is a method

```
1 public <T> T accept(RAVisitor<T> raVisitor) {
2     return raVisitor.visit(this);
3 }
```

Listing 6.4: Accept method for RA expressions

at every subclass of RA `Expression`.

The visitors contain a visit method overloaded for every subclass of `Expression` as well as one for the `Expression` itself, which always looks like this:

```
1 public T visit(Expression expression) {
2     return expression.accept(this);
3 }
```

Listing 6.5: Visit method for Expression

That is because when we, for instance, call a visit method with a projection in the argument, the visit method with an expression type gets called. It then calls the accept method with an instance of the visitor in the argument. That causes an accept method of a correct subclass to get called - a projection class in this case. Now that we are in the context of projection a visit method of the visitor from the argument gets called again, now being invoked directly from the projection class. Finally, the specific visit method (the method having `Projection` in the argument) is called. This process is called *double dispatch* and since Java does not support it directly, we have to use visitors. [24]

Using this pattern, we can unpack the expression from the root and perform a computation at each level. Because the visit method has a return value, we can also work with results from the subtrees.

The class diagram 6.4 shows the structure of the visitors used for parsing.

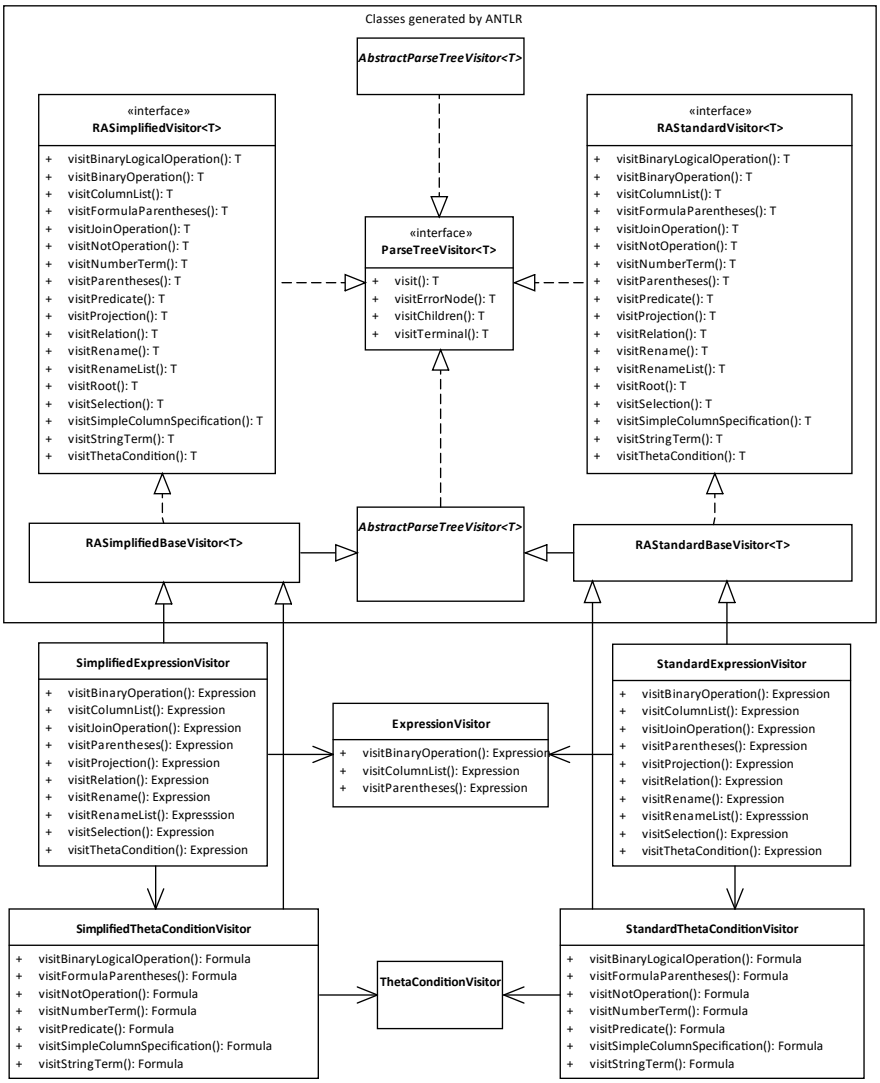


Figure 6.4: Class diagram of visitors used for parsing

The classes in the square are generated by ANTLR. Although the generated base visitors contain all the visit methods, they can be extended by multiple classes, each implementing some of the methods. That feature was used to differentiate between an expression visitor and a theta condition visitor.

Each overridden visit method gets a parsing context in the argument which they can access to create the needed structure. Listing 6.6 shows an example of a `visitProjection` method implementation.

```
1 @Override
2 public Expression visitProjection(RAStandardParser.ProjectionContext ctx) {
3     ColumnList columnList = (ColumnList) this.visit(ctx.columnList());
4     Expression expression = this.visit(ctx.expr());
5
6     return new Projection(columnList, expression);
7 }
```

Listing 6.6: Visit method for projection

Lastly, `ExpressionVisitor` and `ThetaConditionVisitor` are helping classes that implement a common behavior. They are used by the main visitors.

6.4 Conversion Process

The visitors simulate a process that is equivalent to taking a preorder path through the AST. A figure 6.5 shows how the expression 6.1 gets converted into simplified notation.

$$\sigma_{x=y}(R) \times \pi_a(S \bowtie Q). \tag{6.1}$$

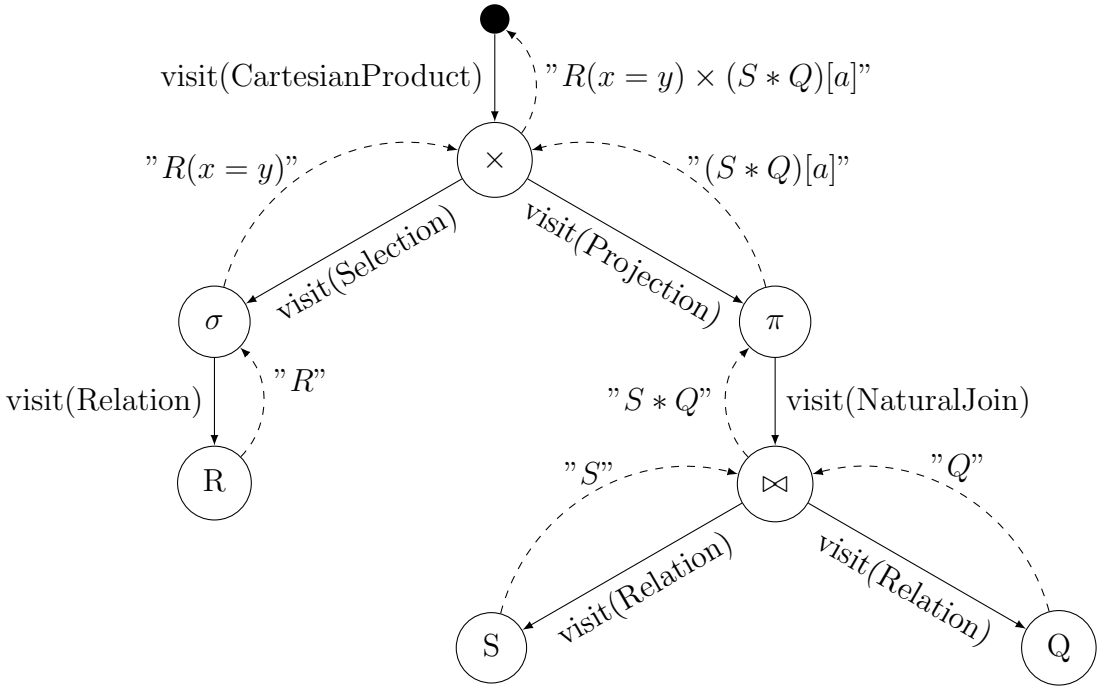


Figure 6.5: Diagram of a notation conversion process

The filled arrows represent a method call, while the dashed ones show the return value. We can see that the branching always stops at the relation level as it is the only terminal symbol in RA.

Conversion into TRC works on a similar basis; however, instead of returning a string straight away, a pair `<Formula, Header>` is returned from the visit method. Some operations such as Cartesian product or division need to know the attributes to construct the TRC expression. Let us have an example

$$(R \cap S) \times \pi_{q_1}(Q), \tag{6.2}$$

where both R and S have attributes r_1, r_2 and Q has attributes q_1, q_2, q_3. Diagram 6.6 shows how the expression 6.2 gets converted into TRC.

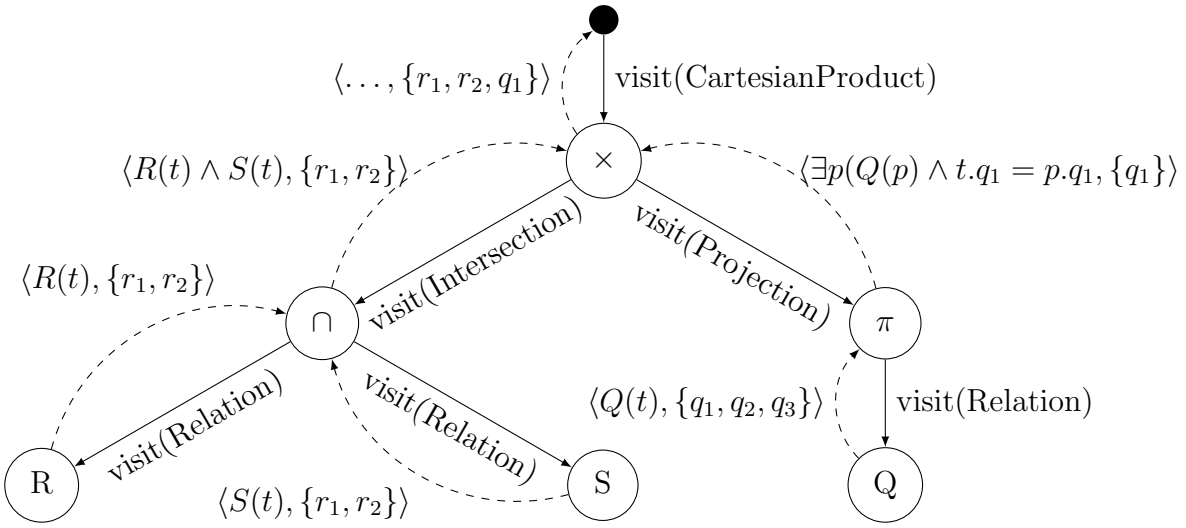


Figure 6.6: Diagram of a conversion from RA into TRC

Notice that the projection and Cartesian product both return a different header. Without the headers, the Cartesian product visit method would have no way of knowing that the q_1 has been projected onto, and it would add the q_2 and q_3 back into the resulting expression.

6.5 Classes

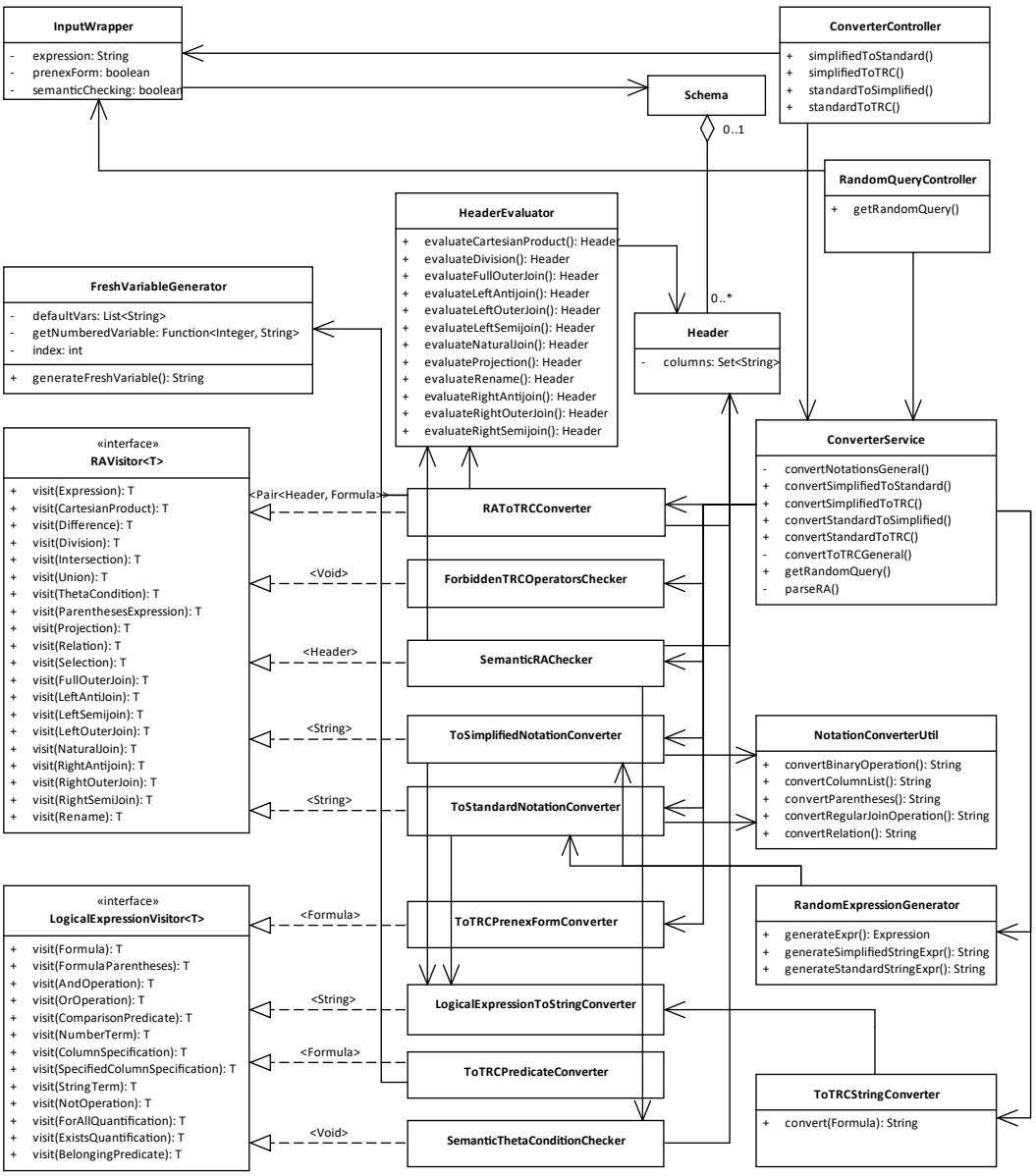


Figure 6.7: Class diagram of the application

The diagram 6.7 depicts relationships between the rest of the classes in the application. Because the visitors use generic types, the label in angle brackets above the dotted arrows signifies which type was implemented.

Input from the HTTP POST request is captured by the `ConverterController` and converted into the `InputWrapper` which contains all the necessary information about the

expression and selected options for the conversion. It uses a `Schema` object to store the schema in a set of strings.

The wrapped input is then passed into the `ConverterService` which is responsible for delivering either the resulting query or an error message. It controls the converters and parsing visitors and uses them to produce the result.

Evaluation of the resulting header based on a given operation is generalized into `HeaderEvaluator`. For example, in the `evaluateRename` method, it returns a new header with renamed attributes. It is used for semantic checking and during conversion into TRC.

The `RAToTRCConverter` uses a fresh variable generator because it needs a new variable for each quantification. The generator takes a list of default fresh variables and when they run out, it uses a given function to generate new fresh variables indefinitely.

We can see that the checkers both implement the visitor interface with a `Void` type. That is because they do not need to propagate results from the subexpressions, they just record errors into a list above the scope of their visit methods. Conversely, `SemanticRAChecker` uses a `Header` type because it needs to know the resulting header from the subexpressions to recognize errors.

6.6 Frontend

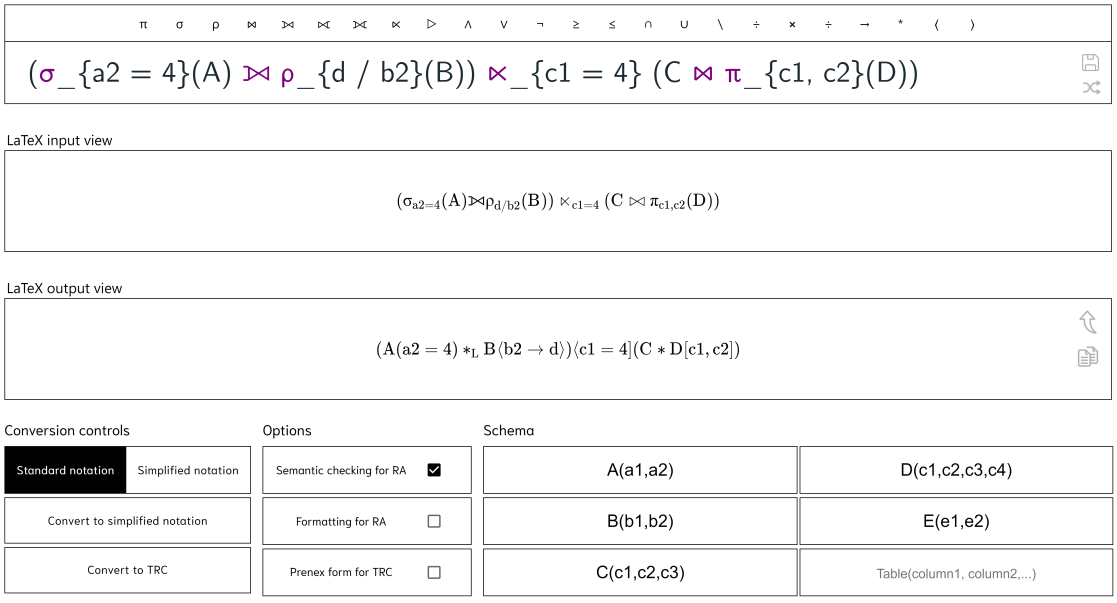


Figure 6.8: Frontend of the application

Regarding the points from the analysis 5.1, all of them have been implemented with the exception of the visual subscript and keyboard shortcuts.

The project uses a rich text input field library called *draft-js* which allows for a substring of the input to be displayed in subscript. The problem is that, unlike in RAT (5.1.3), there is no visual cue that the subscript is on. Because the library does not directly support rendering a separate bordered box for inputting subscript, the decision to make a portion of the input text display as a subscript has to be made just based on the contents of the string. That proved to be very difficult, especially for theta joins.

Due to the clumsiness of the subscript formatting, the input field uses LaTeX notation in the following way:

$$\pi_{a,b}(R) \to \pi\_\{a, b\}(R). \tag{6.3}$$

When the user clicks on a projection, selection, or renaming symbol, the following will appear in the text field:

$$\begin{aligned} \pi &\to \pi\_\{\}(), \\ \sigma &\to \sigma\_\{\}(), \\ \rho &\to \pi\_\{\}(), \end{aligned} \tag{6.4}$$

and the cursor will be placed in the first pair of brackets.

There are two buttons on the right of the input field. The floppy disk saves the current content of the input field into the numbered list above. The saved queries can then be accessed by a left click and discarded by a right-click. The button below generates a random RA expression.

The box labeled *LaTeX input view* is translating the input into LaTeX, so that the user knows what the query looks like.

The second box shows the converted query and error messages. It also has two buttons on the right. The arrow puts the query into the input field and the second button triggers a window containing the query in LaTeX format.

The user can choose which notation they want to convert from in the leftmost section under the views. There are also two buttons that trigger the conversion process. Of course, once the notation is switched to simplified, the text within the middle button changes to *Convert to standard notation*.

To the right of the optional checkboxes are the input fields for the tables from the schema. The user enters the relations and their attributes in a pattern shown in the placeholder of an empty text field. The input fields are checked against a regular expression and the box changes color when the input is not accepted.

6.7 Testing

There are unit tests for every conversion rule between the notations. They test the basic cases like

$$\sigma_{x=y}(R) \rightarrow R(x = y). \tag{6.5}$$

Correctness of the conversion of nested expressions like $\sigma_{x=y}(\pi_{r1}(R))$ is guaranteed by the recursive nature of the conversion process (6.5) - once we are sure that the subexpressions are called recursively, we can inductively assume that if all the simple tests pass, the converter is working properly.

To test the equivalence of the notation conversion, a random query generator has been implemented. It assigns each operation a number and recursively generates the expressions based on the randomly generated integer.

```
1  public static Expression gen(int len) {
2      int n = len <= 0 ? 0 : getRandomNumber(0, 17);
3
4      ThetaCondition tc = new ThetaCondition(null);
5
6      // 50 % chance for an empty theta condition
7      if (n >= 9 && getRandomNumber(0, 1) == 0) {
8          tc = generateThetaCondition(getRandomNumber(1, 3));
9      }
10
11     switch(n) {
12         case 0: return new Relation(getRandomLetter(true));
13         case 1: return new ParenthesesExpression(gen(len - 1));
14         case 2: return new CartesianProduct(gen(len - 1), gen(len - 1));
15         ...
16         case 9: return new Selection(tc, gen(len - 1));
17         case 10: return new NaturalJoin(gen(len - 1), gen(len - 1), tc);
18         case 11: return new RightSemijoin(gen(len - 1), gen(len - 1), tc);
19         ...
20     }
21 }
```

Listing 6.7: Random expression generator

The generator is restricted by a given length because there is only $1 : 17$ probability of selecting a terminal in each recursion level. The probability that the generator terminates is then getting smaller and smaller at each recursion level when selecting binary operations.

The length gets subtracted at each recursion level and when it reaches zero, the variable `n` is turned to 0, guaranteeing that the `Relation` is returned, preventing further recursion.

The generator is then used in a test of conversion equivalence where a random expression is generated, converted into another notation, converted back, and then compared to the original. This process is repeated many times. This test helped to reveal an accumulation of parentheses between the conversions.

The classes for semantic checking have also been tested by unit tests with consideration for edge cases. The tests create a schema and test whether the semantic checker returns an appropriate error.

Furthermore, a series of exercise queries from the Database Systems course at CTU, which were written in both notations by a teacher, have been tested in both ways, and the conversion proved to be correct.

Lastly, there are unit tests for each of the TRC conversion rules defined in chapter 4. Because the conversion into TRC is more nuanced, more complex testing including nested expressions may be needed to give the same assurance of correctness as with the notations conversion.

The tests helped to uncover several errors and gave a reasonable assurance that the application is working properly.

6.8 Deployment Instructions

During the presentation of this thesis, the application will be available at [25]. The react application uses free hosting for one static website by github pages and the backend is deployed on `https://heroku.com`, however, the computation time is limited there.

In case the website is no longer working, the frontend and backend are both accessible on github. To try the application out, it needs to be deployed locally. To run the react application,

1. Install npm[2].
2. Clone or download the application[3].
3. Go to *src/App.js* and change the `BASE_URL` variable on line 10 to `const BASE_URL = 'http://localhost:8080/';`.
4. Open a terminal in the root directory of the application.
5. Run `npm install`.
6. Run `npm run`.
7. Open a browser and go to `http://localhost:3000/`.

To run the backend,

1. Install Java 8 or newer[4].
2. Clone or download the application[5].
3. Download the newest version of Maven[6].
4. Open a terminal in the root directory of the application.
5. Run `mvn clean`.
6. Run `mvn install`.
7. Run `mvn spring-boot:run`.
8. Verify that when going to `http://localhost:8080/`, an OK message gets displayed.

[2]Download and run an installer according to your OS from `https://nodejs.org/en/download/`.
[3]Available at `https://github.com/tomashauser/Relational-Converter-Frontend`
[4]Available at `https://www.oracle.com/java/technologies/downloads/`
[5]Available at `https://github.com/tomashauser/Relational-Converter-Backend`.
[6]Available at `https://maven.apache.org/download.cgi`

Chapter 7

Conclusion

After introducing the different RA notations (2), an analysis of the way of inputting queries in other RA tools has been performed (5), resulting in a comparison table (5.1) which was then used as a reference during the implementation.

A web application for conversion between the notations has been implemented (6). All points from the analysis of other tools (5) were implemented with the exception of keyboard shortcuts and visual subscript. There were technical difficulties in configuring an input field to support it, so a different approach was used. Instead of showing the subscript directly in the input field, user enters it in LaTeX notation (6.3). The input is then in real-time compiled into LaTeX and shown in a separate box below the input field (6.8). It can be argued that this approach has a similar effect as showing the subscript directly in the input field.

After briefly introducing TRC (3), there was the question of the possibilities of a conversion from or into RA. The conversion from TRC into RA has already been established in the literature [5, 8]. The conversion algorithm is very complex and the input queries cannot be sanitized (3.3), so the effort was devoted to the opposite conversion. After deciding to use TRC with only a single tuple variable (3.1) a conversion from RA was assembled using a variety of different sources. [10, 11, 12, 26, 13, 14, 15] The conversion was then implemented into the application (6.6).

The application was thoroughly tested (6.7). A random query generator (6.7) was implemented and used in a test of equivalence between the notation converters. Thanks to the testing, several mistakes and bugs were found and fixed. It can be argued with a reasonable level of confidence that the conversion has been implemented correctly.

The application can be accessed from [25] or downloaded and deployed locally by following the deployment instructions (6.8).

Bibliography

[1] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database Systems Concepts*, 6th ed. McGraw-Hill, 2011, p. 217.

[2] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley Publishing Company, Inc., 1995, pp. 29, 99.

[3] E. F. Codd, "A relational Model of Data for Large Shared Data Banks," vol. 13, no. 6, 1970. [Online]. Available: https://doi.org/10.1145/362384.362685

[4] M. Svoboda, "Lecture notes: Database Systems Lecture 7 - Relational Algebra." [Online]. Available: https://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/lectures/Lecture-07-Relational-Algebra.pdf

[5] E. F. Codd, "Relational Completeness of Data Base Sublanguages," vol. RJ987, 1972.

[6] K. H. Rosen, *Discrete Mathematics and Its Applications*, 8th ed. McGraw-Hill Higher Education, 2019, p. 122.

[7] R. Hull and J. Su, "Domain Independence and the Relational Calculus," University of Southern California, Tech. Rep., 1989. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.9687&rep=rep1&type=pdf

[8] R. Nakano, "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," vol. 15, no. 4, 1990. [Online]. Available: https://doi.org/10.1145/99935.99943

[9] B. Poonen, "Undecidable problems: a sampler," 2012. [Online]. Available: https://arxiv.org/abs/1204.0299

[10] R. Pichler, "Lecture notes: Database Theory - 3. Codd's Theorem," Institute of Logic and Computation DBAI Group TU Wien, p. 11. [Online]. Available: http://www.lsv.fr/~goubault/BD/dbt03.pdf

[11] S. Roman, *Advanced Linear Algebra (Graduate Texts in Mathematics)*, 3rd ed. New York, NY: Springer, 2008, p. 14.

[12] W. A. Weiss, "An Introduction To Set Theory," p. 20, 2008. [Online]. Available: https://www.math.toronto.edu/weiss/set_theory.pdf

[13] "Lecture notes: Introduction to Database Systems - Relational Calculus," University of California, Berkley, 2005. [Online]. Available: https://www-inst.eecs.berkeley.edu//~cs186/fa05/lecs/10CalcSQLI-6up.pdf

[14] R. Ramakrishnan and J. Gehrke, *Database management systems*, 3rd ed. McGraw-Hili, 2003, p. 121.

[15] "Lecture notes: The Logical Structure of Relational Query Languages," Université Libre de Bruxelles, 2008. [Online]. Available: https://cs.ulb.ac.be/public/_media/teaching/infoh303/calculnotes.pdf

[16] J. Kessler, "RelaX - relational algebra calculator," https://dbis-uibk.github.io/relax/, accessed: 2022-03-26.

[17] L. Kotlík, "Rachel," https://kotliluk.github.io/rachel/, accessed: 2022-03-26.

[18] "Rat - Relational Algebra Translator," https://www.slinfo.una.ac.cr/rat/rat.html, accessed: 2022-03-26.

[19] P. D. Terry, *Compilers and Compiler Generators an introduction with C++*. International Thomson Computer Press, 1997, p. 12. [Online]. Available: https://www.kau.edu.sa/Files/830/Files/55520_pdfvers.pdf

[20] T. Parr, "About The ANTLR Parser Generator," https://www.antlr.org/about.html, accessed: 2022-04-06.

[21] T. Jiang, M. Li, B. Ravikumar, and K. Regan, "Formal Grammars and Languages," p. 39, 01 2010.

[22] H. B. Enderton, *A Mathematical Introduction to Logic*, 2nd ed. Academic press, 1972, p. 62.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2009.

[24] L. Bettini, S. Capecchi, and B. Venneri, "Translating Double Dispatch into Single Dispatch," *Electronic Notes in Theoretical Computer Science*, vol. 138, no. 2, pp. 59–78, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066105051352

[25] T. Hauser, "Relational Algebra and Relational Calculus Converter," 2022. [Online]. Available: https://tomashauser.github.io/relational-converter/

[26] J. Pokorný and I. Halaška, "Lecture notes: Query Languages - Relační model dat." [Online]. Available: https://www.ksi.mff.cuni.cz/~pokorny/vyuka/srbd/rmd/

Appendices

Attachment Content