Bachelor's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Evaluation of Query Expressions in Relational Algebra

Lukáš Kotlík

Supervisor: RNDr. Martin Svoboda, Ph.D.
Field of study: Open Informatics
May 2021

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Kotlík Lukáš** Personal ID number: **483719**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Evaluation of Query Expressions in Relational Algebra

Bachelor's thesis title in Czech:

Vyhodnocování výrazů dotazů v relační algebře

Guidelines:

Based on the analysis of the existing approaches, their identified advantages and disadvantages, as well as the consideration of various notations and query constructs of the traditional formal relational algebra query language for the relational database model, the objective of this thesis is to propose and implement a web application that would allow for the evaluation of relational algebra query expressions with respect to reasonably small sample datasets, putting the emphasis on the extent of functionality, user-friendliness, and visual interface.

Bibliography / sources:

1. Relational algebra, PDF lecture <https://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/lectures/Lecture-07-Relational-Algebra.pdf>
2. JavaScript <https://tc39.es/ecma262/>
3. TypeScript <https://www.typescriptlang.org/docs/handbook/>

Name and workplace of bachelor's thesis supervisor:

RNDr. Martin Svoboda, Ph.D., MFF

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **04.03.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **19.02.2023**

RNDr. Martin Svoboda, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student's signature

Acknowledgements

I want to thank my supervisor RNDr. Martin Svoboda, Ph.D., for help and suggestions he provided to me as well as all students who helped with application testing. Last but not least, I want to thank my family and friends for their support.

Declaration

I hereby declare that I have elaborated this thesis on my own and listed all literature I used.

Prague, May 19, 2021

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

Praha, 19. května 2021

Abstract

Relational algebra is a formal query language over the relational model giving relational database systems solid and formally well-defined foundations. In this bachelor thesis, we designed and implemented a web application that serves as a tool for easing its learning. The application evaluates relational algebra query expressions over small sample datasets that can be edited directly in the application. We put emphasis on user-friendliness, e.g., detailed description and highlighting of errors, or visualization of evaluation trees. The application also allows for batch processing of projects, such as homework assignments.

Keywords: relational algebra, query evaluation, expression parsing, web application

Supervisor:
RNDr. Martin Svoboda, Ph.D.
Department of computer science
Faculty of Electrical Engineering
Czech Technical University in Prague
121 35 Praha 2

Abstrakt

Relační algebra je formální dotazovací jazyk nad relačním modelem dávající relačním databázovým systémům pevné a formálně dobře definované základy. V této bakalářské práci jsme navrhli a naimplementovali webovou aplikaci, která slouží jako nástroj usnadňující její učení. Aplikace vyhodnocuje výrazy dotazů relační algebry nad malými vzorovými daty, která lze přímo v aplikaci upravovat. Kladli jsme důraz na uživatelskou přívětivost, např. na přesné popisy chyb a jejich zvýraznění, nebo zobrazení evaluačních stromů. Aplikace dále umožňuje hromadné zpracování projektů jako např. domácích úkolů.

Klíčová slova: relační algebra, vyhodnocování dotazů, parsování výrazů, webová aplikace

Překlad názvu: Vyhodnocování výrazů dotazů v relační algebře

Contents

Chapter 1

Introduction

Relational algebra is a formal basis for relational databases introduced by Edgar Frank Codd in 1972. Although we do not use relational algebra directly in practical problems, database systems still use its constructs in the background for query evaluation. Nowadays, people often first encounter SQL as the leading query language and might find relational algebra operations confusing.

Students of computer science learn relational algebra in database systems courses as it is often their fundamental part. However, there are not many tools for relational algebra learning. Moreover, existing tools do not cover all relational algebra extensions as there are several sets of defined operations or several notations.

One of the not yet sufficiently covered notations is the simplified one that students learn in database courses not just at FEE, CTU. This notation is suitable for algebra learning as it is easy to write both by hand and on a computer. The absence of such a tool is one of the reasons why we decided to implement our application.

Objectives. The main goal of this thesis is to design and implement an application that would evaluate query expression in relational algebra. First, we will analyze existing solutions and propose the application specification. During the development, we will also create programming and user documentation and perform application testing.

The implemented application is supposed to evaluate relational algebra expressions and visualize evaluation trees. Moreover, we will put emphasis on user-friendliness, e.g., convenient graphical interface, wide customization possibilities, explicit error descriptions, or many import/export possibilities. These features should make our application a suitable teaching tool. We list the main goals below:

- user-friendly graphical interface
- definition of custom relational data
- evaluation of relational algebra query expressions
- explicit description of errors

- visualization of evaluation trees

Thesis Outline. Finally, we briefly introduce each thesis chapter:

- *Relational Algebra* – In Chapter 2, we introduce relational algebra. We compare its major modifications and specify its definition for this thesis. Then, we present considered operations with several examples. Finally, we also briefly introduce another formal query languages for relational database systems – relational calculi.
- *Existing Solutions* – In Chapter 3, we analyze existing applications for relational algebra evaluation. We introduce three applications and provide examples of their usage. Then, we list their advantages and disadvantages and compare them.
- *Specification* – In Chapter 4, we provide analysis for our implementation. First, we list the requirements and present intended business processes. Then, we describe business entities and the high-level concept. Finally, we provide detailed use cases and the implementation class model.
- *Documentation* – In Chapter 5, we present technologies we used in the implementation. Then, we describe the hierarchy of code packages and the challenges we encountered in the implementation. Finally, we describe application testing and deployment.
- *User Documentation* – In Chapter 6, we provide a brief user manual with application screenshots.

Chapter 2

Relational Algebra

Relational algebra is a formal query language for relational databases. It was introduced by British-American data scientist *Edgar Frank Codd*[1] in 1972. Further researchers extended the original definition of algebra many times so that there exist several versions.

The data in relational algebra is represented as *relations*. A relation is a set of tuples $\{(a_1, v_1), ..., (a_n, v_n)\}$, where $a_i \in A_R$ are attribute names, $v_i \in D_i$ are attribute values, and D_i are attribute domains (sets of possible attribute values). This structure of relations is called *relational model*.

Each relation has a schema: $X\{a_1 : D_1, ..., a_n : D_n\}$, where X is a relation name, $a_i \in A_R$ are attribute names, and D_i are usually omitted attribute domains. The relational schema is important for relational operations as they may require its specific form[2].

To represent the data and the schema of a relation together, we use the following notation: $\langle R, A_R \rangle$, where $A_R = \{a_1, ..., a_n\}$ is a set of attributes and $R = \{(a_1, v_1), ..., (a_n, v_n)\}$ is a set of data tuples. We will use it for formal description of semantics of relational algebra operations.

We often visualize relations as tables, where columns are attributes and rows are data tuples[3]. The formal model is defined using sets so that it does not allow duplicates nor ordering of attributes and data tuples.

Relational algebra states *relational completeness*. A given query language is *relationally complete* if and only if it can express all operations of relational algebra or possibly even more. For example, SQL[4] is relationally complete.

In most cases, we use relational algebra to describe the data retrieval, but there are more use cases[5]. For example:

- description of data to be updated in a database (i.e., inserted, modified, or deleted)

[1] Codd was born in England in 1923 but later lived in the USA, where he died in 2003. More about his life in a short biography by C. J. Date, accessible on `https://amturing.acm.org/award_winners/codd_1000892.cfm`.

[2] For example, set operations in relational algebra require source relations with equal sets of attributes. For more information about relational algebra operations, see Section 2.2.

[3] In the thesis, we will use words "attribute"/"column" and "tuple"/"row" interchangeably.

[4] SQL (Structured Query Language) is the most used language in relational databases. We can use it for both data definition and manipulation.

[5] Further use cases are presented in [1], pages 192–193.

- definition of integrity constraints (i.e., properties of the data that must be held in all consistent states of the database)

For deeper information about relational algebra, we recommend Chapter 7 of *An Introduction to Database Systems* [1] by British author *C. J. Date*. We will often mention its particular parts in the following sections.

2.1 Extensions

In this section, we will go through syntactic and semantic differences in relational algebra extensions.

Operations. In the original algebra, Codd defined eight operations: *restriction* (we call it *selection*), *projection, union, intersection, difference, Cartesian product, natural join,* and *division*.

We will define the mentioned eight operations and eleven additional ones: *rename, left/right semijoin, left/right antijoin, theta join, left/right theta semijoin,* and *full/left/right outer join*. We can derive all of them from six fundamental ones: *projection, selection, rename, Cartesian product, union,* and *difference*.

For more information about particular operations, see Section 2.2.

Null Values. An important option is the support of `null` values. The model can require all values in rows to be specified or support `null` meta values to mark absent values. We chose to support `null` values in the assumed relational algebra definition as real-world relational databases assume them, too.

Attribute Reference. Relational algebra defined by Codd referenced attributes by their positions. Such definition is complicated for users and needs an ordered attribute set. To solve this issue, we need to refer to attributes by their name. However, a new issue appears when we join two relations with a common attribute name.

There exist two *attribute naming conventions* in relational algebra – we can use attribute names only or involve a relation name as their prefix. In both of them, we need a possibility to change names[6]. The convention without relation prefixes uses the *attribute rename* operation. The second convention uses the *relation rename* operation.

In the assumed definition, we use the first convention, i.e., attribute names only and so with attribute rename operation.

Operator Notation. There are three main notations of relational algebra operators. To distinguish them, we call them *textual, formal,* and *simplified* in this thesis. In the textual notation[7], we use words to describe the operations.

[6] The addition of the rename operation is discussed in Section 7.2 in [1].

[7] The textual notation is used in [1] to write expressions in a friendly way for people who do not know relational algebra.

The formal notation uses many special symbols (e.g., Greek letters or $\bowtie$), places unary operators before their operands, and uses subscript for operator parameters. The simplified notation has fewer special symbols (e.g., no greek letters) and puts unary operators after their operands. We compare the notations on the following query – *join cars with their owners*[8]:

- textual:
  (Owner RENAME id AS owner) JOIN (Car WHERE color = "Blue")
- formal: $\rho_{owner/id}(\text{Owner}) \bowtie \sigma_{color="Blue"}(\text{Car})$
- simplified: $\text{Owner}\langle\text{id} \rightarrow \text{owner}\rangle * \text{Car}(\text{color} = \text{"Blue"})$

Value Atomicity. We demand all values in relations to be *atomic*, i.e., data tuples cannot contain sequences, objects, or nested relations[9]. There exist operations to work with non-atomic values[10], but we do not define them.

SQL Features. The only widely used relational language is *SQL (Structured Query Language)*. It does not follow all original algebra restrictions, e.g., it supports duplicate data rows or ordering. Some extensions of relational algebra add new features to enable it to express all SQL operations, e.g., aggregation functions[11]. For this thesis, we do not define these extensions.

Operation Precedence. In a basic definition, we always need to use parentheses around operators to determine the order of their evaluation. To make the notation simpler, we define precedence and associativity of operations. All relational algebra operations are left-to-right associative, but there is no widely accepted opinion on their precedence.

We thought of two possible definitions of precedence values. The first one was to present eight precedence levels of operations and assign lower precedence values to the operations we usually use as final ones (e.g., division, outer joins, or set operations). These values would cause those operations to be evaluated at the end of the execution automatically. On the other hand, the second possibility was to define only four precedence levels and let the users control the evaluation order mostly by parentheses.

Finally, we chose the second possibility and use the following precedence levels (listed from the highest to the lowest):

- projection, selection, rename
- Cartesian product, natural join, left/right semijoin, left/right antijoin, theta join, left/right theta semijoin, full/left/right outer join, division

[8] We assume to have a relation *Owner* with an *id* attribute and a relation *Car* with *color* and *owner* attributes, where *Car.owner* refers to *Owner.id*.

[9] In other words, we require the *first normal form*.

[10] For more information, see Section 7.9 of [1].

[11] An aggregation function computes a new value from the input set, e.g., count or average value. More about aggregation in relational algebra (i.e., Extend and Summarize operations) in [1], pages 197–202.

- intersection
- union, difference

These values also follow some existing definitions[12,13] or implementations[14,15].

Summary. In the following list, we sum up the assumed relational algebra definition:

- relational model with `null` values
- no duplicate data tuples and attributes
- no ordering of data tuples and attributes
- only atomic values in data tuples
- attribute names without relation prefixes
- no aggregation functions

2.2 Constructs

In the following definitions of assumed relational algebra operations, we will show both formal and simplified notations. We will describe restrictions on input relations, the results[16] and show examples for some operators. In most examples, we will use relations *Car* and *Owner*, displayed in Tables 2.1 and 2.2.

id	owner	color	weight
1	1	Blue	1000
2	1	Green	1200
3	2	Blue	900
4	3	Black	1100

Table 2.1: Relation **Car**. We assume that the *owner* attribute refers to the *id* attribute of the *Owner* relation (i.e., it is a foreign key).

[12] Definition by Juliana Freire on The University of Utah, lecture slide 55, `https://my.eng.utah.edu/~cs5530/Lectures/relational-algebra-cs.pdf`.

[13] Definition by Ramon Lawrence on The University of British Columbia, lecture slide 66, `https://people.ok.ubc.ca/rlawrenc/teaching/304/Notes/304_3_Relational.pdf`.

[14] RelaX calculator (we will analyze it in Section 3.2), `http://clotho.uom.gr/relax/help.htm#relalg-operator-precedence`.

[15] A difference in SQL implementation is that it also has the same value for all set operations, `https://www.ibm.com/docs/en/informix-servers/14.10/14.10?topic=statement-set-operators-in-combined-queries`.

[16] In descriptions of results, we will use $E_S = \langle S, A_S \rangle$ as an operand for unary operators, and $E_L = \langle L, A_L \rangle$ and $E_R = \langle R, A_R \rangle$ as left-hand and right-hand operands for binary operators. The result descriptions are taken from the Relational Algebra lecture by Martin Svoboda [2].

id	name
1	George
2	Adam
3	Michael
4	Joe

Table 2.2: Relation **Owner**.

Projection. We start with unary operators. *Projection* takes a relation and preserves only a given subset of original attributes.

- Formal notation: $\pi_{a_1,\ldots,a_n}(E_S)$
- Simplified notation: $E_S[a_1, \ldots, a_n]$
- Result:
  $\pi_{a_1,\ldots,a_n}(E_S) = \langle\{\{(a, v) \mid (a, v) \in t, a \in \{a_1, \ldots, a_n\}\} \mid t \in S\}, \{a_1, \ldots, a_n\}\rangle$

For example, we want to receive all colors of cars in our data. The expression is $Car[color]$ with the result[17]:

color
Blue
Green
Black

Selection. Unlike projection, *selection* does not change the attribute set of the input relation but it selects a subset of data tuples. It accepts a condition θ that computes a boolean value for a given row. Selection preserves data tuples where the condition θ is evaluated to true.

- Formal notation: $\sigma_\theta(E_S)$
- Simplified notation: $E_S(\theta)$
- Result: $\sigma_\theta(E_S) = \langle\{t \mid t \in S \wedge \theta(t)\},\ A_S\rangle$

As example of selection, we want to receive all cars of the owner with id 1. The expression is $Car(owner = 1)$ with the result:

id	owner	color	weight
1	1	Blue	1000
2	1	Green	1200

[17] Of course, the result contains the row *Blue* only once.

Rename. The last unary operator is *rename of attributes*. It preserves all data tuples and attributes, but it changes the names of certain selected attributes.

- Formal notation: $\rho_{b_1/a_1,...,b_n/a_n}(E_S)$
- Simplified notation: $E_S\langle a_1 \rightarrow b_1, ..., a_n \rightarrow b_n\rangle$
- Result: $\rho_{b_1/a_1,...,b_n/a_n}(E_S) = \langle\{\{(a,v) \mid (a,v) \in t,\ a \notin \{a_1,...,a_n\}\} \cup \{(b_i,v)|(a_i,v) \in t, i \in \{1,...,n\}\} \mid t \in S\}, (A_R \backslash \{a_1,...,a_n\}) \cup \{b_1,...,b_n\}\rangle$

We can use rename, for example, to change the id and owner attribute names to be more explicit. The expression is $Car\langle id \rightarrow carId, owner \rightarrow ownerId\rangle$ with the result:

carId	ownerId	color	weight
1	1	Blue	1000
2	1	Green	1200
3	2	Blue	900
4	3	Black	1100

Union. *Union* is the first of three set operations we define. All set operations require input relations with the *same attribute sets*. The result of set union contains all data tuples that exist at least in one input relation.

- Both notations: $E_L \cup E_R$
- Operand restrictions: equal attribute sets $(A_L = A_R = A)$
- Result: $E_L \cup E_R = \langle L \cup R,\ A\rangle$

Assume we have two relations with the same attributes, e.g., *OldCar* and *LargeCar*. We can use set union *OldCar* $\cup$ *LargeCar* to receive all cars in one relation.

Difference. *Difference* takes two relations with the same attributes and returns data tuples that exist in the first relation but not in the second one.

- Both notations: $E_L \setminus E_R$
- Operand restrictions: equal attribute sets $(A_L = A_R = A)$
- Result: $E_L \setminus E_R = \langle L \setminus R,\ A\rangle$

Intersection. Intersection is the last set operator in relational algebra. It is not fundamental as we can derive it using set difference. It takes two relations with the same attributes and returns data tuples that exist in both of them.

- Both notations: $E_L \cap E_R$
- Operand restrictions: equal attribute sets $(A_L = A_R = A)$
- Result: $E_L \cap E_R = \langle L \cap R,\ A\rangle$

Cartesian Product. The *Cartesian product* is the last fundamental relational algebra operation. It takes two relations with *disjoint attribute sets* and returns all combinations of their data tuples. The result has $|A_L| + |A_R|$ attributes and $|L| * |R|$ data tuples.

- Both notations: $E_L \times E_R$
- Operand restrictions: disjoint attribute sets ($A_L \cap A_R = \emptyset$)
- Result: $E_L \times E_R = \langle \{t_L \cup t_R \mid t_L \in L,\ t_R \in R\},\ A_L \cup A_R \rangle$

Usually, we use the Cartesian product to create all combinations of data and then select and project intended parts. Later, we will present derived binary operators that provide advanced possibilities. If we want to select ids of all cars with added owner names, we can create the expression $(Car \times Owner\langle id \to ownerId \rangle)(owner = ownerId)[id, name]$ with the result:

id	name
1	George
2	George
3	Adam
4	Michael

Natural Join. *Natural join* takes two relations and returns combinations of their data tuples based on equality of values associated with shared attributes. Unlike the Cartesian product, it has no Operand restrictions. If there is no common attribute in input relations, it acts like a Cartesian product.

- Formal notation: $E_L \bowtie E_R$
- Simplified notation: $E_L * E_R$
- Result: $E_L \bowtie E_R =$
  $\langle \{t_L \cup t_R \mid t_L \in L,\ t_R \in R, \forall a \in A_L \cap A_R : t_L(a) = t_R(a)\},\ A_L \cup A_R \rangle$

We use the same example as for the Cartesian product – with natural join, we save an explicit selection of matched rows, but we must rename attributes so the common one is $Car.owner/Owner.id$. The expression is $(Car * Owner\langle id \to owner \rangle)[id, name]$ with the same result as in the previous example.

Semijoin. *Left and right semijoins* are modifications of the natural join, which only return attributes from one input relation. In other words, it returns data tuples from the left-hand or right-hand input relation that natural join would join.

- Formal notation: $E_L \ltimes E_R$ (left), $E_L \rtimes E_R$ (right)
- Simplified notation: $E_L <* E_R$ (left), $E_L *> E_R$ (right)

- Result of left semijoin:
  $E_L \ltimes E_R = \langle \{t_L \mid t_L \in L,\ \exists t_R \in R : \forall a \in A_L \cap A_R : t_L(a) = t_R(a)\},\ A_L \rangle$
- Result of right semijoin:
  $E_L \rtimes E_R = \langle \{t_R \mid t_R \in R,\ \exists t_L \in L : \forall a \in A_L \cap A_R : t_L(a) = t_R(a)\},\ A_R \rangle$

Theta Join. *Theta join* joins two relations based on the given condition. As the Cartesian product, it requires input relations with *disjoint attribute sets*.

- Formal notation: $E_L \bowtie_\theta E_R$
- Simplified notation: $E_L[\theta]E_R$
- Operand restrictions: disjoint attribute sets ($A_L \cap A_R = \emptyset$)
- Result: $E_L \bowtie_\theta E_R = \langle \{t \mid t = t_L \cup t_R,\ t_L \in L,\ t_R \in R,\ \theta(t)\},\ A_L \cup A_R \rangle$

We use the same example as for the Cartesian product and natural join. We must rename attributes to be disjoint sets. The expression is $(Car[owner = ownerId]Owner\langle id \rightarrow ownerId \rangle)[id, name]$ with the same result as in the previous examples.

Theta Semijoin. Similarly to (natural) semijoins, *left and right theta semijoins* return data tuples from one input relation that would be joined by theta join.

- Formal notation: $E_L \ltimes_\theta E_R$ (left), $E_L \rtimes_\theta E_R$ (right)
- Simplified notation: $E_L\ \langle\theta]\ E_R$ (left), $E_L\ [\theta\rangle\ E_R$ (right)
- Operand restrictions: disjoint attribute sets ($A_L \cap A_R = \emptyset$)
- Result of left theta semijoin:
  $E_L \ltimes_\theta E_R = \langle \{t_L \mid t_L \in L,\ \exists t_R \in R :\ \theta(t_L \cup t_R)\},\ A_L \rangle$
- Result of right theta semijoin:
  $E_L \rtimes_\theta E_R = \langle \{t_R \mid t_R \in R,\ \exists t_L \in L :\ \theta(t_L \cup t_R)\},\ A_R \rangle$

Antijoin. *Left and right antijoins* are opposites to semijoins: they return data tuples from one input relation that natural join would *not* join.

- Both notations: $E_L \triangleright E_R$ (left), $E_L \triangleleft E_R$ (right)
- Result of left antijoin:
  $E_L \triangleright E_R = \langle \{t_L \mid t_L \in L,\ \nexists t_R \in R : \forall a \in A_L \cap A_R : t_L(a) = t_R(a)\},\ A_L \rangle$
- Result of right antijoin:
  $E_L \triangleleft E_R = \langle \{t_R \mid t_R \in R,\ \nexists t_L \in L : \forall a \in A_L \cap A_R : t_L(a) = t_R(a)\},\ A_R \rangle$

For example, we use (right) antijoin to select all owners without a car in our data. The expression is $Car \triangleleft Owner\langle id \to owner\rangle$ with the result:

owner	name
4	Joe

Division. *Division* preserves all uncommon attributes of data tuples from the left-hand relation whose all combinations with data tuples from the right-hand relation exist in the left-hand relation[18]. Its behavior is similar to a universal quantifier. It requires *the right-hand attribute set to be a proper subset of the left-hand one.*

- Both notations: $E_L \div E_R$
- Restrictions: right attributes are a proper subset of left attributes $(A_R \subset A_L)$
- Result: $E_L \div E_R = \langle\{t \mid \forall t_R \in R : (t \cup t_R) \in L\},\ A_L \setminus A_R\rangle$

Assume we have created a relation *Colors(color)* with two rows *Green* and *Blue*. To receive ids of owners who own cars of all colors in Colors relation, we write expression $Car[owner, color] \div Colors$ with the result:

owner
1

Outer Join. As we assume the relational model with `null` values, we can also define *outer joins*. Outer join naturally joins input relations and further adds naturally unjoinable data tuples complemented by `null` values. There are three types of outer joins: *full, left,* and *right.*

- Formal notation: $E_L ⟗ E_R$ (full), $E_L ⟕ E_R$ (left), $E_L ⟖ E_R$ (right)
- Simplified notation: $E_L *_F E_R$ (full), $E_L *_L E_R$ (left), $E_L *_R E_R$ (right)
- Restrictions: relational model with `null` values
- Result of left outer join: $E_L ⟕ E_R =$ $\langle\{t_L \cup t_R \mid t_L \in L,\ t_R \in R,\ \forall a \in A_L \cap A_R : t_L(a) = t_R(a)\} \cup$ $\{t_L \cup \{(r, null) \mid r \in A_R\} \mid t_L \in L, \nexists t_R \in R : \forall a \in A_L \cap A_R :$ $t_L(a) = t_r(a)\},\ A_L \cup A_R\rangle$
- Result of right outer join: $E_L ⟖ E_R =$ $\langle\{t_L \cup t_R \mid t_L \in L,\ t_R \in R,\ \forall a \in A_L \cap A_R : t_L(a) = t_R(a)\} \cup$ $\{\{(l, null) \mid l \in A_L\} \cup t_R \mid t_R \in R, \nexists t_L \in L : \forall a \in A_L \cap A_R :$ $t_L(a) = t_r(a)\},\ A_L \cup A_R\rangle$

[18] We can define division in other ways, e.g., as a ternary operator, see [1], page 188.

- Result of full outer join: $E_L ⟗ E_R = E_L ⟖ E_R \cup E_L ⟕ E_R$

For example, we can use full outer join to join cars and their owners and not to lose any data tuple. The expression is $Car *_F Owner\langle id \rightarrow owner\rangle$ with the result:

id	owner	color	weight	name
1	1	Blue	1000	George
2	1	Green	1200	George
3	2	Blue	900	Adam
4	3	Black	1100	Michael
null	4	null	null	Joe

2.3 Relational Calculus

Relational algebra is not the only formal query language for relational databases. In this section, we will briefly present *relational calculus*. For more details, we recommend Chapter 8 of the book by Date [1].

Relational calculus is a *declarative* language in contrast to *procedural* relational algebra. Calculus expressions might be more straightforward, but they do not describe any procedure to retrieve the data. To execute them, we need to transform them into algebraic expressions first[19].

There are two types[20] of relational calculi: *tuple* and *domain*. Both types use variables to represent values in relations. In tuple calculus, variables are bound with values of particular data tuples from relations. In domain calculus, they are associated with values from individual attribute domains. Both types of relational calculi are *relationally complete*.

Finally, we show examples of tuple calculus expressions. We assume to have the same relations as for algebraic operation examples (see Tables 2.1 and 2.2). Now we can select ids and colors of cars with owner id 1:

```
{(c.id, c.color) | c ∈ Car, c.owner = 1}
```

Or retrieve names of owners who have no car in our data:

```
{(o.name) | o ∈ Owner, ∄c ∈ Car: c.owner = o.id}
```

[19] An example of such an algorithm is shown in Section 8.4 of [1].
[20] Both types are described in Sections 8.2 and 8.7 of [1].

Chapter 3

Existing Solutions

In this chapter, we will go through a few existing applications, which deal with the evaluation of relational algebra expressions. First, we will describe each approach in a single section, so that all of them will then be shortly compared at the end.

The goal of this analysis is to find common features, positives, and negatives of these solutions and to learn how our final application should work and look like.

3.1 Raeval

Raeval (Relational Algebra Evaluator) [3] is an application free to download from the project website. We will shortly analyze version 2.0 (Beta) from April 20, 2011, and version 0.3.1 from September 27, 2012.

Relational Algebra Evaluator is an interactive textual application[1], which evaluates relational algebra expressions over relations. We upload the relations from text files in the CSV[2] format. We use English words to write operations in the expressions, i.e., it uses the *textual* notation.

Advantages. The first advantage we mention is the possibility to save results into new, dynamically defined relations. We can use them in other expressions in the same way as relations loaded from files. We can open an editing table by the `edit` keyword and edit the loaded relations, i.e., change their values or add new rows.

The application supports nested operations by the usage of parentheses. Also, we can split expressions into multiple lines or use other whitespaces[3] to format them visually.

[1] Textual application (or textual user interface) uses text only to communicate with a user, i.e., it displays textual information and receives commands as textual inputs. On the other hand, the graphical application displays pictures, tables, animations, etc., and provides buttons and other input possibilities for the user.

[2] CSV (comma separated values) is a simple format of text files that represents tables, it uses end of lines to separate rows and commas (or semicolons or tabulators) to separate values in a row.

[3] A whitespace is a character that is not displayed but it helps to format the text, e.g., space, tabulator, or newline.

Finally, Raeval supports command history to reuse previous commands.

Disadvantages. On the other hand, it is not possible to define brand new relations in the application. Also, we cannot save the evaluated results into text files to use them in further application runs.

We have encountered several errors when uploading source relations from CSV files, specifically when using commas or quotes in string columns. The application expects that a comma separates columns in the CSV format, and quotes are optional enclosing of textual inputs. Unfortunately, we were not able to find a way to use commas nor quotes in string values. Both possibilities of textual inputs (with or without optional quotes) do not support these characters. Also, we cannot escape their functionality by a backslash[4]. The application always parses commas as separators, and it can result in errors when uploading files, which contain commas in string values. Quotes do not result in any error, but the application ignores them so that they are missing in the uploaded rows.

Similarly, language-specific characters are not supported. For example, accented Czech letters (e.g., *á, č, ř*) are loaded from a file but displayed incorrectly. Moreover, the application does not match them with values typed in the application.

Next, error messages are not much expressive, which means that debugging becomes harder in more complicated expressions. We show an example of an error message in Figure 3.1. There is a misspelling of a column name *Color* described as *Semantic error: expression does not evaluate to a boolean value, " instead.*

The last disadvantage is the impossibility of copying the text from the application.

Versions comparison. Version 0.3.1 brings a new user interface, which is more pleasant in colors but still only textual. New functionality is the logging[5] of results into text files. This logging saves a given expression and its result, but it is impossible to trace source relations. We can change the output file of logging at runtime[6]. Moreover, logging output files are not compatible with the input file format, which means we cannot load the saved results back in the application.

Unfortunately, there are some features, which are worse than in the previous version. For example, the formatting of displayed relations in the application is worse (columns are not aligned). Also, version 0.3.1 does not support the `source` keyword, which enables the loading of multiple relations at once.

Example of Usage. Firstly, we need to create text source files from which we load the relations. We assume we have two files, *car.txt* and *owner.txt*, both in the CSV format. In *car.txt*, we define four columns: `Id, Owner,`

[4] Using backslash as an escape character is a standard approach in many programming situations, e.g., in regular expressions or string literals in Java, JavaScript, C, etc.

[5] logging is an automatic process of recording the information about the application run

[6] we do not need to terminate the application

`Color, Wheels`. In *owner.txt*, we define three columns: `Id, Name, Address`. The files have to have two header lines. There are defined column names in the first line and attribute domains in the second one. In each file, we define two data rows, so *car.txt* may look like this:

```
Id,Owner,Color,Wheels
id,id,name,number
1,1,Blue,4
2,2,Red,8
```

and *owner.txt* may look like this:

```
Id,Name,Address
id,name,name
1,Lukas,Praha
2,Jakub,Brno
```

Having launched the application, we can load the defined relations as follows:

```
car := load "car.txt"
owner := load "owner.txt"
```

To show the result of the expression on the standard output, we need to write and submit only the expression itself. The result will not be saved. The following command evaluates a relational algebra expression `car(Color = "Green" || Color ="Blue")`:

```
select car where (Color = "Green") or (Color = "Blue")
```

To save the result, we must specify the intended name of the result relation. The next command evaluates an expression `car*owner<Id -> Owner>` and saves it into a relation *result*:

```
result := car join (owner rename (Id as Owner))
```

We show in Figure 3.1 how the examples look like in the application. It also contains one error message when we misspell a column *Color* with lower-case *c*.

Conclusion. Relational Algebra Evaluator is a simple application with all basic functionality. Although the user interface is only textual, the application is easy to use. For teaching purposes, we miss a more expressive reporting of error messages and the support of simplified notation.

- **Author:** Nick Everitt, University of North Carolina Wilmington
- **Year:** 2012

```
> car := load "car.txt"
Id   Owner Color Wheels
id   id    name  number
1    1     Blue  4
2    2     Red   8


> owner := load "owner.txt"
Id   Name  Address
id   name  name
1    Lukas Praha
2    Jakub Brno


> select car where (color = "green") or (color = "blue")
Semantic error: expression does not evaluate to a boolean value, '' instead

> select car where (Color = "Green") or (Color = "Blue")
Id   Owner Color Wheels
id   id    name  number
1    1     Blue  4


> result := car join (owner rename (Id as Owner))
Id   Owner Color Wheels Name  Address
id   id    name  number name  name
1    1     Blue  4      Lukas Praha
2    2     Red   8      Jakub Brno
```

Figure 3.1: Raeval usage example

- **Application type:** desktop Java application
- **Application availability:** free download of an executable `.jar` file from the web page
- **User interface:** textual user interface
- **Advantages:**
  1. Relations can be loaded from pre-defined files.
  2. Results can be saved into new variables.
  3. There is a possibility of an automatic logging of used expressions and their results.
- **Disadvantages:**
  1. Error messages are not much expressive.
  2. Relations cannot be created in the application.
  3. Relations cannot be saved in the input file format.

3.2 RelaX

RelaX (Relational Algebra Calculator) [4] is an online relational algebra calculator. We will analyze version 0.20, which is the current version in October 2020. RelaX is an interactive web application, which evaluates relational algebra expressions and translates SQL query expressions into the syntax of relational algebra. We can define relations right in the application or load them from files or public online sources. Operations are written in the formal notation.

Advantages. The application has many useful features; some of them are even beyond relational algebra. Because RelaX uses formal notation, it provides more user-friendly options than writing Greek letters on a keyboard. The first option is to include Greek letters by clicking a button. The second one is to use English transcriptions of Greek letters instead of writing Greek letters right away. Our user experience showed that a better solution would be to support the keywords (projection, selection, etc.) because English transcriptions are less expressive.

Due to the educational purpose of the application, the visualization of the interactive evaluation tree[7] is useful. It is possible to show the intermediate results of each subexpression in a query expression. This possibility facilitates a faster and deeper understanding of the individual operations of relational algebra.

We can use inline relations (i.e., temporary, nameless relations) in expression formulations as well. For example, they can provide a closer description of abbreviated values in other relations (e.g., when values are stored as integers to save memory, we can replace them with textual descriptions before displaying them).

The Mobile version of the RelaX website is easy to use. It supports all the functionality of the desktop application, and the graphical interface is user-friendly.

Further advantages are closely related to SQL. The first of them is the evaluation of SQL query expressions. The application shows the result and the evaluation tree of the equivalent relational algebra expression.

We can order the results by the `order by` keyword and aggregate them by the `group by` keyword. We want to emphasize that these operations are not a part of the original relational algebra (`order by` does not respect unordered rows in a relation, and `group by` produces new values). Anyway, these keywords are well-known from SQL and make the result well-arranged.

The application implements many operations to specify column values. From a long list of operations, we mention a function `length(string)` which returns the length of a given string, or a function `string LIKE 'regex'` which checks whether the string matches the regular expression. Further

[7]The evaluation tree is a tree representation of the expression. The nodes represent individual operations, the leaves contain the input data, and the root contains the result.

interesting functions work with the date data type and extract its parts (days, years, hours, etc.).

Disadvantages. The assignment of a result of the relational algebra expression has an unintuitive behavior. We expected that it creates a new relation, which we can use in further expressions. But the assignment works as a substitution. It does not save the result into a new relation, but it replaces usages of the defined variable by the assigned expression. It enables us to shorten individual query expressions by the reuse of written subexpressions. It is a nice feature, but, unfortunately, there is no way to save the result in a new relation. Also, we cannot download the result relations in the compatible CSV format.

Relation definitions support two ways of working with textual values. We can enclose them in single or double-quotes. Anyway, textual values and regular expressions must be enclosed in single-quotes only in query expressions. That means we cannot use single-quotes inside a string literal in the application.

We can download a defined relation in a CSV file but uploading this file back to the application does not work.

The last disadvantage relates to the relational algebra definition assumed in this thesis – the application uses attribute naming convention with relation prefixes.

Example of Usage. First, we need to define source relations. We can use the *Group Editor* tab in the application. We define two relations *Car* and *Owner* with the same columns as in the example for the previous application. RelaX also requires a name of the group of relations. The definition of relations may contain the following text:

```
group: RelaXTest - Car, Owner
Car = {
    Id,Owner,Color,Wheels
    1,1,Blue,4
    2,2,Red,8
}
Owner = {
    Id,Name,Address
    1,Lukas,Praha
    2,Jakub,Brno
}
```

The defined relations are displayed with the *Preview* button and then loaded to the application with the *use Group in editor* button. We can see the loaded relations and their columns in the left part of a screen. Note that RelaX detected column types simply from values in their rows. Now we are ready to evaluate two query expressions with the loaded relations.

First, we use the Greek symbol σ for selection in the expression `Car(Color = "Green" || Color = "Blue")`:

```
σ Color = 'Green' or Color = 'Blue' Car
```

It is equivalent to the version with an English transcription:

```
sigma Color = 'Green' or Color = 'Blue' Car
```

We can comment out a previous expression with a double dash. Now, we evaluate our second expression `Car*Owner<Id -> Owner>`:

```
Car ⋈ ρ Id → Owner Owner
```

It is equivalent to the version without special characters:

```
Car natural join rho Id -> Owner Owner
```

We recommend using extra parentheses to separate logical parts of the expression for better clarity, although the application does not require them. We show the possible equivalent expressions and the result of the second query expression in Figure 3.2

Conclusion. RelaX is a user-friendly application with many supported operations beyond relational algebra. For our needs, we miss the simplified notation of operations and column names without relation prefixes.

- **Author:** Johannes Kessler, University of Innsbruck
- **Year:** 2020 (still in active development at the time of access, October 2020)
- **Application type:** web application
- **Application availability:** free access on GitHub
- **User interface:** graphical user interface
- **Advantages:**
  1. Greek letters can be inserted by a button or replaced by an English transcription.
  2. Inline relations are supported.
  3. SQL query expressions are evaluated and translated into trees consisting of relational algebra operations.
  4. Many operations beyond the relational algebra are supported, e.g., `order by`, `group by`, or `length`.
- **Disadvantages:**
  1. There is no way to save a result into a new relation.
  2. Column names use relation prefixes.

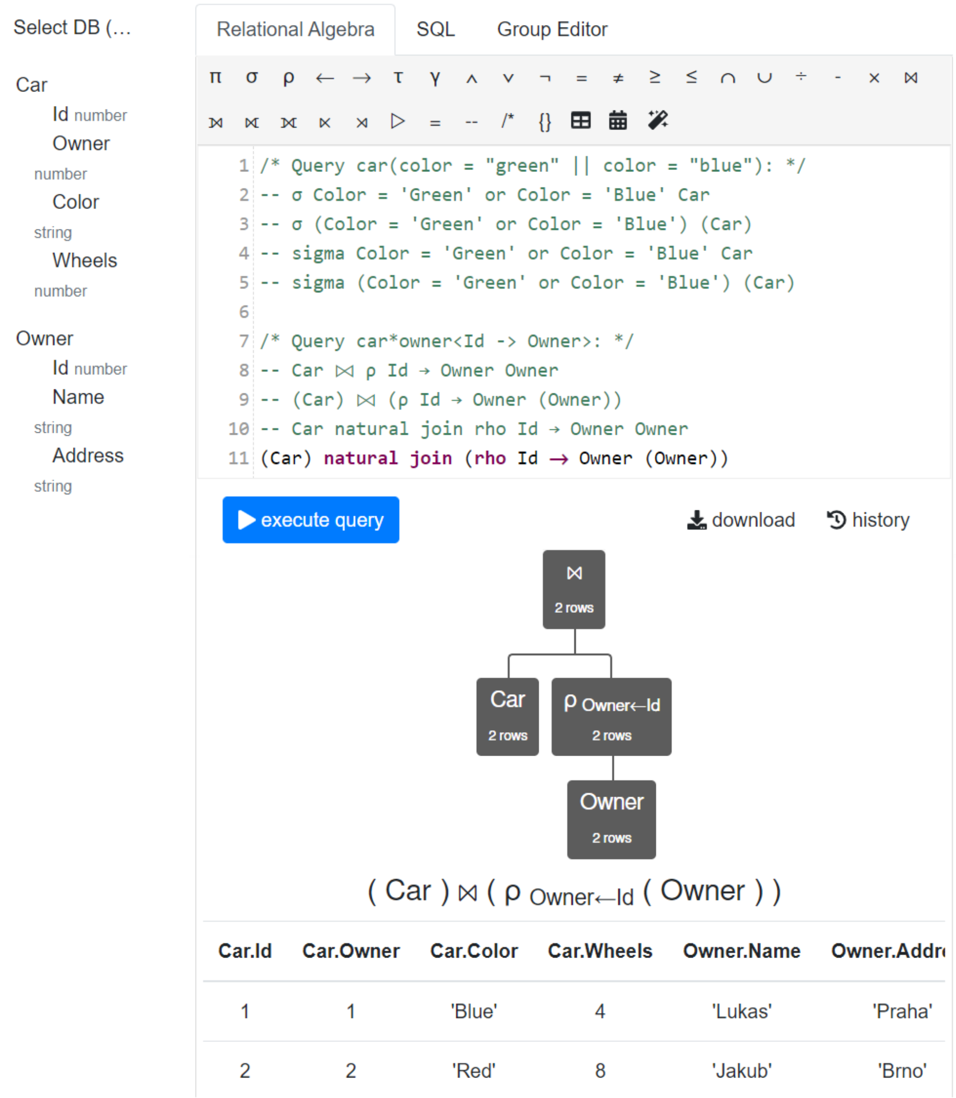


Figure 3.2: RelaX usage example

3.3 RAT

RAT (Relational Algebra Translator) [5] is a relational algebra translator free to download from the project website. We will analyze version 4.2.0.0 from the year 2011, downloaded as the newest one[8].

RAT is an interactive desktop application, which translates relational algebra expressions to SQL and evaluates them. We need to connect to an existing database to be able to evaluate expressions over the data. It uses the formal notation of operations.

[8]The application says it is version 4.2.0.0, but the download link has number 4.1.1, and the installation file has 4.3.

Advantages. Because RAT uses the formal notation, it supports more user-friendly options than writing Greek letters on a keyboard. The first option is to include Greek letters by clicking on a button. The second option is to use a keyboard shortcut, which makes the writing of expressions much faster. For example, *Ctrl+P* inserts Π for projection, or *Ctrl+S* inserts σ for selection.

Due to the educational purpose of the application, display of the evaluation tree is useful. A tree is simple, it shows the evaluation structure, and each node describes an individual operation (e.g., projected columns in projection or a condition in selection).

We can reuse once written expressions later on by assigning them to variables. We can then evaluate the defined variables, but they cannot be composed into more complex expressions. Moreover, we can save a whole group of expressions in the Query library and load them in the next application run.

An interesting feature is a connection to a database and evaluation of expressions over its data.

Disadvantages. The application loses its potential because no English documentation exists. The website of the project is in English, but the available information is not sufficient. The documentation is in Spanish, so we found the described functionality only by trial and error.

Due to the educational purpose of the application, error messages are not expressive enough. The only sign of a syntactic error is the disappearance of the translated SQL expression.

Unfortunately, there is no way to load the input data to the application without connecting to a database. We miss a possibility to import relations from CSV or text files.

The application does not support language-specific characters. It ignores them in evaluation trees and does not translate the expressions containing them to SQL.

As in RelaX, the last disadvantages is that the application uses attribute naming convention with relation prefixes.

Example of Usage. We will not use any data for this example. However, if we connected the application to a database, the application would evaluate the expression and show the result. As in the previous cases, we assume two relations. The first one is named *Car* and contains columns `Id, Owner, Color, Wheels`, the second one is named *Owner* and contains columns `Id, Name, Address`.

In Relational Algebra Statement field, we write an expression `car(color = "green" || color = "blue")` and assign it to a variable `Q1`:

```
Q1 ← σ{Color = 'Green' ∨ Color = 'Blue'}(Car)
```

In the second expression, we show the attribute naming convention with relation prefixes. We assign it to a variable `Q2`:

```
Q2 ← σ{Car.Owner = Owner.Id}(Car × Owner)
```

Now we can use the variable to evaluate the assigned expression. The application displays the SQL translation in the middle field. The evaluation tree of the expression is displayed at the bottom of the screen. If there is only a grey area instead of the tree, we use the third button on the right side (with arrows) to change the view from the results to the evaluation tree. We show how these expressions look like in the application in Figure 3.3.

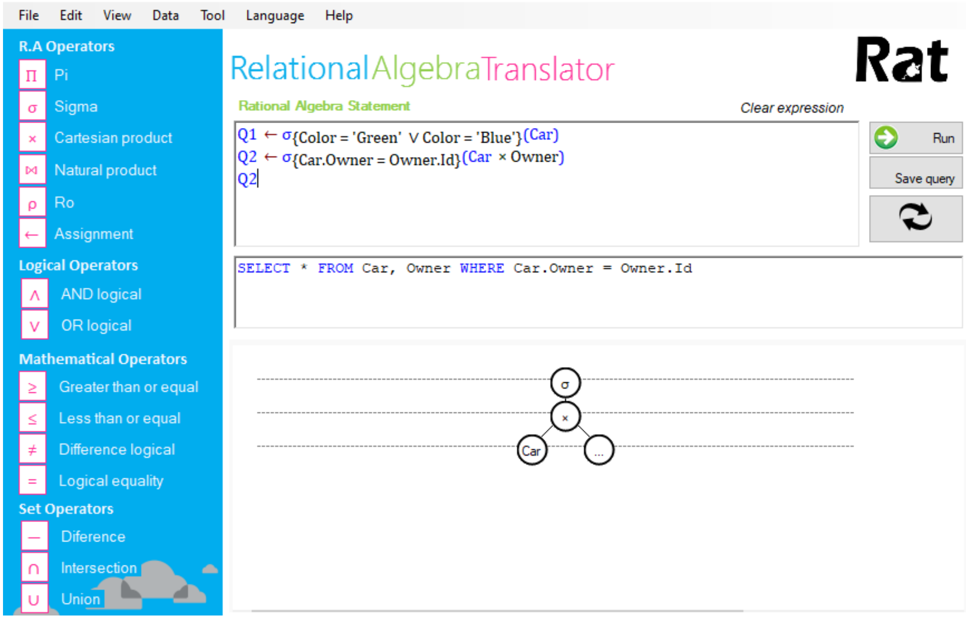


Figure 3.3: RAT usage example

Conclusion. RAT loses its potential because no English documentation exists, and so it is difficult to understand. For teaching purposes, we also miss the possibility to load relations from a file and more explicit error descriptions.

- **Author:** Steven Brenes Chavarría, National University of Costa Rica
- **Year:** 2011
- **Application type:** desktop .NET application
- **Application availability:** free download of an installation file from the web page
- **User interface:** graphical user interface
- **Advantages:**
  1. Keyboard shortcuts for insertion of special characters are supported.
  2. The evaluation tree of the expression is displayed.
  3. Expressions can be reused by assigning them to variables or storing them in the library of the application.

4. The application can connect to a database.

- **Disadvantages:**

  1. Only few relational algebra operations are implemented.
  2. There is no documentation in English.
  3. The application cannot load the input data from files.
  4. Error messages are not sufficient.

3.4 Other Solutions

We encountered a few other existing applications during our analysis. Unfortunately, we were not able to run them as there is no executable file to downloaded, or no documentation exists. The project descriptions were similar to the analyzed applications. For example, they provide relational algebra to SQL translation or describe the translation of the SQL expressions into evaluation trees.

3.5 Comparison

Although all three applications have many similarities, they show different approaches to a problem. We will use our analysis to find the best solutions to several particular aspects. Furthermore, to summarize all approaches, we created Table 3.1.

User Interface. Raeval shows that a textual user interface can be user-friendly. But users expect modern applications to have graphical user interfaces. The graphical interface of both RelaX and RAT is simple, but it is sufficient for a relational algebra evaluator. The main advantage of the graphical user interface is the possibility to display an interactive evaluation tree.

Accessibility. RelaX is the easiest application to use because it requires a web browser only. Also, it is accessible from mobile devices. We need to download other applications to a computer, which could cause possible complications. We need Java installed on a computer for Raeval and .NET Core for RAT.

Loading of Relations. The most user-friendly way to load source relations is to load them predefined from a file. Raeval and RelaX provide this functionality. RAT can connect to a database, which is not necessary for educational purposes. We miss the possibility to define a relation right in the application in Raeval and RAT.

Saving of Results. Raeval is the only application that can save the result of the expression into a new relation in the application. It is the easiest way to reuse the results in further expressions. Unfortunately, we cannot save the results to a file in any application. On the other hand, all three applications have a way to store the expressions. We can use logging in Raeval, download a text file in RelaX, or store them in a library in RAT.

Additional Features. Besides the main functionality, the covered applications present many additional features. RelaX and RAT provide a possibility to insert special characters with buttons. Also, they display an evaluation tree of the expression. RelaX translates SQL query expressions to relational algebra operations. On the other hand, RAT translates relational algebra to SQL.

	Raeval	**RelaX**	**RAT**
Application type	desktop	web	desktop
Accessibility	free download	free online usage	free download
Licence	Apache 2.0	unspecified	unspecified
Requirements	Java	web browser	.NET Core
User interface	textual	graphical	graphical
Languages	ENG	ENG, GER, SPA, KOR, POR	ENG, GER, SPA, ITA
Null values support	no	always	always
RA operations	all basic	all	all basic
Operators	textual notation	formal notation	formal notation
Alternatives to operators	no	insertion by buttons, English transcriptions	insertion by buttons or keyboard shortcuts
Relation import/export	import from CSV files	import and export using CSV files or online sources	connection to a database
Expression import/export	export to the text file	export to the text file	import and export using application library
SQL support	no	SQL to RA translation	RA to SQL translation
Error messages	sufficient	explicit, real-time	no
Multiple expression support	expression (command) history	expression history	expression library
Complex strings handling	no	yes	no
Batch processing	yes	no	no

Table 3.1: Comparison of the existing solutions

Chapter 4

Specification

In this chapter, we present the application specification. It starts with the list of requirements and their description. Then, we show main business processes[1] in UML[2] activity diagrams. We also describe the identified business entities[3] in the next section. In the Concept section, we present the general approach to the implementation. Then, we describe the use cases[4]. The last section contains the description of the implementation classes formed from business entities. In the text, we show only several important UML diagrams. We provide all models in the thesis attachment.

4.1 Requirements

In this section, we will identify the project requirements. We will split the requirements into a few groups.

Application. The application should be accessible on the internet. It should be easy to use, so there are three related requirements. The strongest one is for a simple, modern, user-friendly graphical user interface. Also, the application should support both the Czech and the English language, so Czech, as well as foreign students, could use it. The third requirement is to provide saving and loading of all the application data as a project, so the users could easily save all their work and continue in the next application run.

Relational Algebra. The application should accept the simplified syntax of relational algebra expressions. There should be support for relational algebra with `null` values as well as without them. It should provide all the operations

[1]Business process is a sequence of user actions and application reactions which the application should support, it is described in the analysis.

[2]UML (Unified Modeling Language) is used to graphically describe the specification, design, and documentation of software systems. Its specification is accessible on `https://www.omg.org/spec/UML/About-UML/`.

[3]Business entities are real-world objects that relate to the intended application in some way, they do not necessarily correspond to classes used in the final implementation.

[4]Use case is a detailed description of the particular feature which the application should support.

described in the Relational Algebra chapter (outer joins only if null values support is selected).

Query Expressions. The main purpose of the application is to define and evaluate relational algebra expressions. It should provide a user-friendly input field for the expressions. Pairs of the typed parentheses should be highlighted. Since relational algebra uses not frequently used and hard-to-write symbols, e.g., set union or set difference, the application should provide a user-friendly way to type them. The error messages should be explicit to easily find both the syntactic and semantic errors. Also, the errors should be highlighted in the input field.

The application should support work with multiple expressions at once. The user should be able to import or export them using text files.

Evaluation. After an evaluation of an expression, the application should display the evaluation tree, which helps students to understand each relational algebra operation and its semantics. The tree should be interactive and allow the user to view each intermediate relation created during the evaluation.

On the other hand, maximal computing effectiveness is not expected. The algorithms used for the evaluation of the operations can be implemented in a naive way. It is important only to reliably evaluate the expressions over units of relations and dozens of rows.

Data. The expressions should be evaluated over defined relations. The relations should be imported or exported using CSV files (each file with one relation definition). The application should support all main CSV types (i.e., CRLF/CR/LF line separators, comma/semicolon value separators). Also, the user should add the relations from the evaluation tree to defined ones. It should be possible to edit all the defined relations right in the application. The application should handle complex string values in quotes.

Batch Processing. The application should provide batch processing of multiple input files and saving the results and reports to new files. This feature enables teachers to evaluate the work of the students at once and see their results and errors.

Let us now summarize found requirements.

Application:

1. Web application
2. User-friendly GUI
3. Czech and English language support
4. Import/export of all the application data as a project

Relational algebra:

1. Expressions in simplified syntax
2. Support for relational model with or without `null` values
3. Support for all relational algebra operations

Query expressions:

1. Input field for relational algebra expressions
2. Highlighting of parentheses
3. Alternatives to typing special symbols
4. Explicit descriptive messages for both syntactic and semantic errors
5. Highlighting of errors in the input field
6. Support of multiple expressions in one project
7. Import/export of expressions

Evaluation:

1. Displaying of the evaluation tree
2. Displaying of the intermediate result for each evaluation tree node
3. Evaluation of the expressions over small data sets

Data:

1. Import/export of relations from CSV files
2. Support of all main CSV types
3. Loading relations from evaluation tree nodes
4. Editing of the data for the evaluation
5. Handling of complex string values

Batch processing:

1. Processing of multiple input files
2. Creating of the report files

4.2 Business Processes

The application has a specific purpose, so there are not many business processes to describe. There are three main ones: define relations, evaluate expressions, and select the node from the evaluation tree to display. We split the first two mentioned processes into two smaller ones each. Also, we describe the process of importing/exporting the project and batch processing.

In the *Edit relation definitions* business process, the user manipulates the relation definitions. The user can load them from CSV files, save them in CSV files, and edit them in the application. These manipulations can be done in an arbitrary order so that the business process contains a loop.

In the *Confirm relation definitions* business process, the user confirms the edited relation definitions. After that, the application parses individual relations from valid definitions. Finally, it displays a message with parsing information.

In the *Edit the expression* business process, the user manipulates the relational algebra expressions. The user can load them from a file, save them in a file, and edit them in the application. These manipulations can be done in arbitrary order so that the business process contains a loop.

In the *Evaluate the expression* business process, the user confirms the selected expression, and the application parses it to the evaluation tree (it triggers the *Use the evaluation tree* business process with the root node selected). When an error occurs, the application displays it. We show this business process in the Figure 4.1 as well.

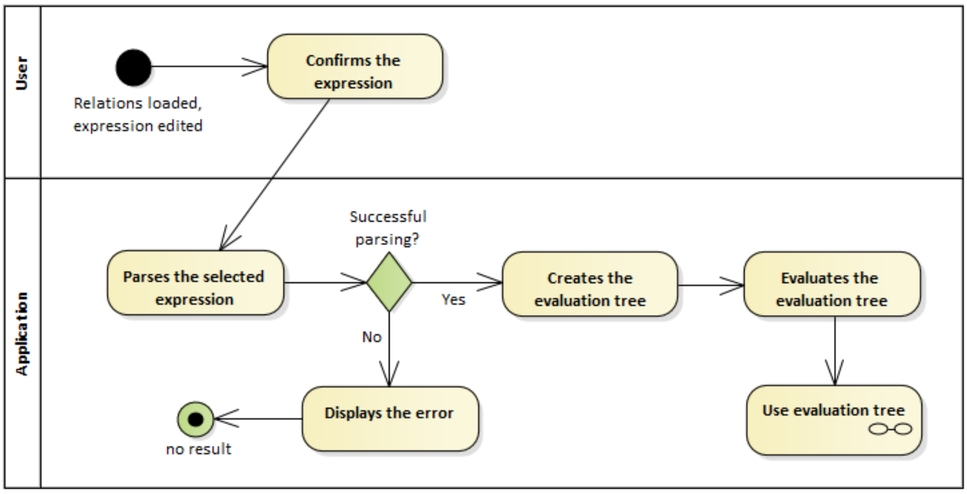


Figure 4.1: *Evaluate the expression* business process

In the *Use the evaluation tree* business process, the user selects an individual node of the evaluation tree. The application displays an intermediate relation evaluated in the node. The user can add it to the relation definitions or save it in a file. The user can change the selected node multiple times so that the business process contains a loop.

In the *Load/save the project* business process, the user loads the project data from a file or saves the current project.

In the *Process multiple project files* business process, the user selects several saved projects. For each project, the application parses relation definitions, evaluates expressions, and creates a report. Finally, it saves the reports in a zip archive.

4.3 Business Entities

In the analysis, we identified a couple of entities that exist in our intended system. We describe them in the following section and show them in UML diagrams 4.2, 4.3, and 4.4.

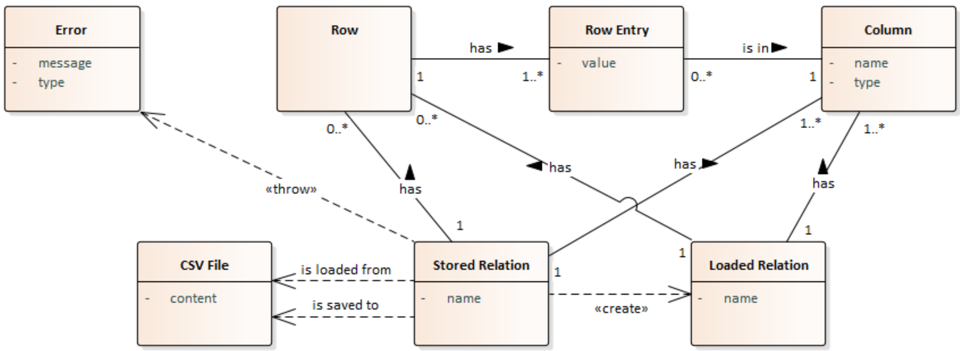


Figure 4.2: Relation and related entities

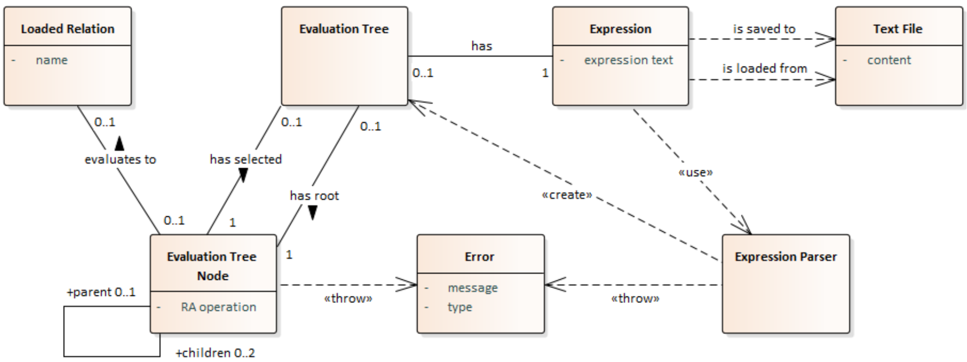


Figure 4.3: Expression and related entities

Stored and Loaded Relations. In the analysis, we found out it is advantageous to distinguish two types of relations – stored and loaded. A Stored Relation represents an editable state which might be invalid when edited. When valid, it can create a Loaded Relation. The application uses Loaded Relations for evaluation as it needs no validity checks.

Both relation types have common attributes. They have a name, at least one Column, and an arbitrary number of Rows. The difference is that we do

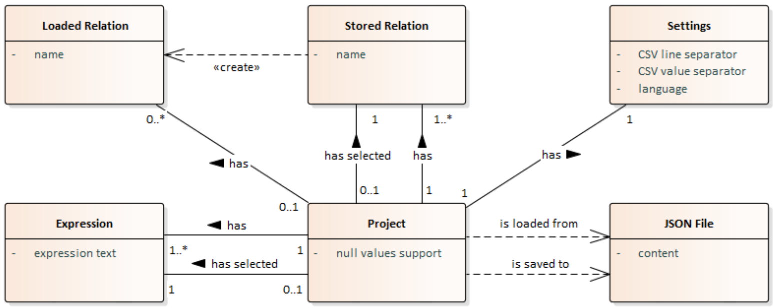


Figure 4.4: Project and related entities

not require Row Entries in the Stored Relation to have correct column types (e.g., all entries can be strings). There always is one Stored Relation selected for editing in the Project. The number of Loaded Relations is not limited. After evaluation, each Evaluation Tree Node contains a Loaded Relation as it must be valid. Stored Relations can be loaded from CSV Files or saved in them.

Column. The Column represents an attribute in the relational schema. It has a name and a type. We plan to support three column types in the application: number, string, and boolean.

Row. The Row represents a data tuple in the relational schema. Each Row belongs to a Relation and contains at least one Row Entry.

Row Entry. The Row Entry entity represents individual value in a data tuple. The Column to which it belongs determines its type.

Evaluation Tree Node. The Evaluation Tree Node represents a single relational algebra operation parsed by an Expression Parser from Expression text. It has one child if it is a unary operation, two children if it is a binary operation, or no child if it represents a relation. When being evaluated, it can throw a Semantic Error. After evaluation, it creates a result relation.

Expression Parser. We use the Expression Parser to parse the Evaluation Tree from the user input. When the input is invalid, it throws an Error.

Expression. The Expression represents one relational algebra expression defined by the user. There always exists an Expression selected for editing in the Project. Expressions can be loaded from a Text File or saved in it.

Project. The Project wraps all data in the application: Stored Relations, Loaded Relations, Expressions, null values support setting, and user Settings. The Project can be loaded from the JSON[5] File or saved in it.

[5]JSON is a transmission data format based on JavaScript object syntax. We will introduce JavaScript in Section 5.1.

Settings. The Settings describe the custom behavior of the application. They do not affect the main functionality. Specifically, they contain the used CSV separators and selected language.

Error. Invalid user inputs can trigger an error in the application. To describe them explicitly, we define a custom Error entity. We distinguish two Error types: syntactic and semantic.

CSV, Text, JSON Files. The application uses three types of files: CSV Files for storing Relations, Text Files for storing the Expressions, and JSON Files for storing the whole Project.

4.4 Concept

We intend the application as an online tool to be easily accessible from all devices and operating systems. We plan to process all data in the browser as in a single-page application. This solution avoids many complications related to the backend and the network communication. There are three main sections on the webpage: relation definition, expression definition, and result display.

Relation Definition Section. In this section, the user maintains relations. It provides a sheet where the user edits the data in the selected relation. The user can create new relations or delete existing ones. Also, the user can import or export relation definitions using CSV files.

There are three supported column types: number, string, and boolean. All entries in the column must have the same type. When null values support is off, we must specify all input values. The application highlights and describes the errors in the sheet.

If there is no error in the selected relation, the user can load it in the application and use it in the expressions. Loading a new relation overwrites a loaded one with the same name. The user can delete all the loaded relations and clear the workspace.

Expression Definition Section. In this section, the user maintains the expressions. It provides a textual input where the user edits the selected relational algebra expression. There can be several expressions in the project. They can be loaded from a text file or saved in it.

We expect the expressions to be in simplified notation. The user can use whitespaces for formating because they are ignored in the parsing. Also, to make writing easier, the user can insert operation symbols using buttons. We plan to support standard C-like one-line comments following after two slashes.

For implementation clarity, we divide the parsing into two stages. The first stage creates an evaluation tree consisting of relational algebra operations. The second stage parses parameterized RA operators, i.e., projection, selection, rename, and theta joins. When the parameter is a condition, we use an evaluation tree for representing the logic-algebraic expression. We implement existing algorithms for building the evaluation trees. We split the input into tokens and use the Shunting-Yard algorithm to create its postfix form[6]. For projection and rename parameters, we implement specific parsing as their syntax is uncommon.

The leaves of evaluation trees represent references or constants (relations in RA trees; columns or literal values in algebraic trees). Other nodes are unary or binary operations. We evaluate the trees recursively, i.e., nodes transform leaf values and propagate them to the root. Implemented RA operations use simple iterating through input rows as we expect small data inputs.

We parse the input periodically to highlight errors while the user edits the expression. Also, we check the cursor position and find suggestions. If the cursor is between RA operators, we suggest the names of all loaded relations. If the cursor is inside a parameterized RA operator, we suggest available column names. The user can use the suggestions for autocompletion.

If an error occurs in the parsing, the application highlights it in the input. If it happens after the user evaluation command, the evaluation fails. Otherwise, the application reports the error but continues in parsing to find cursor position and other errors. We provide a detailed error description as the application is a learning tool.

Result Display Section. The result section displays the result relation and the evaluation tree of the expression. We already need the tree for the evaluation, so this section only creates its visual representation.

By default, it displays the relation from the root node. The user can select a different node, and the application displays its intermediate result. The selected relation can be added to relation definitions or saved to a CSV file.

Other Features. The application provides several features that do not belong to any described section. They are project management, batch processing, and settings.

The user can import or export the current project using a JSON file. In the file, the application saves the stored relations, expressions, and the null values support. It does not save loaded relations nor evaluation trees.

The batch processing is not part of the evaluation section because the user does not edit the input, and the results are not displayed. The user selects multiple saved project files. For each project, the application loads its defined relations, evaluates its expressions, and creates a text report file. The report file contains defined relations, expressions, and evaluation results or errors. Also, it computes the number of used operations. When finished, it downloads report files in a zip archive.

The user can select the application language, the used CSV type, and the null values support. The language and CSV settings are saved in the browser storage to remain to the next application run. The null values support is a part of the current project.

[6]We describe the algorithm in depth in Section 5.3.1.

4.5 Use Cases

The next stage of the analysis is defining the use cases. The use case (UC) is a particular feature which the application provides. Use cases follow up on the requirements and describe their behavior in more detail. They identify classes for the final model. Usually, for a use case, we create a primitive design of the application screen.

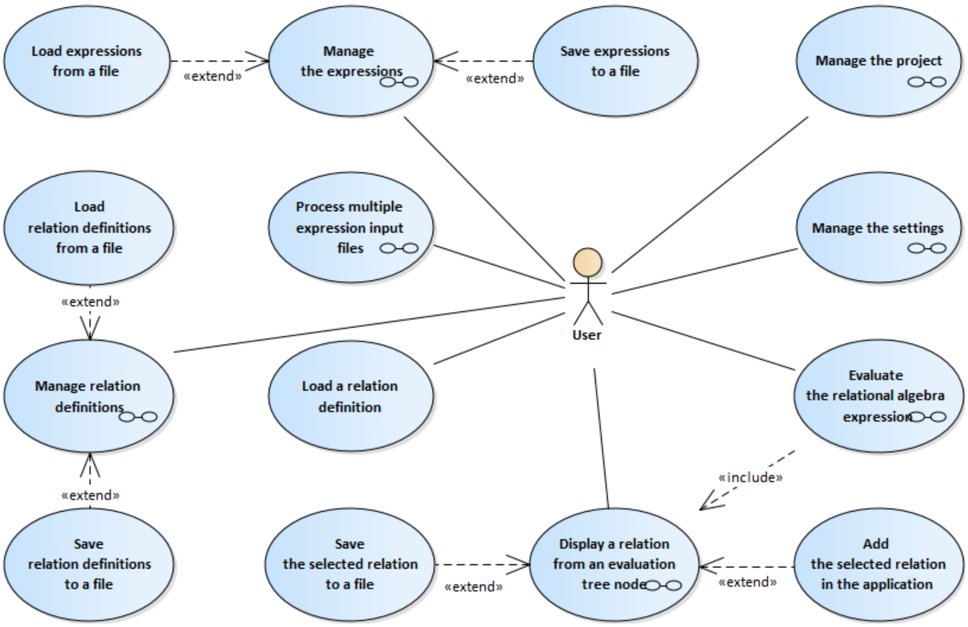


Figure 4.5: Project use cases

We described eight major use cases and six minor ones as shown in Figure 4.5:

- The **Manage relation definitions** UC describes how the user creates, deletes, imports, or exports the relation definitions. It is extended by **Load relation definitions from a file** and **Save relation definitions to a file** use cases.
- The **Load a relation definition** UC describes how the user edits the relation definition and how the application loads it.
- The **Manage the expressions** UC describes how the user creates, deletes, imports, or exports the expressions. It is extended by **Load expressions from a file** and **Save expressions to a file** use cases.
- The **Evaluate the relational algebra expression** UC describes how the user edits the expression and how the application evaluates it. It includes the **Display a relation from an evaluation tree node** use case.

- The **Display a relation from an evaluation tree node** UC describes how the user uses an evaluation tree. It is extended by **Save the selected relation to a file** and **Add the selected relation in the application** use cases.
- The **Process multiple project input files** UC describes how the user selects files for batch processing and how the application process them.
- The **Manage the project** UC describes how the user loads or saves the project.
- The **Manage the settings** UC describes how the user changes the settings of the application.

For each use case, we created a mapping on requirements and its screen design. You can find these files in the thesis attachment. For example, we show the *Evaluate relational algebra expression* use case in Figure 4.6 and list its steps:

1. The user edits the current expression. String values are between quotes. The user can type a backslash before a quote to use it inside a string.
2. The application suggests available relation or column names at the cursor position. The suggestions are ordered by the sequence or characters before the cursor.
3. IF the user selects the suggestion THEN:
    a. The application inserts the suggestion at cursor position.
4. IF there is an error in the input THEN:
    a. The application underlines the error. If the user moves the mouse over it, error description appears.
5. IF the user clicks a button with a special symbol THEN:
    a. The symbol is inserted at the cursor position. GO TO step 1.
6. Once the user finishes the editing, they click the "Evaluate" button.
7. The application parses the evaluation tree from the input.
8. IF there is an error in the input THEN:
    a. The application displays and highlights the error. GO TO step 1.
9. The application evaluates the parsed tree.
10. IF there is an error in during the evaluation THEN:
    a. The application displays and highlights the error. GO TO step 1.

11. The application saves the evaluation tree and the use case *Display a relation from an evaluation tree node* is triggered with the root node selected to display as default.

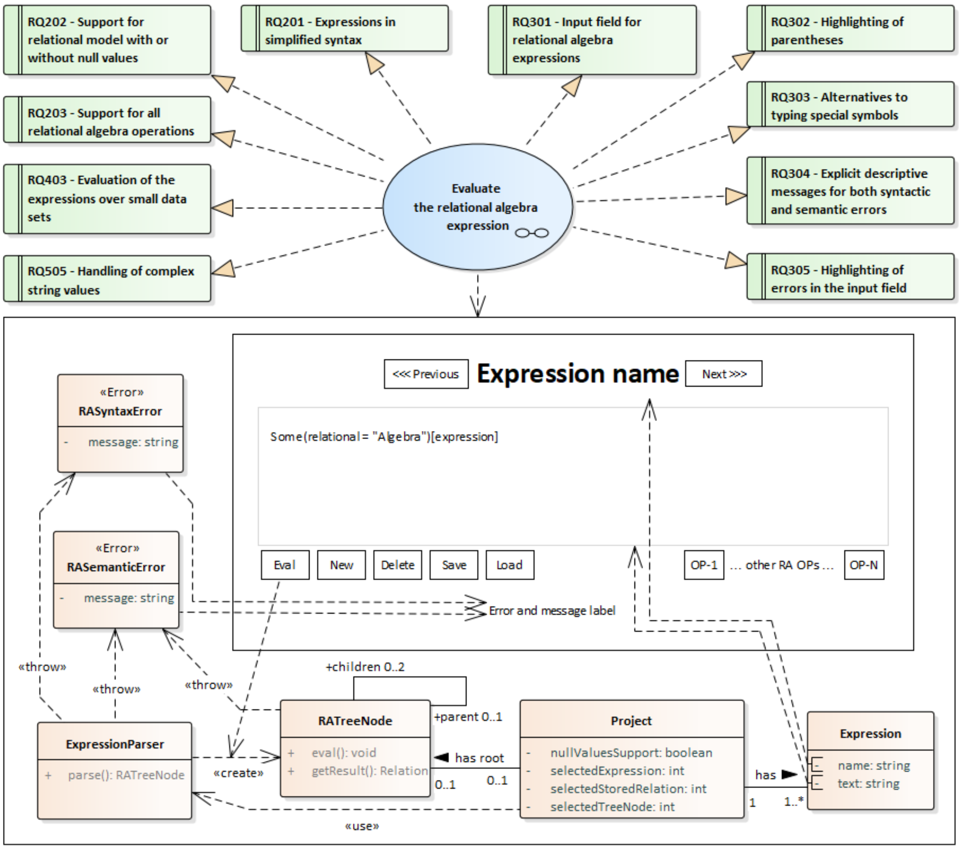


Figure 4.6: Detailed use case: Evaluate relational algebra expression

Further, we show the *Load relation definition* in Figure 4.7 use case:

1. 1. IF the user clicks on the "Rename" buttons THEN:
   a. IF there is no relation with the name in the input field THEN:
      - The name of the displayed relation is changed to the value in the input field.

   b. GO TO step 1.
2. IF the user changes values in the table THEN:
   a. The values are propagated to the relation instance.
   b. IF the new value is invalid THEN:
      - The invalid value is highlighted in the table. When the user moves a mouse over the invalid value, error description appears.

   c. GO TO step 1.

3. IF the user clicks on the "+" button in the last table column THEN:
   a. A new column is added and displayed. GO TO step 1.
4. IF the user clicks on the "+" button in the last table row THEN:
   a. A new row is added and displayed. GO TO step 1.
5. IF the user clicks on the "Load" button THEN:
   a. IF the displayed relation is valid THEN:
      - A (loaded) Relation instance is created from the displayed data. It overwrites a loaded relation with the same name.
   b. IF the displayed relation is invalid THEN:
      - An error message is displayed.
6. IF the user clicks on the "Delete loaded" button THEN:
   a. All (loaded) Relation instances are removed. GO TO step 1.

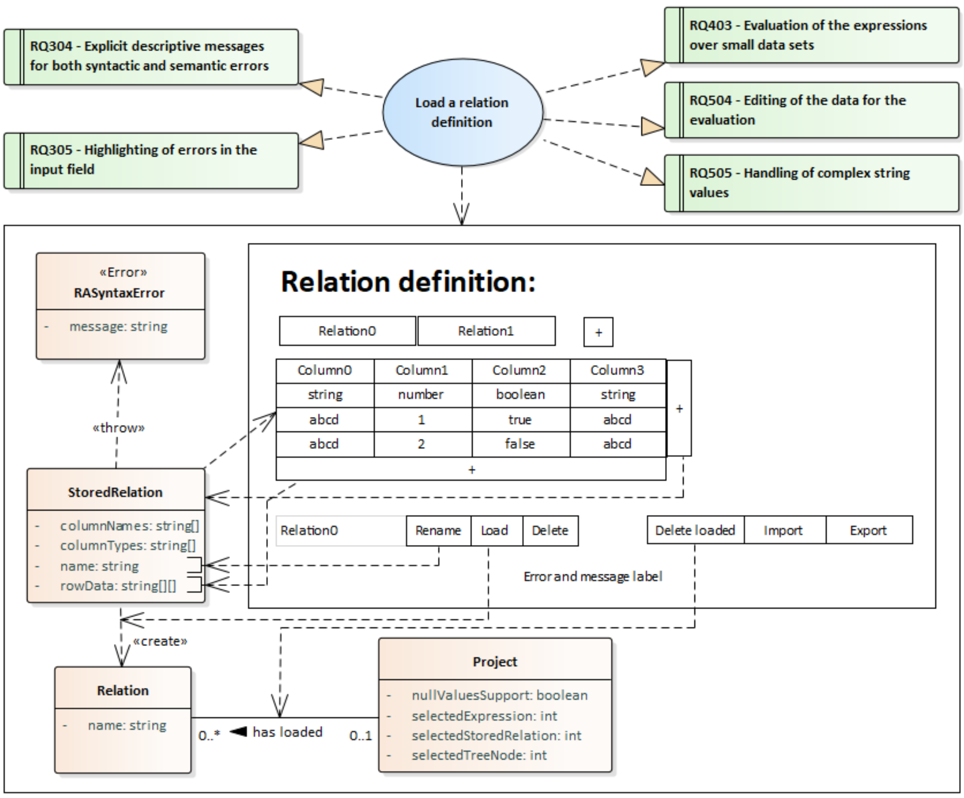


Figure 4.7: Detailed use case: Load relation definition

4.6 Class Model

In the use cases, we referred to several classes that evolved from business entities. We defined these classes for further implementation so that they do not fully correspond to the business entities. In the Class model, we specify data types for intended JavaScript implementation. As the main idea of each class remains the same, we will describe the changed entities only.

In the class model, we do not mention files because they are not a part of the application. We define store managers to separate the process of data storing and data itself. We present the main methods of classes that are important for their behavior, not their data.

Relation-related Classes. We found out that Column entities only represent pairs of the name and the type. As they always appear in a set, we can replace this set with a key-value map, where keys are the names and values are the types.

A similar simplification happens to the Row Entry entity. As it only describes a pair of the column name and the column value, we also replaced it with a key-value map.

We present the enumeration SupportedColumnType to specify column types in detail. We define the ColumnContent data type that extends SupportedColumnType by `null`, as it is a separated type in JavaScript.

Note that we do not use these specific types in the StoredRelation class. It uses string values only as it may be in an invalid state when the user edits it. When the Relation class is created from a valid StoredRelation class, the values are cast from strings to specific types for easier evaluation.

We show the model of the Relation-related classes in Figure 4.8.

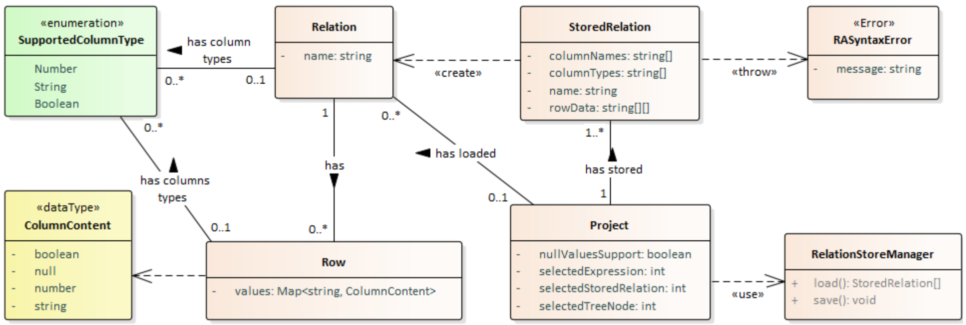


Figure 4.8: Model of the classes related to a relation

Expression-related Classes. The Evaluation Tree entity does not exist in the Class model, and the tree is represented by its nodes only. The eval method recursively evaluates the subtree of the node. The getResult method reuses once evaluated results to save computations.

We show the model of the Expression-related classes in Figure 4.9.

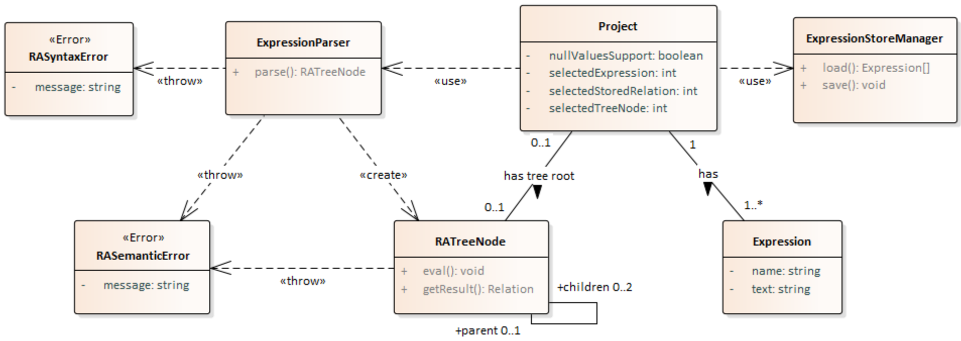


Figure 4.9: Model of the classes related to an expression

Project-related Classes. The Project class contains information about the selected expression, relation, and node of the evaluation tree. We moved the evaluation tree root from Expression entities to the Project as there is always only one evaluation tree in the application.

The BatchProcessor class separates the evaluation of multiple selected files and the Project. That means the batch processing does not affect the state of the loaded Project.

Chapter 5
Documentation

In this chapter, we provide a programming documentation for the application we proposed. We present the used programming languages, frameworks, and libraries in the first section. In the second section, we describe code packages. Then, we present the implementation challenges we encountered, the application testing, and deployment.

5.1 Used Technologies

Now, we will briefly present programming languages, frameworks, and libraries we used in the application implementation.

As we design a static web page application, we thought of two programming languages: Java or JavaScript, both commonly used for web development. Finally, we chose JavaScript. Because there are some disadvantages in plain JavaScript code, we finally decided to use TypeScript that adds static syntax and types checking to JavaScript.

5.1.1 JavaScript

JavaScript [6] is a programming language, which corresponds to *ECMAScript* specification. JavaScript is mainly run in web browsers as a client-side of web applications, but we can also use it in servers or desktop applications. JavaScript files have a `.js` extension.

The first version of ECMAScript (ES) was presented in 1997 by *Brendan Eich*, and then it was used in the *Netscape Navigator* web browser. A year later, it received an international technical specification ISO/IEC 16262. JavaScript is a marketing name for an ES implementation, but there are essential differences between JavaScript and Java (they only share a similar syntax). Besides JavaScript, another relevant implementation of ES specification is *JScript* by Microsoft. ECMAScript became popular, and the majority of web browsers supports it in the early 2000'.

All ECMAScript versions are backward compatible. It ensures that an old code always works properly. On the other hand, old browsers may not support new features. We can solve this problem by adding a *polyfill* – a code that substitutes missing functionality using the available features. Anyway, polyfills cannot implement new syntax. For syntax substitution, we can use *transpilers*, e.g., Babel[1] or Google Closure Compiler[2]. These tools change the new syntax in a code to the old one.

The sixth edition of ECMAScript brought many important innovations. It was released in 2015 after a long development. For example, it supported Unicode characters in string literals, provided the `Promise` object for asynchronous computations, presented new definition keywords `let` and `const` for block-scope variables, or the `class` keyword for easier creation of objects.

In the last six years, there is a new version of ECMAScript standardization every year. Development was moved to GitHub so that a wider group of people could contribute. Anyway, no other edition brought as many changes as the sixth one. In the year 2021, the twelfth version is expected to be released.

Originally, JavaScript was a *web scripting language*, but it became a *general-purpose programming language*. The purpose of a scripting language is to manipulate and automate the facilities of an existing system. A web scripting language adds dynamic features to web pages in browsers or computations to servers.

JavaScript executes computation in a *host environment*. On the client side, the host environment is provided by a web browser. The web browser provides all objects which the program uses, e.g., an abstract representation of the whole screen, displayed elements, or browsing history. Also, it provides listeners for events, as JavaScript is *event-driven* and has no `main` function. On the server-side, the host environment provides communication with clients by request and response objects and access to the files.

Together with *HTML* and *CSS*[3], JavaScript constitutes the three pillars of the *World Wide Web*. JavaScript code can be inserted in `<script>` elements right in HTML files, but it is ineffective as it slows down rendering. The better option is to place scripts in separate files. Also, it separates the logical and graphical parts of the application.

JavaScript is a *mixed paradigm language*. It has functional as well as object-oriented features. Objects in JavaScript are not class-based, but we use prototyping to create them. Class-based objects (e.g., in Java) have a unique state but share methods within the class. In JavaScript, objects share methods defined in the prototype of the constructor function, but they can have their unique functions as well. We can add a new property to an object at runtime. As for the data content, we can see objects as collections of key-value pairs, where the key is the name of the property, and the value is the property itself.

JavaScript uses *hoisting*. It means that it moves the declarations of variables and functions to the top of the current block of code. We can then access a variable that was not declared yet.

[1] Babel is accessible on `https://babeljs.io/`.

[2] Google Closure Compiler is accessible on `https://developers.google.com/closure/compiler`.

[3] We will present HTML and CSS in Subsections 5.1.4 and 5.1.5.

Unfortunately, hoisting and other specific ECMAScripts behavior can lead to mistyping errors. For example, assume we have an object `car` with a property `color`. Typing `car.colour = 'Blue'` is not an error in JavaScript, but it creates a new property in the `car` object. To prevent these errors, we can use the *strict mode*. Modules and definitions of classes (with `class` keyword) use this strict mode automatically. We can also use the strict mode in other files if we add `"use strict"` on the first line.

5.1.2 TypeScript

Although we can use the strict mode, we still miss some useful features in JavaScript. The programming language TypeScript [7] is a typed superset of JavaScript, which provides these features. TypeScript files have a `.ts` extension.

TypeScript works as a *static syntax checker* of the code. Before the code runs, TypeScript checks whether the code has a valid syntax. Because it is a superset of JavaScript, all valid JavaScript code is valid in TypeScript as well. Also, it does not change the runtime behavior of the JavaScript code. TypeScript code is compiled to JavaScript before executing.

As the name suggests, TypeScript enforces types. Once we assign a type to a variable, we cannot change it to a different type. TypeScript determines the type of the variable automatically after an assignment of a value:

```
let str = "string value...";
str = 1; // Error: Type '1' is not assignable to type 'string'.
```

or we can determine it explicitly in a variable declaration:

```
let str: string;
str = 1; // Error: Type '1' is not assignable to type 'string'.
```

We can create types with named values with the `enum` keyword. TypeScript supports enums with numeric, string, or combined values. We can create new types by combining the existing ones. For example, the following code creates a type, which accepts numeric as well as string values:

```
type NumOrStr = number | string;
let a: NumOrStr = 1;
a = "we can assign string as well";
```

Also, we can create special types by the explicit enumeration of the possible values:

```
type SpecialType = "some string" | 123 | false;
```

The next important feature of TypeScript are *interfaces*. An interface determines a set of properties that a given object has to provide to pass type checking. We do not have to say explicitly that an object implements an interface (e.g., we do have to say this in Java), as TypeScript compares shapes[4] of objects and interfaces automatically:

```
interface NameHaving {
    name: string;
}
// no explicit implementation of NameHaving interface
let person = {name: "Joe", age: 50};
let sthWithName: NameHaving = person; // no error
```

As we can see, an object can provide more properties than the interface and still pass type checking. That is called a structural type system. We can force a class to provide all properties of one or more interfaces with the `implements` keyword.

5.1.3 React

React [8] is a widely used JavaScript framework that makes the creation of user interfaces easy and fast. We can use it for the development of web, desktop, or mobile applications.

React provides a syntactic extension of JavaScript. Files with React specific syntax have `.jsx` extension (or `.tsx` if TypeScript is used). This syntax allows to use HTML-like definitions of elements right in the JavaScript file, so the code is more readable. For example, we can define a paragraph element as follows:

```
const element = (
    <p color="red">
        Hello, world!
    </p>
);
```

This block of code is changed to plain JavaScript after compilation:

```
const element = React.createElement(
    'p',
    {color: 'red'},
    'Hello, world!'
);
```

To embed JavaScript code in React element definitions, we have to wrap it in curly brackets.

React *elements* are not the same as HTML elements. They are plain, i.e., without functions, so they are fast to create. Also, they are immutable, which means they need to be created again after each their modification. Immutability seems like an ineffective feature but React can track which

[4] A shape of an object is a set of its properties, i.e., its data and methods.

particular parts were actually changed and create again only them. Rendering the web page transforms React elements to standard HTML.

React *components* are at a higher level than elements – components consist of elements. They provide a render function, which returns the representing React element. Components accept parameters, called `props` (properties). In props, we can define how the component looks like, or provide some more complex data. Props are immutable, too. We store mutable inner variables of components in their `state`.

React uses one-way binding. It means that components know their children but do not know their parents. Parents affect their children by the modification of props, which causes React to rerender affected parts. We can use some props as callbacks[5] to pass the information from children to parents. One-way binding causes that we must store the shared state of components in some common ancestor.

React does not use inheritance between components. Instead, it recommends using composition. When the application structure is well designed, we can reuse components by changing their props.

There is an easy way to get started with a new React app. Create React App [9] allows preparing all required configuration and dependencies with a single command. It can save plenty of time as it hides a complex structure of dependencies and provides a default configuration for both development and production.

5.1.4 HTML

As mentioned before in the JavaScript introduction, HTML, CSS, and JavaScript constitute the pillars of the *World Wide Web*. We do not use HTML in our code directly, as we create it by React or JavaScript functions. Anyway, we should briefly introduce it, as it is a key part of the compiled application.

HTML [10] stands for *HyperText Markup Language*. It is similar to XML[6], but it has a predefined set of elements, attributes, and structure. The HTML files have `.html` or `.htm` extensions. The current version is HTML5 from 2014. It is backward compatible with previous versions. Because authors of HTML decided to ensure backward compatibility even in the future, there are no longer any new versions. All new features are only extensions of HTML5.

HTML describes the web page content as an element tree. The root element is `<html>` which has `<head>` and `<body>` children. Each element has an opening and closing tag. In the opening tag, we can specify attributes associated with a given attribute. Between the tags, we place contents of a given attribute, i.e., its successors.

[5] Callbacks are functions given as parameters, in our case, they are functions of a parent which we call in a child.

[6] XML (Extensible Markup Language) is used mainly for data storage and transmission. It represents the content as a tree of elements with attributes. Unlike HTML, it has no predefined tags – users can use custom names. Its specification is accessible on `https://www.w3.org/TR/xml/`.

HTML <head> element contains page metadata, for example:

- title, author, language, keywords, description
- links to external resources (e.g., CSS files or font definitions)
- appearance on small (mobile) screens

There is no compulsory metadata, but the more metadata we specify, the better the page works. The metadata also helps web search engines to classify pages, or browsers to render page contents correctly.

Besides metadata, we can define CSS in <style> elements or client-side JavaScript in <script> elements.

We define the visible content of the page in the <body> element. There is a lot of available elements to use in the body. All of them are containers and can contain nested elements or plain text. We should use their various types to describe semantic parts of the page. For example, it is possible to use a button element like a paragraph container. We can use CSS styling to make buttons appear the same as paragraphs. But doing so we violate the semantic meaning, and so the page structure becomes hard to understand. Also, modern browsers often use semantic meaning for advanced features as text reading or forms pre-filling.

In HTML, we can refer directly to JavaScript and CSS. For example, we can call a JavaScript function when an event is triggered, and style the element with a CSS class:

```
<button class="my-button" onclick="clickHandler()">
    Click me
</button>
```

Let us describe what the code does. We define a button element by its opening tag <button> and closing tag </button>. In the opening tag, we assign values to its `class` and `onclick` attributes. We assume that there exists a CSS class `my-button` and a JavaScript function `clickHandler`. Between the tags, we add the text *Click me* as the content of the button.

5.1.5 CSS

The last pillar of web pages to introduce is CSS [11], which stands for *Cascading Style Sheets*. CSS defines over 500 properties that change the appearance of HTML elements. Properties usually form groups with specific focus, e.g., font styling, or positioning. The current CSS version is CSS3 from the year 2005. There are no new versions because backward compatibility is guaranteed, so the CSS3 is only extended by new features.

There are three ways to include CSS in a web page. We can specify a property value as an attribute of a given element in an HTML file, which is called *inline styling*:

```
<button style="width: 100px; border: 2px dotted green;">
```

The second way is to define CSS in `<style>` element in the HTML head:

```
p {
    font-size: 16px;
}
.my-button {
    width: 100px;
    border: 2px dotted green;
}
```

The last way is to define CSS in a separated `.css` file. The definition looks the same as in the `<style>` element. The last approach is the recommended one because it separates the page content from its appearance.

In inline styling, we define property values that apply only to a given element. In a general styling definition in HTML head or `.css` files, we must describe which elements are the intended targets. To do so, we can use element tags or define our CSS *classes* (e.g., `my-button` above[7]). Further, we can combine tags and classes by selectors. For example, the following styling will apply to all elements with the my-button class that are direct descendants of any div element:

```
div > .my-button {
    background-color: blue;
    color: #ffffff;
}
```

More advanced styling can use CSS *pseudoclasses*. Elements gain and lose these classes at runtime by user actions. For example, the following definition overwrites the color styling defined above when the user moves the mouse over the my-button element:

```
div > .my-button:hover {
    color: rgb(255, 0, 0); /* color property overwritten */
    font-width: bold; /* new property set */
}
```

Note that the background color will still be blue. Also, we can see that there are many predefined keywords and functions in CSS.

One element can correspond to multiple style definitions. Inline styling has a higher priority and overwrites general styling. If there are more corresponding definitions at the same priority level, the last defined is applied. Unspecified properties keep default browser values.

We can create rich, adaptive web pages with CSS. Using *media queries*, we can change the styling on different screen sizes. Furthermore, with advanced CSS properties, we can create *animations* with no JavaScript code needed.

[7]Using classes is an often approach, so that CSS presents a shortcut for their reference – `.my-button` is equivalent to `[class="my-button"]`, that means all elements with the `class` attribute equal to `my-button`.

5.1.6 Other

There are millions of JavaScript users and projects[8] worldwide. The large community provides many open source[9] and permissively licensed[10] libraries and frameworks. These libraries make the development easier and faster, as a wide range of problems was already solved in previous projects. For their installation, we used the JavaScript package manager npm [13]. Let us now provide a short description of the particular libraries we utilized in our project.

jest. Jest [14] is a JavaScript testing framework. It provides intuitive functions for unit testing. Testing expressions are easy to understand as they form sentence-like chains.

lodash. Lodash[11] is a general-purpose JavaScript library. It provides functions to advanced work with arrays, objects, and strings. We used its function isEqual for deep equality checking of JavaScript objects[12].

FileSaver.js. FileSaver.js[13] is a library for saving files from JavaScript to the user device. We used it for all downloads from the application.

JSZip. JSZip[14] is a library for managing ZIP files in JavaScript. We used it to zip up CSV relation definitions and the results of batch processing before downloading.

visx. Visx[15] is a powerful library for advanced visualization in React. It provides a wide range of visualization components for graphs, drawing, maps, zooming, and many more. We used it for rendering the evaluation tree.

export-svg-with-styles. Export-svg-with-styles[16] is a specific-purpose library for downloading HTML SVG components as PNG pictures. We used it for exporting the evaluation tree as a PNG file.

---

[8] Over 97% of webpages use JavaScript in April, 2021, according to Q-Success [12].

[9] Open source projects also have benefits for their owners as anyone can contribute. For example, we found a bug in React when developing our application, reported it and people offered help with fix: `https://github.com/facebook/react/issues/21094`.

[10] *"A "permissive" license is simply a non-copyleft open source license – one that guarantees the freedoms to use, modify, and redistribute, but that permits proprietary derivative works."*, a description by *Open Source Initiative*, [cit. 2021-04-30]. Accessible from: `https://opensource.org/faq#permissive`.

[11] Lodash is accessible on `https://github.com/lodash/lodash`.

[12] There is no such feature in plain JavaScript. Both equality operators '==' and '===' use reference comparison when used on objects. It means that they return true only if both compared objects refer to the same memory address.

[13] FileSaver.js is accessible on `https://www.npmjs.com/package/file-saver`.

[14] JSZip is accessible on `https://stuk.github.io/jszip/`.

[15] Visx is accessible on `https://github.com/airbnb/visx`.

[16] Export-svg-with-styles is accessible on `https://www.npmjs.com/package/export-svg-with-styles`.

JSDoc. JSDoc[17] is a documentation generator for JavaScript projects. It creates a HTML documentation based on comments in the source code.

better-docs. As JSDoc is implemented for JavaScript, we use the better-docs[18] extension to support TypeScript files.

5.2 Code Packages

We logically structure the code of the application into eleven packages, which wrap related functionalities. Packages contain code files as well as tests in sub-packages. In this section, we go through each of them and describe their purpose.

Expression. The expression package defines the `Expression` interface and provides functions for its maintaining. It contains the core application algorithms[19] in `ExprParser` and `ValueParser` classes that parse the relational algebra and logic-algebraic expressions, respectively. *ExprTokens.ts* and *ValueTokens.ts* files define tokens used in the tokenization. The `ExpressionStoreManager` provides expression importing and exporting using text files. This package closely relates to *ratree* and *vetree* packages.

Ratree. The ratree package contains nodes of evaluation trees of the relational algebra expressions. Most nodes represent one specific operation, but similar operations share a single node (e.g., `NaturalJoinNode` provides natural join and semi joins, `SetOperationNode` provides set union, intersection, and difference). `RATreeNode`, `BinaryNode`, and `UnaryNode` abstract classes define the interfaces of all extended nodes. The `RATreeFactory` provides a centralized creation of new nodes.

Vetree. The vetree package contains nodes of evaluation trees of the logic-algebraic expressions. We distinguish five types of `VETreeNodes`:

- `ComparingOperator` represents operators that produce a boolean value from all input types (i.e., ==, !=, <, >, <=, >=).
- `ComputingOperator` represents mathematical operators (i.e., +, -, *, /)
- `LogicalOperator` represents logical operators (i.e., $\wedge, \vee, \neg$)
- `LiteralValue` represents constants
- `ReferenceValue` represents variables (i.e., column names)

We use `VETreeNodes` to represent conditions in `SelectionNodes` and `ThetaJoinNodes` in the *ratree* package.

[17] JSDoc is accessible on `https://github.com/jsdoc/jsdoc`.
[18] Better-docs is accessible on `https://github.com/SoftwareBrothers/better-docs`.
[19] We will describe these algorithms in depth in Section 5.3.1.

Relation. The relation package defines two relation representations used in the application.

The `StoredRelation` class represents relations edited by the user. We store their data in string arrays because they have an order (given by position in the editing table) and can be in an invalid state (we cannot use the `ColumnContent` data type). The `RelationStoreManager` provides importing of the `StoredRelations` from a file and exporting them to it.

The `Relation` class represents relations loaded in the application. We represent individual relation rows by the `Row` class. They always are in a valid state, which means we can use specific `ColumnContent` data types. They use unordered maps to store the data (because the formal relational schema has no order).

Project. The project package defines the `Project` interface. We use it in the `ProjectStoreManager` to load the project from a file or save it.

Components. In the components package, we implemented the user interface. The `MainScreen` component wraps four page sections: `RelationSection`, `ExpressionSection`, `ResultSection`, and `ManagementSection`. These sections correspond to their description in the analysis.

Most components use the React framework for updating, but there are two exceptions. The `MessageBox` uses plain JavaScript to display messages and errors in a pop-up box. The `XTextArea` uses React to create its root container, but other logic is in plain JavaScript as it gives us more control.

We define the component-specific CSS styles in the *css* subpackage. General styling of the application is defined in the *index.css* file in the project root directory.

Batch. The batch package provides the `BatchProcessor` class for processing multiple project files. As we implemented all parsing and evaluating algorithms in other packages, most code in the `BatchProcessor` is the formatting of report files.

Utils. The utils package provides auxiliary functions used in the whole application. Some of them are simple, e.g., date formatting or modulo operation for negative numbers. We implemented advanced string functions in `StringUtils` and `IndexedStringUtils`. The `FileDialog` handles the uploading of files from the user device. The `LocalStorage` wraps access to the browser storage and provides initial values of stored variables. We define reserved column and relation names in the *keywords.ts* file. The application uses the `Mail` class for automatic error reporting.

Types. In the types package, we implemented custom data types. Simple but interesting data types are `ISToISMap` and `NNToSMap`. They are specific-purpose wrappers of the built-in key-value map allowing us to use `IndexedStrings` and pairs of numbers, respectively, like keys. An advanced data type is `IndexedString` which allows storing characters and their indexes in the original text[20].

Error. The error package provides custom error types and functions. Besides `RASyntaxError` and `RASemanticError`, we implemented `CodeError` class which represents unexpected errors in the application. `ErrorWithTextRange` is a predecessor of RA errors and stores the error range to highlight in the input text. We use `ErrorFactory` in the application for creating errors with their predefined messages.

Language. The last package we mention is the language package. The main file in the package has the same name and contains a map of supported languages in the application. We define each language in a separated file as a `LanguageDef` object which provides over 150 phrases used in the application. We implemented the application to be easily expandable by new languages. The contributor only duplicates and translates one file and edits three lines in the *language.ts* file.

5.3 Implementation Challenges

Now, we will describe several implementation challenges that we encountered. The most challenging tasks were dynamic parsing of unfinished expressions and highlighting of errors in the input field. We used proven approaches (e.g., the Shunting-Yard algorithm or HTML textarea element) as well as our own ideas (e.g., solution of errors in the user input or extension of the built-in string object).

5.3.1 Expression Parsing

The key algorithm of the application is the parsing of relational algebra expressions. The algorithm takes textual input from the user and creates an evaluation tree of operation nodes. The user uses the infix form that means that binary operators are between their operands (in our case, between their source relations). This form is comfortable for a human but hard to work with for a computer. For computer processing, we need to change the expression to the prefix or postfix form[21]. In these forms, operators are before or after their parameters, respectively. In particular, we use the postfix form that.

For example, we change an expression:

```
Car*Owner(name = "Lukas")
```

[20] We will describe its implementation in depth in Section 5.3.2.

[21] Prefix and postfix forms are often called normal and reverse polish notations. The normal polish notation was invented by a Polish logician and philosopher Jan Łukasiewicz in 1924. More about his life and work in an article by Peter Simons on the Stanford Encyclopedia of Philosophy, 2020, accessible on `https://plato.stanford.edu/archives/sum2020/entries/lukasiewicz/` [cit. 2021-05-05].

to the following representation:

```
Car Owner (name = "Lukas") *
```

To do so, we use the *Shunting-Yard algorithm*. This algorithm receives an array of infix form tokens and returns its postfix form. Before we describe the Shunting-Yard algorithm, we introduce how the application parses the infix form tokens.

Parsing of Tokens. The textual input is processed from the start to the end. When we reach the next character, we find the maximum following sequence that constitutes one syntactic part of the expression. For example, after reading *C*, we add all the following letters to create a relation name *Car*.

We need to know the assumed syntax rules to create correct parts. Usually, the first character describes the current syntactic part unambiguously. In ambiguous situations, we need additional checking. For example, after reaching ***, we must check whether the next character is $>$ before we return a natural join token to handle right semi-joins.

We add metadata needed for the Shunting-Yard algorithm to the parsed tokens. Besides the textual representation, we store a token type and the precedence of binary tokens. The parsing of the example above creates the following array:

```
{Relation "Car"},
{Binary operator (Natural join) "*" with precedence 10},
{Relation "Owner"},
{Unary operator (Selection) "(name = "Lukas")"}
```

Shunting-Yard Algorithm. The algorithm changes an array of tokens in the infix form to the postfix form. It was invented by Edsger Dijkstra in the year 1961 [15]. The general version of the algorithm can process complex expressions with constants, left- or right-associative operators, and functions.

For our purposes, we implemented a simplified version as no considered operators are right-associative. Also, our implementation does not support functions, although some operators act like functions. These operators are selection, projection, rename, and theta joins. They accept a parameter that affects their behavior. In fact, that means that all unary operators should be actually treated as binary, and, analogously, theta joins as ternary. To handle this, we use the Shunting-Yard algorithm at two levels. The first level works with tokens where each parameterized operator constitutes one token. After creating an evaluation tree from these tokens, we handle parameterized operators. In selections and theta joins, we also use the Shunting-Yard algorithm to build an evaluation tree for their algebraic-logical condition.

Algorithm 1 describes our simplified Shunting-Yard implementation. It uses only one auxiliary data structure – a stack of encountered binary operators and parentheses. We can see that it does not push parentheses to the result array as the operator evaluation order is determined unambiguously in the postfix form. The algorithm recognizes mismatched parentheses. A closing parenthesis before an opening one causes an error on Line 14. A missing closing parenthesis causes an error on Line 23.

Algorithm 1: The Shunting-Yard algorithm

```
   Input: inTokens: an array in the infix form
   Data: operators: a stack for storing of operators
   Result: postTokens: an array in the postfix form
1  foreach token in inTokens do
2      if token is a relation or token is a unary operator then
3          postTokens.push(token);
4      else if token is a binary operator then
5          while head of operators is a binary operator and
            token.precedence <= head.precedence do
6              head = operators.pop();
7              postTokens.push(head);
8          operators.push(token);
9      else if token is an opening parenthesis then
10         operators.push(token);
11     else if token is a closing parenthesis then
12         while true do
13             if operators is empty then
14                 throw ERROR: mismatched parentheses;
15             head = operators.pop();
16             if head is an opening parenthesis then
17                 break while;
18             else
19                 postTokens.push(head);
20 while operators is not empty do
21     head = operators.pop();
22     if head is an opening parenthesis then
23         throw ERROR: mismatched parentheses;
24     else
25         postTokens.push(head);
26 return postTokens
```

Evaluation Tree. After receiving a token array in the postfix form, we can process it from the end to the start. We describe the algorithm in our example. Recall the expression after changing it to the postfix form:

```
Car Owner (name = "Lukas") *
```

We start with a binary operator *** (natural join). We need to supply two parameters for it. To do so, we recursively process the previous part: a unary selection operator *(name = "Lukas")*. This operator needs one parameter, so we process the relation part *Owner*. Relations need no parameters and so we stop the recursion branch. This branch created the right-hand source for the natural join. To get the left-hand one, we repeat the same process and receive a single relation *Car*.

Although the Shunting-Yard algorithm handles mismatched parentheses, the given postfix expression may still be invalid in other ways. For example, if we omit the relation *Car* in the original infix form, the Shunting-Yard algorithm does not find the error. We find the error while building the evaluation tree when there is a missing left-hand source for natural join.

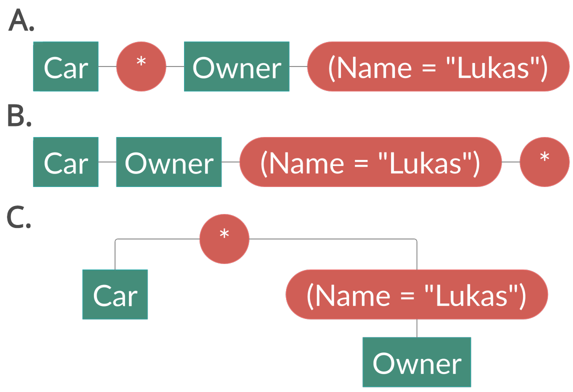


Figure 5.1: Three possible representations of an expression. The infix form (**A**), the postfix form (**B**), and the tree form (**C**).

Continuos Parsing. Previous algorithms were able to detect errors, but they were not able to solve them. It is sufficient when we want to evaluate an expression – when an error occurs, we terminate the process and report it to the user. The user can solve it and try again. The problem is when we need to parse incomplete expressions to find available suggestions at the cursor position. Also, users expect to see errors during typing, not only after the evaluation.

There are multiple places where an error can occur. The first one is the parsing of tokens. When the user types, parameterized operators or parentheses are often unclosed. When the expected closing character is missing in the input (e.g., closing parenthesis after opening parenthesis in selection), we add it at the input end. Further, we skip and ignore unexpected characters (e.g., question mark outside quotes or closing parenthesis before opening one). These simple solutions ensure an error-free Shunting-Yard algorithm.

Other errors can occur in the building of an evaluation tree. We chose to solve them before the Shunting-Yard algorithm starts. At the time, the expression exists as an array of tokens in the infix form. We described a few rules which the well-formed expressions must follow. Rules for each adjacent

	Relation	Unary	Binary	Opening	Closing
Relation	✗	✓	✓	✗	✓
Unary	✗	✓	✓	✗	✓
Binary	✓	✗	✗	✓	✗
Opening	✓	✗	✗	✓	✗
Closing	✗	✓	✓	✗	✓

Table 5.1: Rules for expressions in the infix form. Each adjacent pair of tokens must belong to any ✓-marked cell. Rows describe the first token in a pair and columns the second one. We can see common patterns in the behavior. Relations, unary operators, and closing parentheses expect the same token types before them, as well as binary operators and opening parentheses. Also, relations and opening parentheses expect the same token types after them, as well as unary operators, binary operators, and closing parentheses.

pair of tokens are displayed in Table 5.1. Besides these rules, the first token of an array must be a relation or an opening parenthesis, and the last token must be a closing parenthesis, relation, or unary operator.

We process an array from the start to the end and check the rules for each adjacent pair. If any pair does not follow the rules, we insert a new token in the middle. We can solve all violations by inserting a single relation token or a single binary operator token. We use the relation with a forbidden empty name so that such a relation is not defined. As the inserted binary operator, we use a natural join.

These solutions ensure that the application builds a valid evaluation tree every time. All errors are stored, passed to the presenting components, and highlighted to the user.

5.3.2 Text Position

In the previous section, we have described how to find and handle errors in unfinished expressions. The next step is to report them in a user-friendly way. We use a usual approach – highlight the error by underlining it and display its message when the cursor is over it.

The problem is to locate an error in the advanced stages of input processing. In the beginning, we have the whole input as one string. As the processing continues, we split the input text multiple times to describe individual tokens, we may skip some white spaces, or add new characters to solve errors. After that, the character index does not correspond to its original position in the input. We implemented a custom string representation to handle it. Also, we extended a textarea element for a simpler presentation.

IndexedString. The idea of an `IndexedString` is simple – we store a text as an array of pairs `{character: string, index: number}`, called `IndexedChar`. When we create a new `IndexedString`, we set character indexes to their original values. After creation, no `IndexedString` method changes indexes in `IndexedChars`.

The most important method is `getRange` that returns indexes of the first and the last `IndexedChar`. There are more additional methods for handling character indexes, e.g., `getFirstIndex`, or `getLastIndex`. Besides these functions, `IndexedString` implements many built-in string methods with identical signatures.

XTextArea. Now we know how to find and locate errors. Also, we can find a cursor position in the string and suggest available relation or column names, or match pairs of parentheses. The last step is to display this information. To do so, we extended a textarea HTML element by additional methods.

These methods compute the pixel position of the character on the screen. We use a monospace font so that all characters have the same pixel width. With this assumption, we compute an upper-left corner `[x, y]` position of a character on the i-th index as follows:

```
line = number of newlines before the i-th character
lastNewLine = last newline index before the i-th character
x = (i - lastNewLine - 1) * fontWidth
y = line * lineHeight
```

The position is relative to the upper-left corner of the textarea. For proper rendering, we need to change it by the current vertical and horizontal scroll values. At the final position, we can render a div element with suggestions or with an underlining style.

A possible alternative is not to use a textarea element at all. It is possible to implement a custom input element that uses editable div elements. In such a solution, we do not need to compute positions as we can enclose each character group in a single element. It brings many custom feature possibilities as well as implementation difficulties. The built-in textarea element handles newlines, scrolling, text inserting, or undoing. Because of these verified features, we chose to reuse a textarea element.

5.4 Testing

Testing is an important part of each software project. Sufficiently large test coverage can detect a majority of errors in the code before the deployment. It means that we do not have to change the software when customers already use it. It is more user-friendly as the application is always available. Also, the project costs are lower as error solving is more expensive in later stages of the development.

Unit Tests. The most frequent tests are called *unit tests*. They test short blocks of code, usually single functions. Developers should write unit tests just after finishing a new code or even before it.

We use the *jest* [14] framework for unit testing. In each unit test, we follow the *AAA* pattern: *arrange, act, assert*. First, we prepare input data and expected output, then we call the tested function, and finally, we compare its output with the expected one.

We use unit tests for testing particular parts of complex algorithms. Most of the unit tests are for correct expression parsing and evaluation. There are many classes and functions from multiple packages that cooperate. These packages contain a subpackage */tests* where their unit tests are.

Crucial tests are in the *expression* package. They check valid parsing of tokens, changes of the infix token form to the postfix one, and correct automatic expression completion.

The longest unit tests are in the *ratree* package for testing relational operators. These tests contain many relation definitions as we tried to capture many input combinations. We tried to reuse the input relations in several test files, but we do not think it adds much clarity as there still exist differences. Retrospectively, we understand we should have created one set of relations and used them in all tests.

Similar but more straightforward tests are in the *vetree* package as the operators process simpler inputs – not relations, but numbers, strings, and booleans. Further strong testing is, for example, for `IndexedString` in the *types* package, string utils in the *utils* package, and relation classes in the *relation* package.

Logging. Logging is a common technique in software development. We use it to capture program behavior by saving its state in certain moments. Usually, we save the state as messages in files or print them in the console.

Logging does not substitute tests. We use it mainly for debugging, as we can easily add new information to the log as needed. Server developers use logging for security. When the server crashes, saved logs can help to identify the crash cause. Also, when hackers attack, logging can approximate their position or describe their intentions.

User Testing. During the development, we can test our code to ensure its correctness. Anyway, correct algorithms do not imply a user-friendly application. To handle it, we can use the feedback of real-world users in later development stages. In our case, we presented our application to a group of approximately 20 CTU students in April 2021. At the time, they were learning relational algebra, so our application could have helped them.

Acquired feedback from involved students as well as teachers led to the following improvements:

- modern colors, clear layout, bigger font size
- simplified relation definition – no exposing of inner double representation of relations
- better behavior of suggestions in the expressions – appearing on Ctrl+Space, highlighting of matched letters

More intensive usage of the application could be an opportunity for revealing the remaining errors. When an unexpected error occurs, the application shortly describes it in a pop-up message box and asks the user to report their last actions. We understand that users do not want to write detailed reports, so we also use automatic reporting. As there is no backend server, the application saves its current state and sends it as an email. The email contains, e.g., a detailed error description, current loaded and edited relations, or current expressions.

Other Testing Methods. Unit tests are not the only development tests. For example, integration tests inspect the correct cooperation of multiple classes, functions, or modules. We use unit tests only, but some of them actually focus on the integration aspect as well. For example, a parsing method of `ExprParser` uses other `ExprParser` methods, `IndexedString` utils, and much more. A unit test for this method explicitly inspects only its expected output, but it tests its integration in the background.

There exist test techniques for testing graphical user interfaces. They automatically test the content of the page and simulate user interaction. We decided not to use them as their preparation is complex and potential benefits would most likely not outweigh the necessary time overhead. The page content (e.g., button titles, page layout) often changed in the development, so it was faster to test its behavior manually.

5.5 Deployment

We used *localhost*[22] for the development of our application. It is a common approach, as it is easily accessible and quick. The application in development never runs under the same conditions as the production version. Often, many configuration errors occur after deploying.

Fortunately, there are modern solutions that hide complex configurations and let programmers focus on the code itself. One of them is Create React App [9]. It is a set of predefined scripts for both development and deployment. We solve most of the tasks by a single command. Easy deployment and maintaining was the reason why we designed our application as a single page application with no backend.

A modern approach is not to maintain an own server but to use hosted cloud servers. We chose *GitHub Pages*[23]. for hosting. It is directly connected with the project repository and for free.

[22] Localhost is a reserved IP address that works as a loopback. Data sent to localhost do not leave the computer, but they appear to come from the network.

[23] GitHub Pages overview is accessible on `https://pages.github.com/`.

Launch of the Application. To launch the application on your localhost, follow these steps:

1. install *Node.js*[24]
2. install *npm* [13]
3. open your terminal in the folder with project source files
4. in the terminal, run `npm install` to install all project dependencies
5. in the terminal, run `npm start` to launch the application in your browser
6. you can edit the files and see updates realtime in the browser

[24] Node.js is accessible on `https://nodejs.org/en/`.

Chapter 6

User Documentation

In this chapter, we present the application usage. We named the final version of the application as **Rachel** (**R**elational **A**lgebra **CH**ecker and **E**va**L**uator).

We start with a description of basic terms we will use in the following text:

- *Editable relations* – They are the relations we can edit. They are not available for usage in queries directly (see *Loaded relations* below).
- *Selected relation* – It is the editable relation that is edited in a given moment. The application displays it in the container in the upper part of the screen.
- *Loaded relations* – They are available for usage in queries. We create them from valid editable relations by loading.
- *Expressions* – There may be multiple relational algebra expressions in the application.
- *Selected expression* – It is the expression that is edited in a given moment. The application displays it in the text area in the second part of the screen.
- *Project* – We use the project to save/load our work using *.rachel*[1] file. It stores editable relations, expressions, and an indicator of whether we assume `null` values.
- *Application* – The application always contains one project, which can be saved or overwritten by loading a new one. Furthermore, the application provides additional settings or batch processing of multiple project files.

Typical Workflow. Now, we will describe the high-level workflow in the application. We will describe particular parts in depth in the following paragraphs.

When using the application for the first time, we must prepare our relations first. To do so, we use the relation section of the page. In the relation section, we can create new editable relations, delete them, or import/export them using CSV files. At each time, we can edit the selected relation in

[1] The *.rachel* extension describes JSON files generated by the application.

the container. Once we prepare (valid) relations, we load them into the application.

After loading the relations, we can use them in query expressions. In the expression section, we can create new expressions, delete them, or import/export them using a text file. Each time, we can edit the selected expression in the text area. Once we are done with editing, we can evaluate the selected expression.

After the evaluation of the selected expression, the application displays its evaluation tree and result in the bottom part of the page. We can use the evaluation tree to browse intermediate relations created during the evaluation.

Anytime in the described process, we can save the project to a file and continue later on.

Relation Section. We define relations in a sheet in the upper part of the screen. To be able to use a relation in the expressions, we need to load it to the application when all its values are valid. After loading, we can continue editing the relation while the last loaded (valid) state is still available in the expressions. We show this section in Figure 6.1.

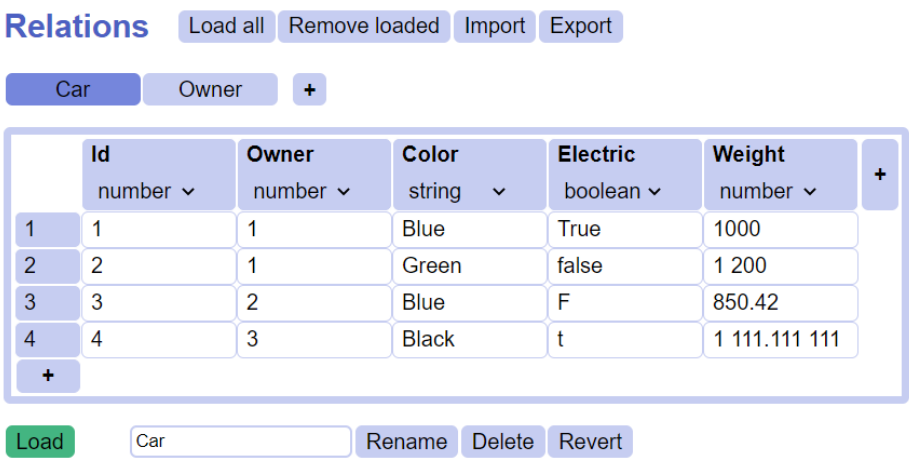


Figure 6.1: Relation section

There are four buttons in the header menu, which affect all relations:

- The *Load all* button loads all valid editable relations into the application memory. If any loaded relation with the same name exists, it is overwritten. Invalid relations are skipped.
- The *Remove loaded* button removes all loaded relations (editable relations are not changed).
- The *Import* button enables us to import new editable relations from CSV files.
- The *Export* button saves all editable relations in CSV files. The saved relations may be in an invalid state.

In the menu above the sheet, we can select one relation to be edited. A star before the relation name marks changed relations since the last loading. We can add a new editable relation by the + button.

In the first row of the sheet, we define column names and types. Column names cannot be duplicated inside one relation and must contain letters, numbers, and underscores only and not start with a number. Also, column names "null", "true" and "false" are forbidden. There are three supported column types in the application: `number`, `string`, and `boolean`.

The + buttons in the last column and last row adds a new column or row, respectively.

Other sheet cells define the data itself. We can use integers or decimals in number columns and character sequences in string columns. Note that the application trims trailing whitespaces before loading, so the string `" a b c "` is loaded as `"a b c"`. If `null` values are supported, "null" inputs are valid in all column types and are loaded as `null` values. Also, empty inputs in number and boolean columns are loaded as `null` values.

When the cursor hovers over the first row, a button for deleting a given column appears. Similarly, when over the first column, a button for deleting a given row appears.

There are four buttons in the menu under the sheet, which affect the selected relation:

- The *Load* button loads the relation into the application memory. If any loaded relation with the same name exists, it is overwritten.
- The *Rename* text field renames the relation. We cannot change the name to any existing relation name. Allowed characters are the same as in column names, but forbidden words are "F", "L", and "R".
- The *Delete* button deletes the relation from the editable list.
- The *Revert* button reverts the relation to the last loaded state (if the relation was not loaded yet, it is reverted to its initial state).

Expression Section. The second section of the application provides the input for expressions. We show it in the Figure 6.2.

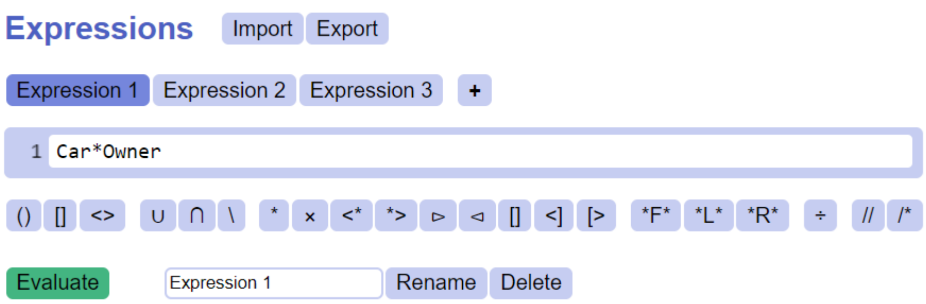


Figure 6.2: Expression section

There are two buttons in the header menu, which effects all expressions:

- The *Import* button enables us to load new expressions from text files.
- The *Export* button saves all expressions in a text file.

We can have multiple named expressions loaded in the application at a time. Again, we use the upper menu for selecting an expression to edit and the + button for adding a new one.

In the text area, we define the expression itself. We can use buttons under the text area to insert RA operators. While typing, the application suggests relation or column names available at the cursor position. We can use *arrows/Enter* keys or *mouse* to insert the suggestion. Pressing *Ctrl+Space* hides or displays the suggestions list.

To use quote characters inside string literals in expressions, we must escape their default behavior (i.e., starting or ending a string) by a backslash. Similarly, to use a backslash character, we must type it twice.

There are three buttons on the bottom of the section, which affect the selected expression:

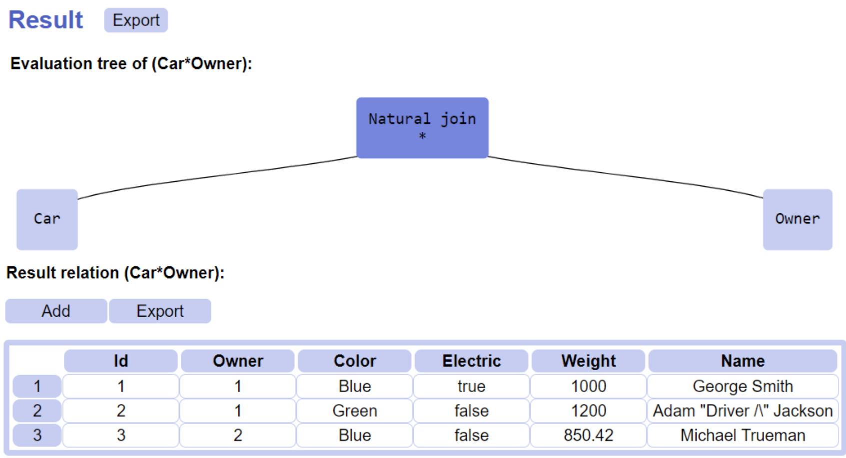
- The *Evaluate* button evaluates the selected expression and updates the result section.
- The *Rename* text field renames the selected expression. There are no restrictions on expression names.
- The *Delete* button deletes the selected expression.

Result Section. The result section appears after the evaluation of an expression. It displays the evaluation tree and the result relation. Moreover, for every individual operation node within the tree, we can display a relation with data corresponding to a given intermediate result. We can also sort the rows in a relation using the specific column values. We show the result section in Figure 6.3.

The *Export* button above the evaluation tree downloads the tree as a PNG image. We can use the *Add* and *Export* buttons above the table to add the displayed relation to the editable ones or save it in a CSV file.

Management Section. The last-mentioned section is the upper one. It provides general management of the application.

- The *Load* button loads the whole project from *.rachel* files. Rachel files contain all editable relations, all expressions, and a configuration value indicating whether usage of `null` meta values is enabled.
- The *Save* button saves the current project to a new *.rachel* file.
- The *Batch* button lets us select multiple project files to be all processed and their reports generated.



	Id	Owner	Color	Electric	Weight	Name
1	1	1	Blue	true	1000	George Smith
2	2	1	Green	false	1200	Adam "Driver /\" Jackson
3	3	2	Blue	false	850.42	Michael Trueman

Figure 6.3: Result section

- The *Samples* button shows prepared sample projects. It is a convenient starting point for users who are just getting acquainted with Rachel.
- In the *Settings*, we can set:
  - *Null values support* – whether the project supports `null` values in relations and expressions
  - *CSV separator* – used value separator in downloaded CSV files
  - *Theme* – the theme of the application (light/dark)
  - *Language* – the language of the application (English/Czech)
- The *About* button navigates to the project repository.

Operators. Rachel provides a wide set of relational algebra operations[2]. In the following list, we show their syntax and precedence (lower numbers mean higher precedence). Anyway, we recommend using parentheses to avoid unexpected precedence behavior.

- Precedence level 1 – unary operations:
  - *Projection*: `Relation[column, ...]`
  - *Selection*: `Relation(condition)`
  - *Rename*: `Relation<Old -> New, ...>`
- Precedence level 2 – joins and division:
  - *Natural join*: `A * B`
  - *Cartesian product*: `A` × `B`
  - *Left/right semijoin*: `A <* B, A *> B`

[2] The application uses simplified syntax as defined in Section 2.2. There are little differences in rename and outer join operators as they use symbols available on keyboards.

  - *Left/right antijoin*: `A ▷ B, A ◁ B`
  - *Theta join*: `A [condition] B`
  - *Left/right theta join*: `A ⟨condition] B, A [condition⟩ B`
  - *Full/left/right outer join*: `A *F* B, A *L* B, A *R* B`
  - *Division*: `A ÷ B`

- Precedence level 3 – intersection:
  - *Intersection*: `A ∩ B`

- Precedence level 4 – union and difference:
  - *Union*: `A ∪ B`
  - *Difference*: `A \ B`

We use algebraic operators (+, -, *, /) in the conditions to calculate new number values. If a number column evaluates to `null`, `null` is returned. Other input types trigger an error.

Comparison operators (==, !=, <, >, <=, >=) accept any pair of input operands of the same type and produce a boolean value, i.e., `true` or `false`. Inequality checking of booleans uses `false < true`. Inequality checking of strings uses alphabetic comparison (e.g., `"abc" < "def"`, `"a" < "aa"`). If a column evaluates to `null`, the only condition which returns true is `column == null`. Different input types trigger an error. There are two ways to write equality (==, =) and inequality (!=, <>) operators.

We can use boolean values in selection and theta semijoins with no testing operator (e.g., `Relation(BooleanColumn)`). Theta joins always require some testing operator (e.g., `RelA[BooleanColumn = true]RelB`).

Logical operators (not, and, or) accept boolean values and computes a new boolean value. When a column of boolean type evaluates to `null`, it holds: `!column == false`, `column && boolean == false` for any boolean value, and `column || boolean == boolean` for any boolean value. Other input types trigger an error. There are three ways to write logical operators: negation (!, ~, ¬), and (&&, &, ∧), or (||, |, ∨).

Tips. In expressions, we can use C-style line and block comments.

We can use *Ctrl+Enter* in the relation table to load the current relation. In the expression textarea, we can use it to evaluate the current expression.

All tabulators loaded into the application in files are replaced by four spaces. In case of editing the files outside Rachel, we recommend using spaces to ensure expected indenting.

We can use a mouse to move relations and expressions in their menus.

Known Issues. The application does not support the *Internet Explorer*[3] browser. We decided not to support it as Microsoft recommends using newer browsers and announced the end of its support as well.

[3]Internet Explorer was the major Microsoft browser in the previous Windows operating systems. Nowadays, Microsoft recommends using the *Microsoft Edge* browser.

Chapter 7

Conclusion

The goal of this thesis was to design and implement an application for the evaluation of relational algebra query expressions. Before we started our implementation, we analyzed similar existing tools. First, we found out that no application supports the simplified notation suitable for learning relational algebra. We identified advantages and disadvantages of three applications: Raeval [3], RelaX [4], and RAT [5]. Their common disadvantage is a weak reusability of defined relations, expressions, and evaluation results. Furthermore, their error descriptions are often insufficient. The analysis confirmed that a user-friendly application should have a graphical user interface, be accessible online, and provide a detailed user manual.

Having analyzed existing applications, we created a list of requirements for our implementation. We used them together with the described business processes to identify business entities in our solution and its general concept. For example, we defined two types of relations – one for evaluation (i.e., it follows the formal definition) and one for editing (i.e., it has ordered rows and may contain invalid values). Finally, to properly design the application, we created detailed use cases and a class model.

The core aspect of the application is the parsing of relations and expressions and evaluation of queries. Although it might seem trivial, many difficulties appeared during the implementation. The biggest challenges related to user-friendly highlighting of errors and suggestions in expressions. We needed to correctly parse unfinished expressions to determine available suggestions while users are in the middle of typing. To do so, we implemented their automatic completion and fixing. Furthermore, we needed to map characters to their positions in the original string to highlight invalid parts. For this purpose, we implemented an extended a string data type capable of storing the original positions of individual characters.

Contributions. The implemented application, named **Rachel**, is a web teaching tool ready for usage. It is available at `https://kotliluk.github.io/rachel/`. Its main features are:

- user-friendly graphical user interface
- evaluation of relational algebra query expressions

- built-in environment for relation definition and editing
- visualization of evaluation trees and intermediate results
- support of the relational algebra definition assumed in this thesis, i.e., 19 operations, column names without relation prefixes, atomic values
- support of the simplified operator notation
- errors with detailed messages and highlighting in the text
- suggestions of available relations or columns in particular parts of expressions
- possibility to edit multiple expressions at a moment
- wide range of import/export actions, e.g., for relations, expressions, or whole project with all relations and expressions
- automatization of homework processing, i.e., loading multiple student projects, evaluating them, and generating reports

Future Work. Although about 20 students used the application in the testing phase, not yet revealed errors or further user suggestions might appear the next year, when roughly two hundred students will use it on FEE, CTU. We plan to respond to the teaching needs that might come after this more extensive use.

We also have several ideas for further improvements, e.g., the possibility to support duplicate data rows, the date data type, or advanced functions in conditions as the length of strings. However, we do not find them crucial, and so we did not implement them yet. Furthermore, we will release the application as an open-source project so that interested people can contribute.

Bibliography

[1] DATE, C.J. *An Introduction to Database Systems. 8th ed.* Boston: Pearson/Addison-Wesley, 2003. ISBN 0-321-19784-4.

[2] SVOBODA, Martin. *Database Systems – Relational Algebra Lecture* [online]. 2020-03-31 [cit. 2021-04-28]. Accessible from: `https://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/lectures/Lecture-07-Relational-Algebra.pdf`.

[3] EVERITT, Nick. *Raeval – Relational Algebra Evaluator* [computer application]. Versions 2.0 and 0.3.1. Wilmington (North Carolina, USA), 2011-2012 [cit. October, 2020]. Free download from: `https://code.google.com/archive/p/relational-algebra/`, or `http://people.uncw.edu/narayans/courses/csc455/Relational%20Algebra/Relational%20Algebra%20Interpreter.html`. Requires installed Java.

[4] KESSLER, Johannes. *RelaX – Relational Algebra Calculator* [online application]. Version 0.20. Innsbruck, 2020 [cit. October, 2020]. Accessible from: `https://dbis-uibk.github.io/relax/landing`.

[5] CHAVARRÍA, Steven Brenes. *RAT – Relational Algebra Translator* [computer application]. Version 4.2.0.0. San José (Costa Rica), 2011 [cit. October, 2020]. Free download from: `https://www.slinfo.una.ac.cr/rat/rat.html`. Requires installed .NET framework version v2.0.

[6] GUO, Shu-yu, Michael FICARRA and Kevin GIBBONS, ed. *ECMAScript® 2022 Language Specification* [online]. 2021-04-06 [cit. 2021-04-21]. Accessible from: `https://tc39.es/ecma262/`

[7] *TypeScript: The starting point for learning TypeScript* [online]. [cit. 2021-04-21]. Accessible from: `https://www.typescriptlang.org/docs/`

[8] *React – A JavaScript library for building user interfaces* [online]. [cit. 2021-04-21]. Accessible from: `https://reactjs.org/`

[9] *Create React App* [online]. [cit. 2021-05-05]. Accessible from: `https://create-react-app.dev/`

[10] FAULKNER, Steve, EICHOLZ, Arron, LEITHEAD, Travis, DANILO, Alex, MOON, Sangwhan, NAVARA Erika Doyle, O'CONNOR Theresa, BERJON, Robin. *HTML 5.2 - W3C Recommendation* [online]. W3C, 2021-01-28 [cit. 2021-05-05]. Accessible from: https://www.w3.org/TR/html52/

[11] BOS, Bert. *All CSS specifications* [online]. W3C, 2021-01-20 [cit. 2021-05-05]. Accessible from: https://www.w3.org/Style/CSS/specs.en.html

[12] Q-SUCCESS. *Historical yearly trends in the usage statistics of client-side programming languages for websites, April 2021* [online]. 2021-04-21 [cit. 2021-04-21]. Accessible from: https://w3techs.com/technologies/history_overview/client_side_language/all/y

[13] NPM, Inc. *npm Docs* [online]. [cit. 2021-05-05]. Accessible from: https://docs.npmjs.com/

[14] *Jest - Delightful JavaScript Testing* [online]. 2021-05-03 [cit. 2021-05-05]. Accessible from: https://github.com/facebook/jest

[15] DIJKSTRA, E. W. *Algol-60 Translation.* Amsterdam: Stichting Mathematisch Centrum, 1961. Accessible from: https://ir.cwi.nl/pub/9251

Appendix A

Attachment Content

- `analysis/` – models for the analysis
- `code/` – source codes of the application
- `documentation/` – generated HTML documentation for source codes
- `examples/` – prepared files with projects, relations, or expressions to import into the application
- `thesis/` – LaTeX source of the thesis text