UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Relational conditional set operations

Alexis Iván Aspauza Lescano

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

Alexis Iván Aspauza Lescano

Relational conditional set operations

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Robson Leonardo Ferreira Cordeiro

USP – São Carlos
January 2022

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

A838r

Aspauza Lescano, Alexis Iván
Relational conditional set operations / Alexis Iván Aspauza Lescano; orientador Robson Leonardo Ferreira Cordeiro. -- São Carlos, 2021.
105 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2021.

1. Relational Databases. 2. Relational Algebra. 3. Theory of Sets. 4. Conditional Queries. 5. Custom Predicates. I. Ferreira Cordeiro, Robson Leonardo, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Alexis Iván Aspauza Lescano

Operações de conjuntos relacionais condicionais

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Robson Leonardo Ferreira Cordeiro

USP – São Carlos
Janeiro de 2022

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.
Em especial, ao pesquisadores do Instituto de Ciências Matemáticas e de Computação (ICMC).

ACKNOWLEDGEMENTS

First of all, I would like to thank my family for their unconditional love and support. Then, I want to thank my advisor Prof. Dr. Robson L. F. Cordeiro for his guide and patience. Finally, I would like to thank the financial support from the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) – Finance Code 001, the Fundação de Amparo à Pesquisa do Estado de São Paulo - Brasil (FAPESP) – 2021/05623-2, 2020/07200-9, 2018/05714-5 and 2016/17078-0, and; the Conselho Nacional de Desenvolvimento Científico e Tecnológico – Brasil (CNPq).

“As invenções são, sobretudo,
o resultado de um trabalho de teimoso.”
(Santos Dumont)

RESUMO

ASPAUZA, A. I. **Operações de conjuntos relacionais condicionais**. 2022. 107 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2022.

Um conjunto é uma coleção de objetos distintos entre si. Algumas operações básicas da Teoria dos Conjuntos são a pertinência ($\in$), inclusão ($\subseteq$), intersecção ($\cap$), e diferença ($-$). A Álgebra relacional adapta as operações de conjuntos para trabalhar com relações. No entanto, as operações de conjuntos têm limitações por causa do uso implícito do predicado de identidade. Ou seja, uma tupla é membro de um conjunto se for idêntica a qualquer tupla do conjunto. Por exemplo, vamos considerar duas relações. A primeira é uma lista de produtos que uma pessoa quer comprar. A segunda é uma lista de produtos que uma loja tem. Agora, poderíamos pegar qualquer item da lista de produtos desejados e perguntar "podemos comprar esse item na loja?" com o operador de pertinência ($\in$). Com o operador de pertinência como base, podemos também fazer outras consultas, tais como subconjunto, interseção e diferença. O operador de subconjunto ($\subseteq$) responderia a "posso comprar todos os produtos desejados na loja?". A interceção ($\cap$) responderia a "quais produtos desejados posso comprar na loja?". E, finalmente, a diferença ($-$) responderia a "quais são os produtos desejados que não consigo comprar na loja?". Ainda assim, muitas aplicações precisam de outras formas de comparação que não se limitem à identidade. Por exemplo, se acrescentar os atributos de quantidade e preço aos conjuntos de produtos desejados e aos produtos da loja, a comparação das tuplas por identidade não terá muito sentido, já que um produto na loja com estoque maior do que o exigido deve ser válido, e também é válido um produto com um preço inferior ao orçamento máximo do usuário para esse produto. O presente trabalho apresenta as novas **Operações de Conjunto Relacionais Condicionais**. Os novos operadores encapsulam a ideia de operações de conjunto com consultas condicionais, facilitando operadores específicos para eles e permitindo sua otimização. Por exemplo, eles são potencialmente úteis em aplicações de vendas de produtos com unidades e preços, promoções de empregos com habilidades desejadas e estágios com notas mínimas. Validamos a semântica e a escalabilidade de nossa proposta estudando o primeiro desses aplicativos. Além disso, abrimos caminho para trabalhos futuros como: implementação dos operadores em um SGBD; propor consultas SQL capazes de responder a esse tipo de consulta e compará-las com nossa abordagem atual; estender a ideia para trabalhar com *bag algebra*; estudar a otimização para nossos algoritmos; adicionar suporte para dados complexos, permitindo comparações de similaridade no predicado; e, estudar o uso dos novos operadores como base para outras operações que utilizam a operação de conjunto tradicional como base; entre outros.

Palavras-chave: Operações de conjuntos, Álgebra relacional, Teoria de conjuntos.

ABSTRACT

ASPAUZA, A. I. **Relational conditional set operations**. 2022. 107 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2022.

A set is a collection of different objects. Some basic operations from the Theory of Sets are the set membership ($\in$), subset ($\subseteq$), intersection ($\cap$), and difference ($-$). The relational Algebra adapts the set operations to work with relations. However, as we show in this work, the set operations have limitations because of the implicit use of the identity predicate. That is, a tuple is a member of a set if it is identical to any tuple in the set. For example, let's consider two relations. The first one is a list of products that a person wants to buy. The second one is a list of products that one store has. Now, we could get any item from the desired products list and query "can we buy this item in the store?" with the set membership operator ($\in$), being true if the item is a member of the second set or false if not. With the set membership operator as a basis, we can also perform other queries such as subset, intersection, and difference. The subset ($\subseteq$) query would answer to "can I buy all the desired products in the store?". The intersection ($\cap$) would answer to "what products can I buy in the store?" And finally, the difference ($-$) would answer to "what are the desired products that I cannot buy in the store?". Still, many applications need other comparison predicates that are not limited to identity. For example, if we add quantity and price to the sets of desired products and store's products, comparing the tuples by identity won't have much sense, since a product in the store with stock greater than the required should be valid, and it is also valid a product with a price lower than the user's maximum budget for that product. This MSc work presents the new **Relational Conditional Set Operations**. The novel operators encapsulate the idea of set operations with conditional queries, facilitating specific operators for them, and allowing their optimization. For example, they are potentially useful in applications of product sales with units and prices, job promotions with skills that have enough experience or certification level, and internships with minimum grades. We validate our proposal's semantics and scalability by studying the first of these applications. Also, we open path for future works such as: to implement the operators in a DBMS; to propose SQL queries able to answer these kind of queries and compare it with our current approach; to extend the idea for bag algebra; to explore a whole new path of optimization for our algorithms; to add support for complex data, allowing similarity comparisons in the predicate; and, to study the use of these operators as basis for other operations that currently use the traditional set operation as basis; among others.

Keywords: Set Operations, Relational Algebra, Theory of Sets.

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF SYMBOLS

t — Single tuple

T — Relation

$\in$ — Set membership

$\subseteq$ — Subset

$\cap$ — Intersection

$-$ — Difference

${}_c\mathsf{T}$ — Relational Conditional Set

$\in_c$ — Relational Conditional Set membership

$\subseteq_c$ — Relational Conditional Subset

$\cap_c$ — Relational Conditional Intersection

$-_c$ — Relational Conditional Difference

CONTENTS

CHAPTER

1

INTRODUCTION

In this chapter, we discuss the motivation to propose the new operators, exemplifying the limitations of the traditional relational set operations. For this, we explore a case study, showing the expansion of our new operators which are named as relational conditional set operations, or "relcond set operations" for short. They are based on the definition of custom predicates used to compare tuples.

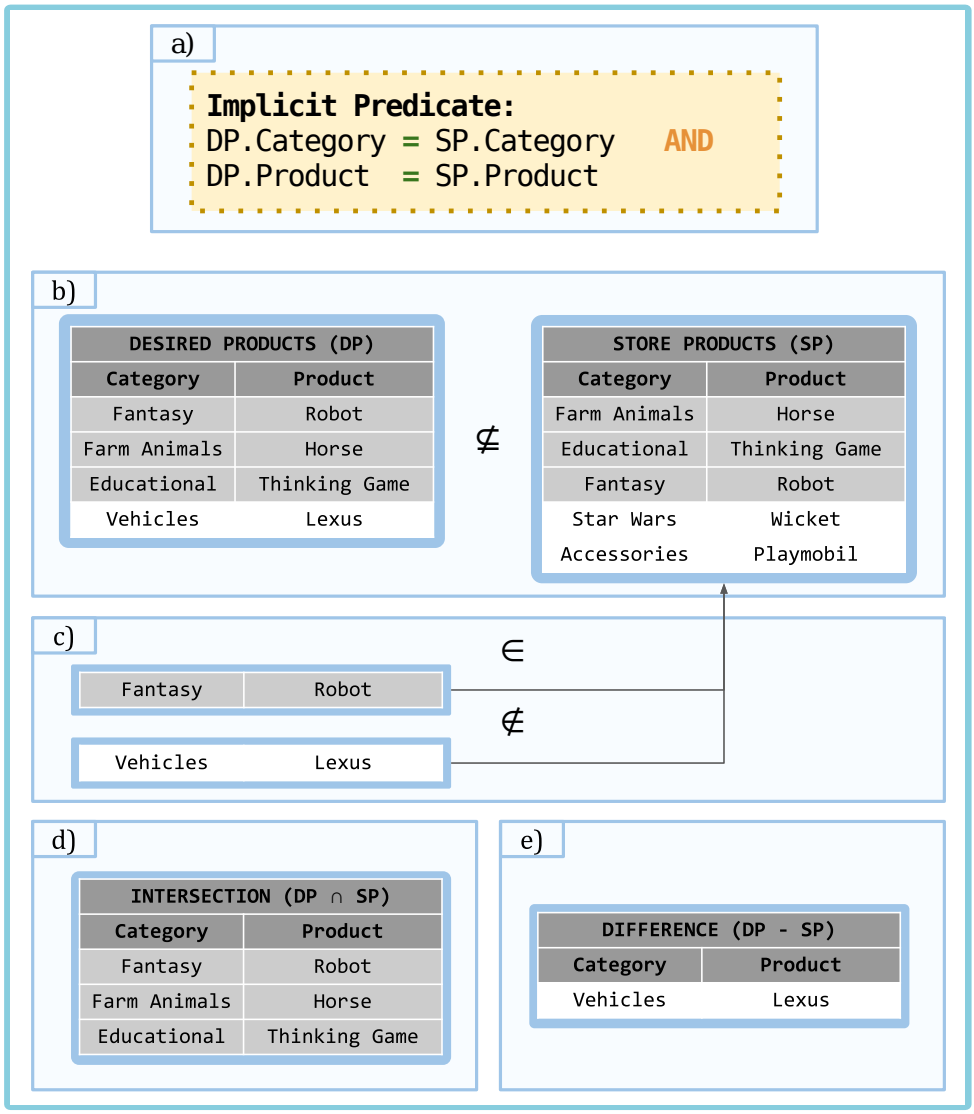
1.1 Context

The set membership ($\in$), subset ($\subseteq$), intersection ($\cap$), and difference ($-$) are basic operations from the Theory of Sets (STOLL, 1963). A set can be defined as a collection of **different** objects or elements. In the relational model, we can define a relation as a collection of tuples. Also, the Relational Algebra (CODD, 1972; CODD, 1990; ELMASRI; NAVATHE, 2015) employs the set operations to work with relations and their usability is very intuitive. For example, in Figure 1.b we have a relation with Desired Products (DP) of a user, and another one containing a Store's Products (SP) available. Now, with the traditional set operators, we can answer the four next queries:

Q1: *Can I buy a certain product X in the store?* – it can be answered with the **set membership** operator ($\in$). A tuple **is a member** of a relational set if it is identical to any tuple in the set. For example, in Figure 1.c the tuple $(Fantasy, Robot)$ is a member of *SP* since *SP* contains an identical tuple $(Fantasy, Robot)$. In contrast, the tuple (Vehicles, Lexus) is not a member of SP since SP does not contain any identical tuple to it.

Q2: *Can I buy all the desired products in the store?* – it can be answered with the **subset** operator ($\subseteq$). A set is a subset of another one if all of its elements are also members of the second set. For example, in Figure 1.b, *DP* is not a subset of *SP* because the tuple $(Vehicles, Lexus)$ of *DP* is not a member of *SP*.

Figure 1 – Example of the relational set operations.



Source: Elaborated by the author.

Q3: *Which desired products can be bought in the store?* – it can be answered with the **intersection** operator ($\cap$). Figure 1.d illustrates that the result of $DP \cap SP$ is a new set with all tuples from DP that are also members of SP, i.e., the desired products that are available in the store. In this case, the resulting set is $\{(Fantasy, Robot), (Farm\ Animals, Horse), (Educational, Thinking\ Game)\}$.

Q4: *Which desired products won't be found in the store?* – it can be answered with the **difference** operator ($-$). Figure 1.e shows that the result of $DP - SP$ is a new set with all tuples from DP that are not members of SP, i.e., the desired products unavailable in the store. In this case, we have the set: $\{(Vehicles, Lexus)\}$.

We have just introduced an example of sales of products. In this example, a client wants to buy a list of products from a store. Here, we have category and name for each product, and

the tuples were compared by identity. However, what if we take this example, and add other two attributes in each relation: *desired units* by the client vs *available stock* at the store, and *maximum price* that the client is willing to pay vs *tag price* for that product at the store. Here, comparing elements by identity would be senseless. Thus, we will explain our motivation in the next section.

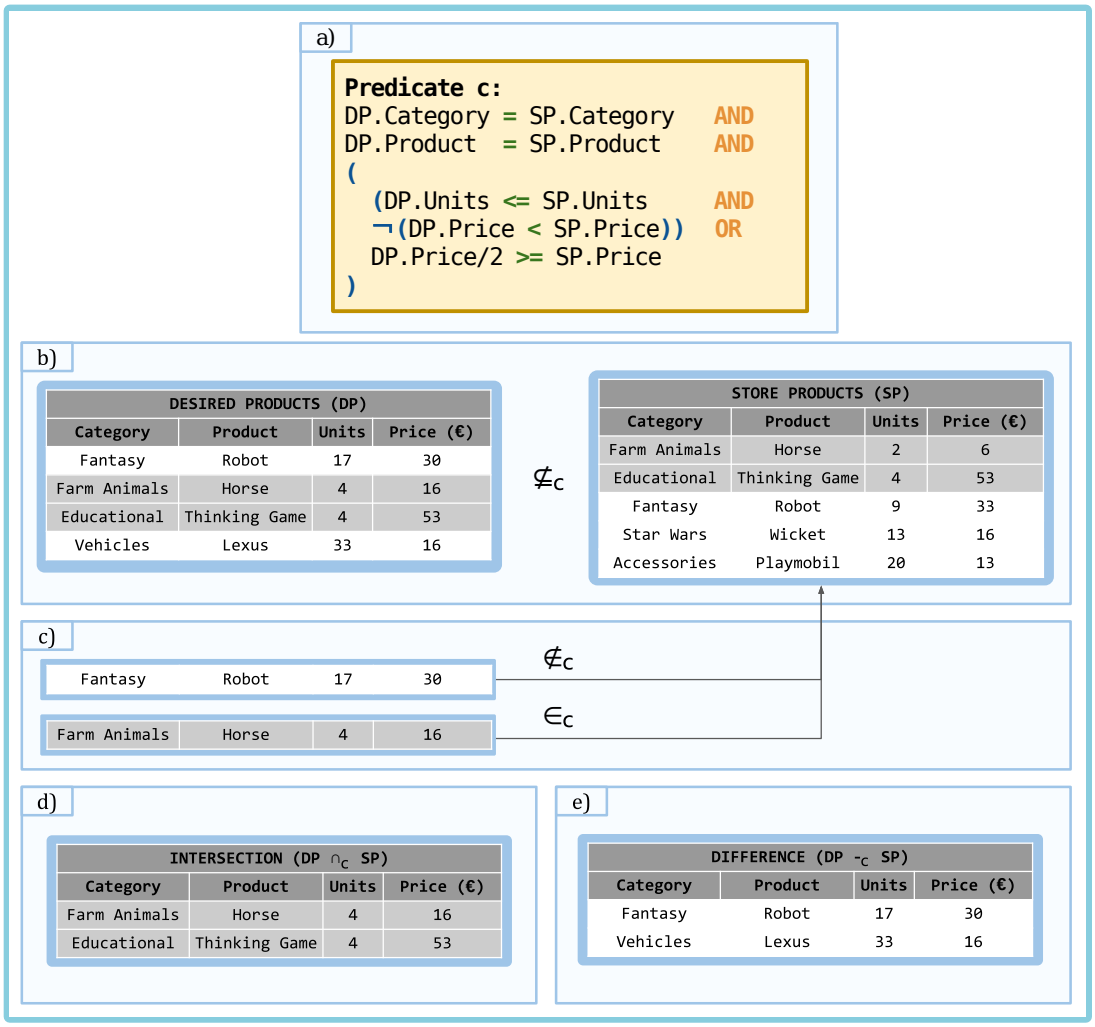
1.2 Problem and Motivation

The traditional set operators are very useful when elements are compared by identity. That is, when it makes sense to compare elements by the implicit identity predicate that says that a tuple is a member of a relation if and only if all of its attributes' values are identical to all the corresponding values of any tuple in the set. This predicate was illustrated in Figure 1.a. However, this implicit predicate has severe limitations. Note that instead of comparing tuples by identity, we could easily expand the comparison predicate to support any custom predicate with **and** and **or** connectors, $=, <, \leq, >$, and $\geq$ logical operators, $+, -, *, \div$ arithmetic operators, and negations $\neg$. For instance, let us expand our original example of sales of products with categories and producs' names. Now, let's add two more attributes to both sets *DP* and *SP*: units and price. This is shown in Figure 2.b, where "DP.units" refers to the client's desired number of units and "SP.units" refers to the stock available in the store. Also, "DP.price" is the maximum client's budget for that product, while "SP.price" is the product price in the store. Let us assume also that the client can accept fewer units than the desired ones if the price is at most half of his/her budget. Now, to compare tuples by identity would not be useful. We need to employ a custom predicate to select products with the client's preferences, i.e., having enough units and acceptable price, or simply low price. The predicate c is shown in Figure 2.a. This work then, encapsulates the concept of set operations with conditional queries into the new **Relational Conditional Set Operations**. This is, we aim to facilitate the specification of these kind of operations as well as make possible their optimization. This is analog to the join operation, which is conceptually a Cartesian product with a projection, but it is also an specific operator and more efficient that implement it by using the other two operations. With these concepts, let's analyze how our previous queries are adapted and how our operators allow us to use conditional queries to answer them:

Given a predicate which represents our desired condition for each product, ...

Q1: *... can I buy a certain product X in the store?* – it could be answered with the **RelCond Set Membership** operator ($\in_c$). One tuple x **is a relcond$_c$ member** of one set T if, given a predicate c, the evaluation $c(x,y)$ is true for one tuple $y \in \mathsf{T}$. For example, in Figure 2.c, tuple $a = (FarmAnimals, Horse, 4, 16)$ is a relcond$_c$ member of *SP*, since *SP* contains a tuple $y = (FarmAnimals, Horse, 2, 6)$ such that $c(a,y)$ is true. By contrast, tuple $b = (Fantasy, Robot, 17, 30)$ is not a relcond$_c$ member of *SP* because there is not a tuple y

Figure 2 – Example of the relational conditional set operations.



Source: Elaborated by the author.

in SP where $c(b,y)$ is true. We can also follow this idea to define a **RelCond Set** ${}_c\top$ as a set in which $c(x,y)$ is false for any pair of tuples $x,y \in {}_c\top$.

Q2: *... can I buy all the desired products in the store?* – it could be answered by the **RelCond Subset** operator ($\subseteq_c$). A set is a relcond$_c$ subset of another set if all of its elements are relcond$_c$ members of the second set. For example, in Figure 2.b, DP is not a subset of SP as there are tuples in DP that are not members of SP, i.e., (Fantasy, Robot, 17, 30) and (Vehicles, Lexus, 33, 16).

Q3: *... which desired products can be bought in the store?* – it could be answered with the **RelCond Intersection** operator ($\cap_c$). Figure 2.d illustrates the result of $DP \cap_c SP$, which is a new set with all tuples from DP that are relcond$_c$ members of SP. These are the desired products that are available in the store.

Q4: *... which desired products won't be found in the store?* – it could be answered with the **RelCond Difference** operator ($-_c$). Figure 2.e shows the result of $DP -_c SP$, which is a

new set with all tuples from *DP* that are not relcond$_c$ members of *SP*. These are the desired products that are unavailable in the store.

As we discussed in this section, the use of customized predicates have prompt applications in real life. However, operators that support them have not been defined in the literature. In this regard, this MSc work focuses on providing support to the Relational Conditional Set Operators, exploring the hypothesis as follows:

> **Hypothesis:** Custom predicates applied to set operations are valuable tools to process relational set operations with conditional queries.

1.3 Contributions

This MSc work investigates Hypothesis 1.2 focused on querying relational set operations with customized predicates. Specifically, we present the new **RelCond Set Operators** ($\in_c, \subseteq_c, \cap_c, -_c$). Our main contributions are:

C1 Operators Design and Usability – we identified severe limitations on the usability of the traditional set operations, which are caused by the use of implicit identity predicates. Thus, we tackled the problem by extending these operations into our RelCond Set Operators ($\in_c, \subseteq_c, \cap_c, -_c$) that are naturally well suited to answer queries in sets with customized predicates.

C2 Formal Definition and Algorithms – we formally defined the new operators as the RelCond Set Membership ($\in_c$), RelCond Subset ($\subseteq_c$), RelCond Intersection ($\cap_c$), and RelCond Difference ($-_c$), thus enabling their use in queries along with the existing algebraic operators. Also, we designed novel algorithms to execute these new operations in a fast and scalable manner.

C3 Semantic Validation – we performed a case study by analyzing real data from thousands of toy products available for sale at Amazon[1]. The results corroborate the practical usability of our operators in real-life applications.

C4 Generality and Usability – we present other applications where our operators are well suitable, thus corroborating their general usability.

C5 Basis for other operators – our new operators can also be used as the basis for other relational algebra operations. For example, in Appendix A, we discuss the utility of generalizing the relational division, and present a new operator that can also support custom predicates, using as a basis our relational conditional set operations.

[1] <www.amazon.com>

Reproducibility: for the purpose of reproducibility, all codes, results, and datasets studied in this work are freely available for download online[2].

1.4 Organization

The rest of this monograph follows a traditional organization: background concepts (Chapter 2), related work (Chapter 3), proposed operators (Chapter 4), algorithms (Chapter 5), experiments (Chapter 6), discussion (Chapter 7), and conclusions and future work (Chapter 8). Additionally, we also present the generalization of the relational division in Appendix A.

[2] <https://github.com/alivasples/RCSetOp>

CHAPTER

2

BACKGROUND CONCEPTS

In this chapter, we present the fundamental concepts related to this MSc work. The chapter begins with a review of the Theory of Sets followed by a discussion of the Relational Algebra. We also present a brief review about infix and postfix notations of arithmetic expressions. The relevance of the discussion of Theory of Sets and Relational Algebra is very intuitive since this MSc work proposes to extend the operators from the Theory of Sets applied to relations to support custom predicates. Precisely, these custom predicates are given by expressions. Thus it is important to read and process them in an efficient manner, that is the reason we also discuss the most common forms to represent expressions, such as infix and postfix notations.

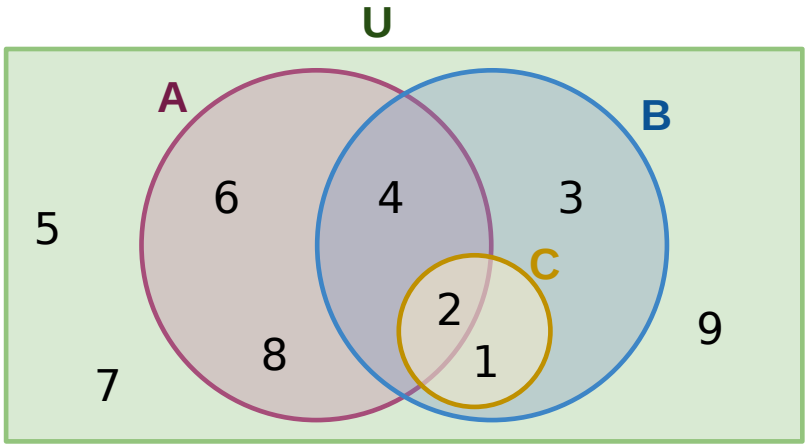
2.1 Theory of Sets

A set is a collection of ***distinct*** objects, typically of the same type, which are called the elements or points of the set (UNDE; KURHE, 2018; ALSALLAKH *et al.*, 2016; RAMSDEN, 2004). A key characteristic of this collection is that it does not impose an ordering of the elements. There are different forms in which a set can be represented. One of the most basic is to represent it by enumeration. For instance, we can represent the following sets:

- Let $\mathbb{U}$ be a set of positive integer numbers lower than 10:
  $\mathbb{U} = \{1,2,3,4,5,6,7,8,9\}$.
- Let A be a set of pair numbers lower than 10:
  $A = \{2,4,6,8\}$.
- Let B be a set of positive integer numbers lower than 5:
  $B = \{1,2,3,4\}$.
- Let C be a set of the first 2 natural numbers:
  $C = \{1,2\}$.

Another representation of a set can be through a Venn Diagram. For example, we can represent our examples with the set illustrated in Figure 3.

Figure 3 – Sets represented in a Venn Diagram.



Source: Elaborated by the author.

2.1.1 Basic concepts

The Theory of Sets (STOLL, 1963; PINTER, 1971) defines some fundamental relationships between elements and sets, but also between sets and sets.

1. **Set membership ($\in$):** The membership relationship between objects and sets denotes whether an element is an element of a set. For example, from Figure 3 we can say that 8 is a member of set A ($2 \in A$). In an opposite case, 1 is not a member of set A ($1 \notin A$).

2. **Subset ($\subseteq$):** The subset relationship between two sets denotes whether a set is a subset of another. One set is a subset of another, or one set is included in another if all its elements are members of the second set. For example, from Figure 3 we can say that set C is a subset of set B ($C \subseteq B$) since all elements of C (1 and 2) are members of B. However, C is not a subset of A ($C \not\subseteq A$) since there is at least one element in C, in this case, 1, that is not a member of A.

2.1.2 Algebra of sets

The Theory of Sets also defines some basic operations for sets (STOLL, 1963; PINTER, 1971), from them, the most relevant for our research are:

1. **Union ($\cup$):** The union of two sets S_1 and S_2, symbolized by $S_1 \cup S_s$ and read "S_1 union S_2" is the set of all objects that are members of either S_1 or S_2. It is important to point that

the result is another set, consequently, there are not repeated elements. For example, from Figure 3, $A \cup B = \{1,2,3,4,6,8\}$.

2. **Intersection ($\cap$):** The intersection of two sets S_1 and S_2, symbolized by $S_1 \cap S_2$ and read "S_1 intersection S_2" is the set of all objects that are members of both S_1 and S_2. For example, from Figure 3, $A \cap B = \{2,4\}$. If $S_1 \cap S_2 = S_1$ we can say that $S_1 \subseteq S_2$.

3. **Difference ($-$):** The difference or relative complement of two sets S_1 and S_2, symbolized by $S_1 - S_1$ and read "S_1 minus S_2" is the set of all objects that are members of S_1 but not S_2. For example, from Figure 3, $A - B = \{1,3,5\}$.

4. **Complement ($\overline{S}$):** The complement of a set S, symbolized by $\overline{S}$ is the set of all objects in the universal set that are not members of S. In order to define this set, it is important to know what is the universal set. For example, from Figure 3, $\overline{A} = \{1,3,5,7,9\}$.

5. **Symmetric Difference ($\triangle$):** The symmetric difference of two sets S_1 and S_2, symbolized by $S_1 \triangle S_2$, is the union of the differences $S_1 - S_2 \cup S_2 - S_1$. This is, all elements that are members of S_1 or S_2 but not of both. For example, from Figure 3, $A \triangle B = \{1,3,6,8\}$.

6. **Cartesian Product ($\times$):** The Cartesian product of two sets S_1 and S_2, symbolized by $S_1 \times S_1$ is the set of all pairs of objects (a,b) where $a \in S_1$ and $b \in S_2$. For example, from sets B and C of Figure 3, $B \times C = \{(1,1),(1,2),(2,1),(2,2),(3,1),(3,2),(4,1),(4,2)\}$.

2.2 Relational Algebra

The Relational Algebra (CODD, 1972) is defined as a set of operations, not necessarily binary, that are performed on relations resulting in another relation, which are suitable for selecting data from a relational database.

According to Garcia-Molina, Ullman and Widom (2008), an algebra, in general, consists of atomic operators and operands, for example, when we work with real numbers, atomic operands are variables as x and constants as 15. The operators are the usual arithmetic ones: addition, subtraction, multiplication, and division. Any algebra allows us to construct expressions by applying operators to atomic operands and/or other algebraic expressions. Usually, parentheses are required to group operators and their operands. For example, in arithmetic we have expressions such as $(x+y)*z$ or $((x+7)/(y-3))+x$.

The Relational Algebra is another example of an algebra. Its atomic operands are variables that stand for relations and constants, which are finite relations. The Relational Algebra, then, consists of simple but powerful ways of building new relations from others. Expressions in relational algebra begin from relations as operands.

Some operations included in the relational algebra (ELMASRI; NAVATHE, 2015; RAMAKRISHNAN; GEHRKE, 2000; GARCIA-MOLINA; ULLMAN; WIDOM, 2008; ULLMAN,

1997; YU; MENG, 1998; CODD, 1972; CODD, 1990) are the unary operations: renaming(ρ), selection (σ), and projection (π); the set operations from Theory of Sets: union ($\cup$), intersection ($\cap$), set difference (-), and cartesian product ($\times$); and the binary operations: join ($\bowtie$) and division ($\div$). We will discus those operations in this section.

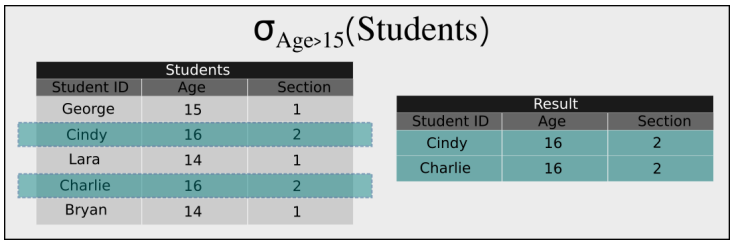
2.2.1 Unary Operations

The renaming operation (ρ)

This operation does not affect the tuples of the relation but changes the schema of the relation, for example, the names of attributes or even the name of the relation itself.

Selection (σ) The selection operator (ULLMAN, 1997), applied on a relation T, produces a new relation with a subset of the tuples of T. The tuples in the resulting relation are those that satisfy some condition c that implies the attributes of T. This operation is denoted as $\sigma_c(T)$. The schema of the resulting relation is the same as the schema of T and the attributes are conventionally displayed in the same order as those of T. An example of Selection is presented in Figure 4.

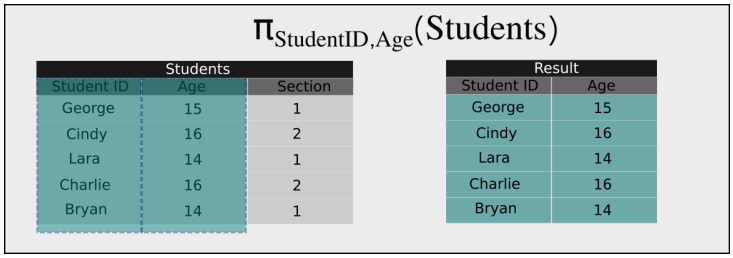
Figure 4 – Example of the selection from a relation.



Source: Elaborated by the author.

Projection (π) The projection operator (ULLMAN, 1997) is used to produce from a relation T, a new relation that has only some columns of T. The value of the expression $\pi_{A_1,A_2,\ldots,A_n}(T)$ is a relation that only has the columns of the attributes $A_1, A_2, \ldots, A_n$ of T. The schema of the result is a set of attributes $A_1, A_2, \ldots, A_n$, which are normally displayed in the order listed. An example of Projection is presented in Figure 5.

Figure 5 – Example of the projection from a relation.



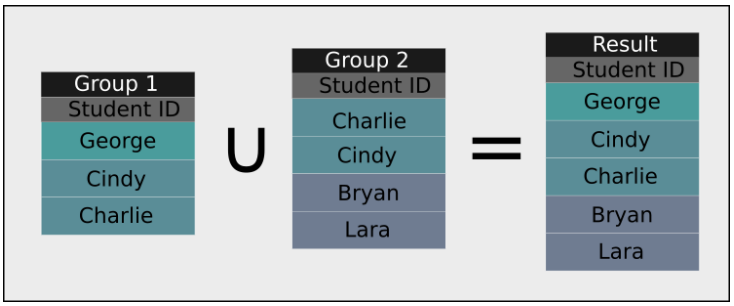
Source: Elaborated by the author.

2.2.2 Binary Operations

2.2.2.1 Operations from Theory of Sets

Union (∪). The union (ULLMAN, 1997) of two union compatible relations T_1 and T_2 is the set of elements that are in T_1 and T_2 or in both relations. Two relations are union compatible if they both have the same number of attributes and each attribute from T_1 has the same domain of its counterpart in T_2. Each element only appears once in the resulting relation, even if it belongs to both T_1 and T_2. An example of Union is presented in Figure 6.

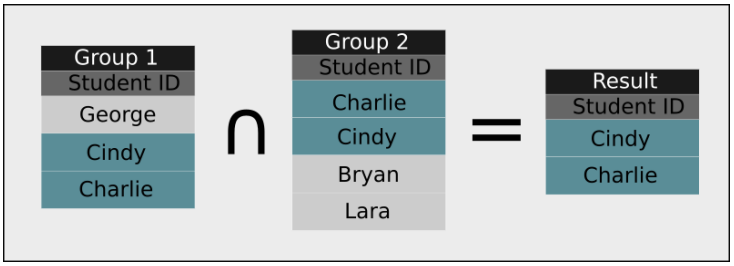
Figure 6 – Example of the union of two relations.



Source: Elaborated by the author.

Intersection (∩). The intersection (ULLMAN, 1997) between two union compatible relations T_1 and T_2 is the resulting set with the elements that are present in both T_1 and T_2. An example of Intersection is presented in Figure 7.

Figure 7 – Example of the intersection of two relations.

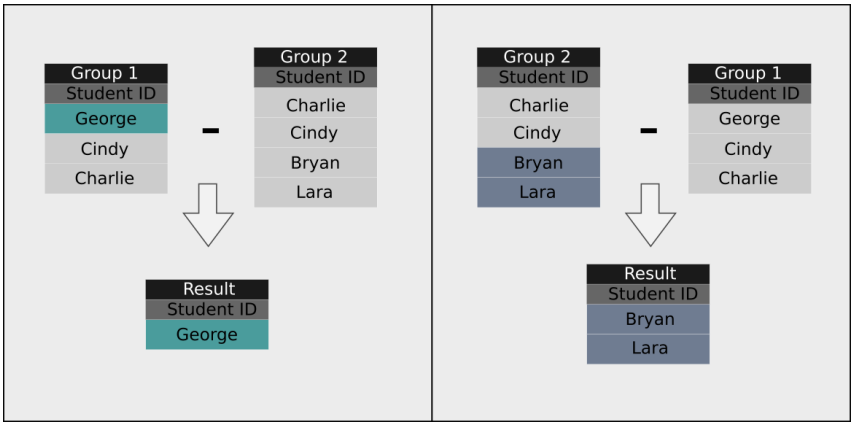


Source: Elaborated by the author.

Difference (−). The difference (ULLMAN, 1997) of two union compatible relations $T_1 - T_2$ is the set of elements that are in T_1 but not in T_2. In addition, this operation is not commutative, that is, $T_1 - T_2$ is not necessarily the same as $T_2 - T_1$, since in the first we obtain the elements that are in T_1 and not in T_2, while in the second, we obtain the elements that are in T_2 but not in T_1. This is shown in Figure 8 with an example where $T_1 = Group1$ and $T_2 = Group2$.

Cartesian Product (×) The Cartesian product (ULLMAN, 1997), also called simply a product, of two relations T_1 and T_2 is the set of pairs that can be formed by choosing any tuple

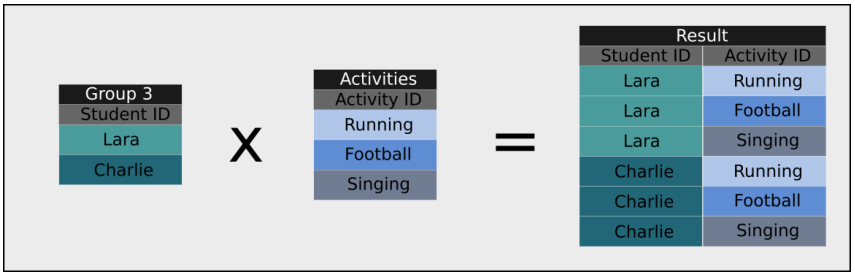
Figure 8 – Example of the differences between two relations.



Source: Elaborated by the author.

from T_1 as the first element and any tuple from T_2 as the second element. Normally this product is denoted with $T_1 \times T_2$. Since the members of T_1 and T_2 are tuples, usually consisting of more than one component, the result of joining a tuple of T_1 with a tuple of T_2 is a larger tuple, with a component for each of the constituent tuples, in addition the components of T_1 precede the components of T_2 in that order. The schema of the resulting relations is the union of the schemas of T_1 and T_2. An example of Cartesian Product is presented in Figure 9.

Figure 9 – Example of the cartesian product of two relations.

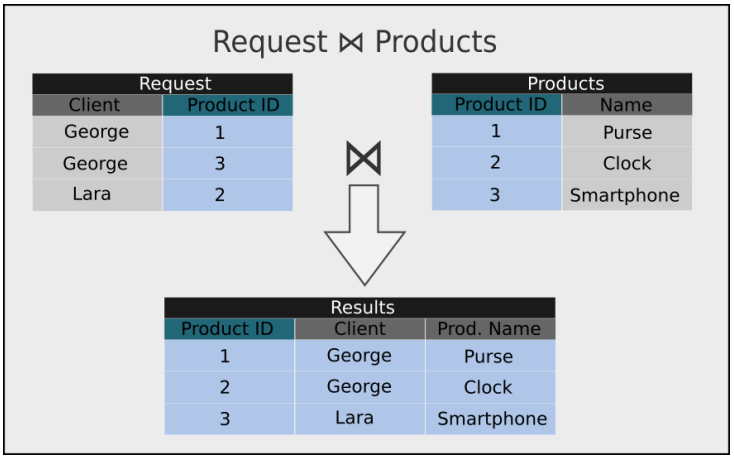


Source: Elaborated by the author.

2.2.2.2 Other binary operations

Natural Join ($\bowtie$) More often than making the Cartesian product of two relations, it is the need to join them (ULLMAN, 1997), joining only those tuples that match in some way. The simplest form of match is the natural join of two relations ($T_1 \bowtie T_2$), in which only those attributes that are common to the schemas (same name and compatible domain) of T_1 and T_2 are matched. More precisely, being $A_1, A_2, ..., A_n$ the common attributes in both schemas of T_1 and T_2, then a tuple of T_1 and a tuple of T_2 are matched if and only if T_1 and T_2 agree on each of the attributes $A_1, A_2, ..., A_n$. If tuples T_1 and T_2 are satisfactorily matched in join $T_1 \bowtie T_2$, then the result of the pairing is a tuple, called joined tuple, with a component for each of the attributes of the schemas of T_1 and T_2. An example of Natural Join is presented in Figure 10.

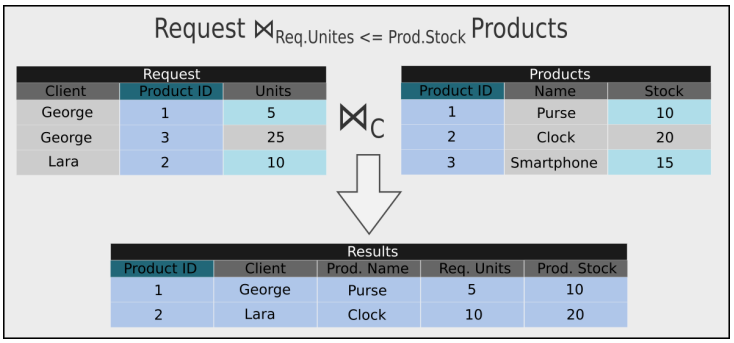
Figure 10 – Example of the natural join of two relations.



Source: Elaborated by the author.

Theta Join ($\bowtie_c$) The natural join forces to match tuples under a specific condition but sometimes it is desired to match two relations under other conditions (ULLMAN, 1997). For this purpose, the theta join allows representing an arbitrary condition, which is denoted by c in the following expression $T_1 \bowtie_c T_2$. The result of this operation is constructed in two steps: first, the product of T_1 and T_2 is taken, then, only the tuples that satisfy condition c are selected from the product. As it happens with the Cartesian product, the relation schema resulting is the union of the schemas of T_1 and T_2. An example of Theta Join is presented in Figure 11.

Figure 11 – Example of a theta join of two relations.



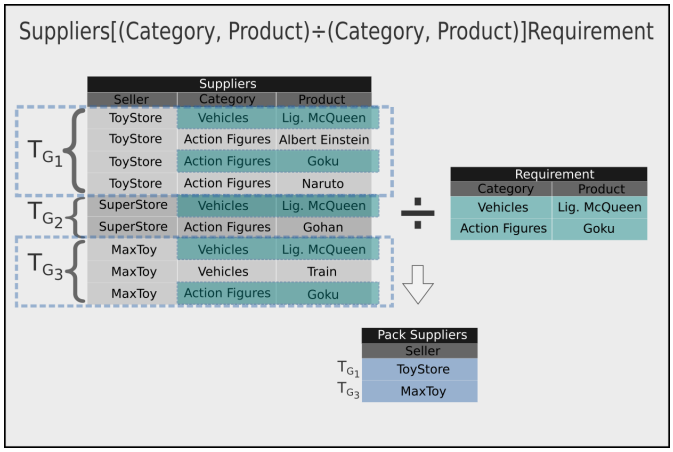
Source: Elaborated by the author.

Relational Division ($\div$) The relational division (CODD, 1972; CODD, 1990) is similar to the division of integer numbers. In both cases there exists one dividend, one divisor, one quotient and even one residual. Although instead of being integers, these will be relations. When a relation is divided by another one, at least one pair of columns (one of the dividend and the other of the divisor) must have the same domain so that each pair of columns can be compared.

Let us consider relation T_1 as the dividend and relation T_2 as the divisor. The columns to be compared will be L_1 (a list of attributes of T_1) and L_2 (a list of attributes of T_2), obtaining

as a quotient T_R, such that $T_R \times \mathsf{T}_2$ is contained in T_1. In addition, L_1 and L_2 must be union-compatible, that is, the number of attributes in $L_1 = (A_1, A_2, ..., A_n)$ must be equal to the number of attributes in $L_2 = (B_1, B_2, ..., B_n)$ and their elements $< A_1, B_1 >, < A_2, B_2 >, ..., < A_n, B_n >$ must be comparable. This operation is defined as $\mathsf{T}_1[L_1 \div L_2]\mathsf{T}_2 = T_R$.

Figure 12 – Example of the relational division of two relations.



Source: Elaborated by the author.

An example of relational division is presented in Figure 12, which aims to solve the question of *Which sellers manage to send all the required products?*, where we have as T_1 to the Suppliers relation and as T_2 to the Requirement relation. Each group T_{G_i} is formed by tuples that have the same Seller, the attributes $[L_1 \div L_2]$ under which the division is being made are $(Category, Product) \div (Category, Product)$ and obtaining as a result the list of suppliers that manage to meet all the requirements. Thus, for this specific case, the equivalent of $\mathsf{T}_1[L_1 \div L_2]\mathsf{T}_2 = T_R$ would be $Suppliers[(Category, Product) \div (Category, Product)]Requirement = PackSuppliers$.

Note 1: When we perform a relational division, we group the dividend's tuples. For example, in 12 we have T_{G_1}, T_{G_2}, and T_{G_3}. If we do a projection in L_1 for each group then we will have three sets. Let's call them S_1, S_2, and S_3. We also can note that if the divisor is included in one set, the corresponding group is part of the quotient. In this case, S_1 and S_3 contain the divisor. Thus, T_{G_1} and T_{G_3} are part of the quotient.

Note 2: If the relational division can be approached based on traditional set operations. Also, if our hypothesis of generalizing the traditional set operations can be expanded to support custom predicates. Then, we could also explore the possibility of generalizing the relational division to support custom predicates. This possibility is explored in Appendix A.

2.3 Infix and Postfix Notations

2.3.1 Basic Concepts

When we work with arithmetic expressions, the form of the expression provides us the information to interpret it correctly (MILLER; RANUM, 2011). For example, if we have 5×2, we know that 5 is being multiplied by 2 since the multiplication operator $\times$ appears between both numbers in the expression. This type of notation is referred to as **infix** since the operator is *in between* the two operands that it is working on.

Now, let's consider the expression $3+5\times 2$. The operators $+$ and $\times$ still appear between the operands, but the expression does not inform which operation should be performed first. The answer is that the multiplication should be solved first. This is due to the precedence level of each operator. The operators with higher precedence must be used before operators of lower precedence. For example, multiplication and division $\{\times, \div\}$ have higher precedence than addition and subtraction $\{+, -\}$. However, if we have an association term in the expression, it needs to be solved first. For example, in the expression $(3+5)\times 2$, we need to solve $3+5$ and its result multiplies it by 2. For another part, in an expression with the same level operators, for example, $3-2-1$, the operations are performed from left to right, i.e, $3-2$ and its result subtracted by 1.

Despite these expressions can be solved intuitively by humans, computers need to know exactly what operators to perform and in what order. One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a fully parenthesized expression. This type of expression uses one pair of parentheses for each operator, so the parentheses dictate the order of operations avoiding ambiguity. In this case, there is no need to know the precedence of the operators. For example, the expression $1+2*3+4$ can be written as $((1+(2*3))+4)$. However, despite the fact that the use of parenthesis avoids ambiguity, the expression might become unnecessarily complex.

For the aforementioned reasons, we will explore other two very important expressions that guarantee the order of the operations. These are the Polish Notation also known as **prefix** notation and the Reverse Polish Notation also known as **postfix** notation. For example, let's consider the infix expression 5×2. What would happen if we moved the operator before the two operands? The resulting expression would be $\times$ 5 2. Likewise, we could move the operator to the end, having 5 2 $\times$. These changes to the position of the operator with respect to the operands create the expressions' formats, "prefix" and "postfix". Prefix notation requires that all operators precede the two operands that they work on. On the other hand, postfix notation requires that its operators come after the corresponding operands. From both notations, the most efficient one is the postfix notation. This is because when we evaluate the expression, every time we find an operator, we know that the last two operands need to be operated with our current operator. This is the opposite with the prefix expression, where every time we found an operator, we need to

look for the next two operands, but these ones could need to be operated with other following operands first. Thus, we will discuss deeply the postfix notation. Let's explore another example now, considering the infix expression $3+5\times2$. In postfix, the expression would be $3\ 5\ 2\times+$. This means that when the first operator $\times$ is read, we need to compute 5 multiplicated by 2, this is 10. The expression will be reduced to $3\ 10\ +$, resulting in 13. Again, note that the order of operations is preserved since the $\times$ appears immediately after 5 and 2, denoting that $\times$ has higher precedence than $+$. We present more examples of these equivalences on Table 1.

Table 1 – Examples of Infix, Prefix and Posfix equivalences.

Infix Expression	Postfix Expression
5×2	$5\ 2\times$
$3+5\times2$	$5\ 2\times3+$
$a+b+c$	$a\ b+c+$
$(a+b)\times c$	$a\ b+c\times$
$(a+b)\times(c+d)$	$a\ b+c\ d+\times$
$a\times(b-c+d)$	$a\ b\ c-d+\times$
$a\div((b-c)+(d\times e))$	$a\ b\ c-d\ e\times+\div$

2.3.2 Conversion of infix to postfix and expression evaluation

As it was described before, infix expressions are the most intuitive way to write an expression for humans. However, this form is very complicated for computers. Instead, postfix expressions are used to solve calculations. The big advantage of this notation is that it is extremely easy and fast, for a computer to analyze (RASTOGI; MONDAL; AGARWAL, 2015). Postfix expression can be easily obtained by push - pop operation on a stack data structure. This greatly simplifies the expression's computation within computer programs.

Let's suppose we have an infix expression I, we can transform it to a postfix expression P by executing Algorithm 1. This algorithm receives a string representing an expression in infix notation and returns a list of tokens in postfix order. First of all, in lines 1 and 2, we define a stack S and a list named Postfix which will store the tokens of the expression in postfix notation. Then, in line 3, we need to push an open bracket "(" onto our stack S and add the close bracket to the end of our infix string. By doing this, we guarantee that the whole expression will be converted to postfix. Next, in line 4 we tokenize our infix string and save the tokens in our Infix list, which now contains all tokens in infix order. Now, in lines 5 to 15, the main loop is executed, reading token by token from the Infix list. If the read token is an operand, the token will be directly added to the postfix notation. If the read token is an open bracket "(", we will add the open bracket to our stack S. If, instead, the token is a close bracket ")", we will pop all operators from our stack S and add them to our Postfix list until an open bracket is found and just removed. If the token is an operator, we will pop from our stack S all operators with the same or higher precedence than

Algorithm 1 – InfixToPostfix(*InfixStr*)

Input: InfixStr: Expression in infix notation
Output: Postfix: List of tokens in postfix notation

```
1: create stack S;
2: create list Postfix;
3: push "(" onto S and add ")" to the end of InfixStr;
4: Infix ← tokenize InfixStr;
5: for each token of Infix do
6:     if token is an operand then
7:         add token to Postfix;
8:     else if token is "(" then
9:         add token to S;
10:    else if token is ")" then
11:        each operator is added to Postfix by popping from S until "(" is read.
12:        remove "(" from S
13:    else if token is an operator then
14:        each operator is added to Postfix by popping from S which has the same or higher
       precedence than the operator encountered;
15:    end if
16: end for
17: return Postfix;
```

Algorithm 2 – EvaluatePostfix(*Postfix*)

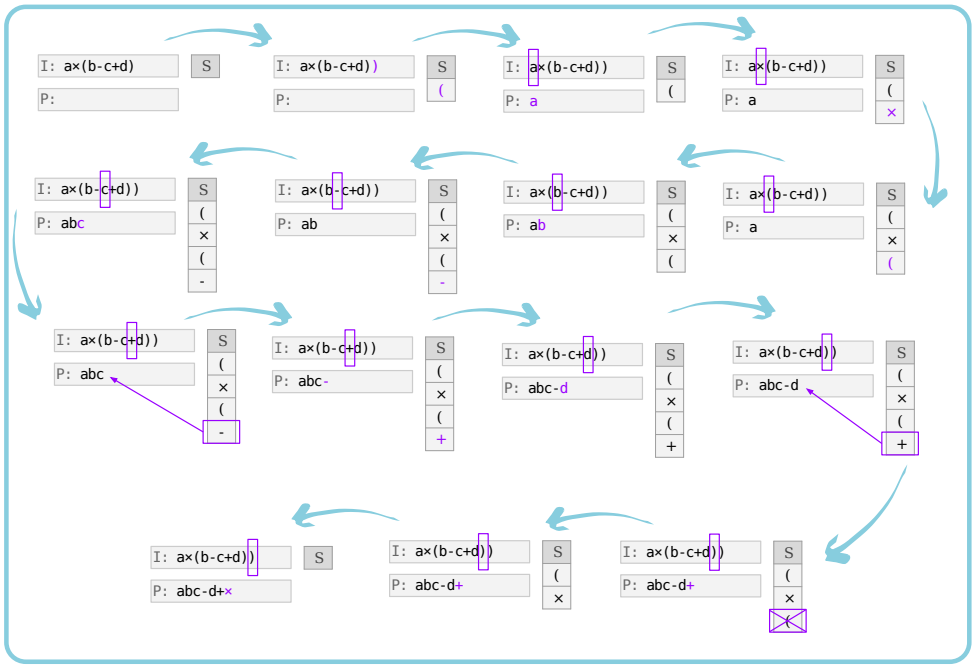
Input: Postfix: Expression in postfix notation
Output: Result: Result of the expression

```
1: Create an empty stack called S;
2: for each token of Postfix do
3:     if token is an operand then
4:         push token into S;
5:     else if token is an operator then
6:         rightOp = pop from S;
7:         leftOp = pop from S;
8:         push into S the result of leftOp operated with righOp;
9:     end if
10: end for
11: Result = pop from S;
12: return Result;
```

the operator encountered and add them to our Postfix list. Finally, we will just return the Postfix list. We can observe an example of this algorithm's execution in Figure 13.

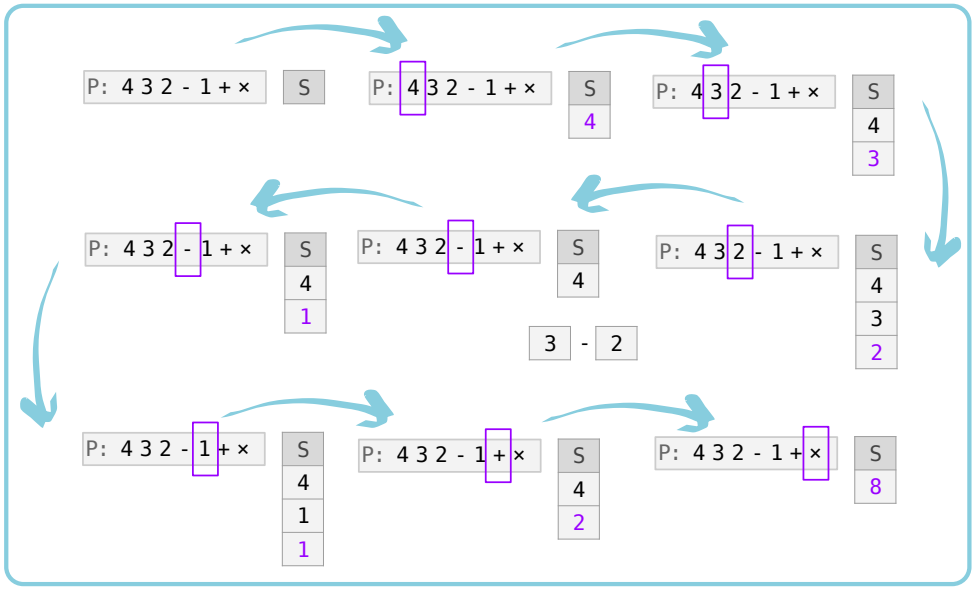
Once we have our expression in postfix notation, our next step would be to evaluate the expression itself. To do this, we can follow steps from Algorithm 2. This algorithm receives a postfix expression and returns the evaluation of the expression. As first step, in line 1, we need to create an empty stack called S. Then, the main loop, in lines 2 to 10, we will read the postfix expression from left to right, token by token. If the token is an operand we need to push its

Figure 13 – Conversion of infix expression to postfix.



Source: Elaborated by the author.

Figure 14 – Evaluating postfix expression.



Source: Elaborated by the author.

value onto S. If instead, the token is an operator, it will need two operands. Pop from S the right operand, then, pop again from S the left operand. Now, evaluate the operation and push the result into S. Once the postfix expression has been completely processed, the result is on the stack. Finally, pop the only value from S and return it. We can observe an example of this algorithm's execution in Figure 14.

2.4 Concluding Remarks

This chapter presented the background concepts used as the basis of this MSc work. We presented the basic concepts and operations of the Theory of Sets such as the relations of set membership and subset, and the operations of Union, Intersection, Difference, Complement, Symmetric Difference, and Cartesian Product. In another section we discussed the Relational Algebra. This is the set of operations that are performed when we work with relations. One important note is that the Relational Algebra employs some of the set operations to work with relations such as the Union, Intersection, Difference, and Cartesian Product. Finally, we also discussed different ways to write expressions, these are infix, prefix, and postfix notations. Here, we mentioned that infix notation is the most intuitive way for humans. However, for a computer to be able to process the expression fast, the best alternative is the postfix notation. We also show how to convert an infix expression to postfix, as well as to evaluate the expression. The next chapter presents related works regarding implementations or extensions of operations from the Relational Algebra.

CHAPTER

3

RELATED WORK

In this chapter, we discuss the main works related to our proposal. This is, works that focus on the traditional Relational Algebra with emphasis on the operators from the Theory of Sets. Then, we discuss the works aimed to extend the operators to support other types of elements, such as complex data types. Naturally, we also emphasize the set operators.

3.1 Works on the Traditional Relational Algebra

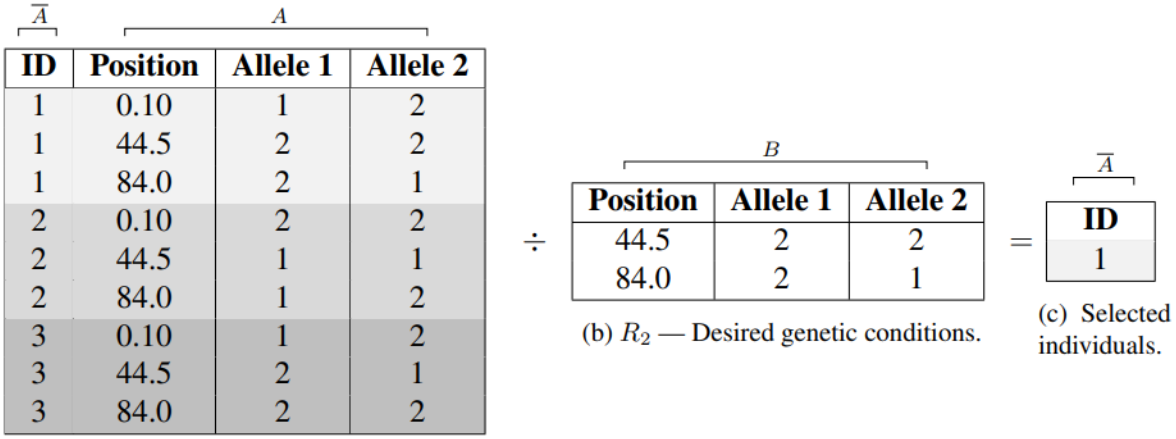
There exist several works that focus on the whole set of operators from the Relational Algebra(KESSLER *et al.*, 2019; VINAYAKUMAR *et al.*, 2018; GORMAN *et al.*, 2014; MCMASTER; SAMBASIVAM; HADFIELD, 2012; MCMASTER *et al.*, 2011; MCMASTER *et al.*, 2010; LITORIYA; RANJAN, 2010; CAO; BADIA, 2007). Since these operators are fundamental for each relational database system, most of these works propose learning tools to teach Relational Algebra through web applications, MS Access, or interactive block-based programming tools. Still, there are some of them that study the Relational Algebra for query optimization (CAO; BADIA, 2007).

Also, there exist works that specifically focus some operators. One case is the join operation and the works that focus on its implementation or optimization (KVET; MATIASKO, 2021; SHIN *et al.*, 2021; SHIN *et al.*, 2020; XUE *et al.*, 2020; ZHOU *et al.*, 2019; NGO *et al.*, 2018). The authors usually do it by implementing one of the three base algorithms for joining, or with variations of them. The first one is the algorithm based on nested loops, which iterate for each row of the left relation and for each row of the right relation, comparing one by one if there is a match in the condition to join the tuples. The second one is the merge join algorithm, which expects to work with two sorted columns or iterating them in order. This way, when the comparing value in one relation is lower than the comparing value in the other one, we know that we must continue with the next tuple on the first side and so on until a tuple identifier is greater than the other one. Then, repeat the same process until one column ends. In the middle of

the process, if two comparing values are identical, both tuples are joined. The third algorithm is the hash join. This algorithm receives this name because it uses hash tables. It also requires that at least one term in the predicate is formed by two columns compared by identity (one equi-join). As the first step, for the left relation column values, the algorithms call a hash function and calculate the keys for all values and store them in buckets. Then, for each of the right relation column values, the key is calculated and then it is searched in the bucket if there exists a value identical to the current one. If it does, then both tuples are joined as part of the result.

There are also several works focused on implementing and expanding the relational division. Since most implementations in SQL do not directly implement the division as in the case of the join operator (ELMASRI; NAVATHE, 2015), some authors implement the relational division as queries in SQL, stored procedures or even with external codes using an API (IMAMUDDIN; NAHAR; CHANDRA, 2020; GONZAGA; CORDEIRO, 2016; CAMPS, 2014; MATOS; GRASSER, 2002; CELKO, 2009). One of these works (GONZAGA; CORDEIRO, 2016) compares the performance of the best implementations for the division operator in SQL, as well as proposes a new algorithm for the division, which is implemented through stored procedures. This work presents a case study on the selection of individuals that have certain genetic characteristics. Given the genetic data of several individuals and the desired genetic conditions, all those individuals that satisfy all conditions are obtained in response. Figure 15 exemplify the use case. Finally, the results indicate that the proposed implementation is potentially faster than the best implementation in SQL.

Figure 15 – Example of the relational division used to select individuals that satisfy desired genetic conditions.



Source: Gonzaga and Cordeiro (2016).

Note 1: Let's observe in Figure 15 the tuples' groups (groups of ID = 1, groups of ID = 2, and groups of ID = 3). Then, let's do a projection of only the desired attributes (Position, Allele 1, and Allele 2) for each group. Let's call these projections as sets S_1, S_2, and S_3. Now, if each set in the dividend contains as a subset the divisor, we can say that the group satisfies all

requirements and its ID is part of the quotient.

Note 2: If the relational division can be approached based on traditional set operations. Besides, if our hypothesis of generalizing the traditional set operations can be expanded to support custom predicates. Then, we could also explore the possibility of generalizing the relational division to support custom predicates. This possibility is explored in Appendix A.

Continuing with this work, the authors proposed Algorithm 3, which assumes that there are index structures for the dividend attributes. Using index structures makes this algorithm to be way faster than if it doesn't. This approach starts with a set T_R containing the $\overline{L_1}$ attributes of all possible tuples, assuming that at the beginning all divisor tuples meet the requirements. Then, for each tuple t_j of T_2 a set T is selected that contains the selection of the tuples of T_1 that match in $\mathsf{L}_1 = \mathsf{L}_2$. Finally, the set of elements of T_R is updated for its intersection with all the selected elements of T.

Algorithm 3 – Index_RelationalDivision(T_1, L_1, T_2, L_2)

1: $T_R = \pi_{(\overline{L_1})}(T_1)$
2: **for** each tuple $t_j \in \pi_{(\mathsf{L}_2)}\mathsf{T}_2$ **do**
3: $\quad T = \pi_{(\overline{L_1})}\left(\sigma_{(L_1 = t_j)}T_1\right)$
4: $\quad$ **for** each tuple $t \in \mathsf{T}$ **do**
5: $\quad\quad T_R = T_R \cap t$
6: $\quad$ **end for**
7: **end for**
8: **return** T_R

3.1.1 Works on Relational Set Operators

We previously discussed the works that focus generally on the Relational Algebra (KESSLER *et al.*, 2019; VINAYAKUMAR *et al.*, 2018; GORMAN *et al.*, 2014; MCMASTER; SAMBASIVAM; HADFIELD, 2012; MCMASTER *et al.*, 2011; MCMASTER *et al.*, 2010; LITORIYA; RANJAN, 2010; CAO; BADIA, 2007). Naturally, these works include the relational set operations. However, there are also some works that study specifically the traditional set operations from the Relational Algebra (RED'KO *et al.*, 2017; BILLE *et al.*, 2007). These works generally study the behavior of the set operators in real-world applications or attempt to optimize the queries for the diverse set operators. This is the case of a recent work (RED'KO *et al.*, 2017) that compares different algorithms for the intersection, union, and difference operations. The main goal of this work was to optimize queries for set operations, proposing nested loops algorithms from their most naive form and adding some changes to find the fastest algorithms for each operation.

Starting with the intersection of two relations T_1 and T_2, the authors presented the first algorithm, which consisted of comparing all rows from table T_2 with each row $t_1 \in \mathsf{T}_1$. If there was a row $t_2 \in \mathsf{T}_2$ such that $t_1 = t_2$, then the row t_1 was added to the resulting table T_R and

continue to the next row of the table T_1. Then, they added a feature and create the second algorithm. Here, when the row $t_1 = t_2$ is found, they still add t_1 to result but also delete t_2 from T_2. The third variation doesn't return T_R as result anymore, instead, it will be T_1 which will be modified to contain only the result of the intersection. Here, we need to work as left relation with the relation of lower size. Let's assume it is T_1. Here, we remove all tuples t_1 from T_1 that doesn't match any t_2 from T_2. Finally, the fourth and best algorithm proposed by the authors for the intersection was Algorithm 4. In this last algorithm, we can see in lines 1-3 that if the relation T_1 has a greater size than T_2, it returns the intersection of both relations transposed. If not, then in lines 4 to 13, it iterates for each tuple t_1 in the left relation, and inside, for each tuple t_2 in the right relation. When $t_1 = t_2$ then t_2 is removed from T_2. Otherwise, t_1 is removed from T_1 and the inner loop is leaved to continue with the next tuple in T_1.

Algorithm 4 – Intersection($\mathsf{T}_1, \mathsf{T}_2$)

Input: T_1: Left Relation, T_2: Right Relation
Output: The intersection of both relations.

```
1:  if |T1| > |T2| then
2:      return Intersection(T2, T1)
3:  end if
4:  for each t1 ∈ T1 do
5:      for each t2 ∈ T2 do
6:          if t1 = t2 then
7:              remove t2 from T2
8:          else
9:              remove t1 from T1
10:             break
11:         end if
12:     end for
13: end for
14: return T1
```

Following with the union of two relations T_1 and T_2, the authors presented the first algorithm, which consisted of adding all rows from T_1 to the result T_R. Then, comparing all rows from table T_2 with each row $t_1 \in \mathsf{T}_1$. If there was not a row $t_2 \in \mathsf{T}_2$ such that $t_1 = t_2$, then the row t_2 was added to the resulting table T_R and continue to the next row of the table T_1. In the next variation, they changed a feature and create the second algorithm. Here, instead of adding the row t_2 from T_2 when there was no equal row in T_1, they remove the each row t_2 from T_2 if a tuple t_1 equal to it exists in T_1. Finally, the third and best algorithm proposed by the authors for the union was Algorithm 5. In this last algorithm, we can see in lines 1-3 that if the relation T_1 has a greater size than T_2, it returns the union of both relations transposed. If not, then in lines 4 to 10, it iterates for each tuple t_1 in the left relation, and inside, for each tuple t_2 in the right relation. When $t_1 = t_2$ then t_1 is removed from T_2 and we can continue with the next tuple in T_2. Finally, in lines 12 and 13, all remaining tuples from T_1 are added to the result in T_2 and this one is returned.

Algorithm 5 – Union($\mathsf{T}_1, \mathsf{T}_2$)

Input: T_1: Left Relation, T_2: Right Relation
Output: The union of both relations.

```
1:  if |T1| > |T2| then
2:      return Union(T2, T1)
3:  end if
4:  for each t1 ∈ T1 do
5:      for each t2 ∈ T2 do
6:          if t1 = t2 then
7:              remove t1 from T1
8:              continue with next tuple of T2
9:          end if
10:     end for
11: end for
12: Add remaining tuples from T1 to T2
13: return T2
```

Finally, for the difference of two relations T_1 and T_2, the authors studied five algorithms. The first one consisted of comparing all rows from table T_2 with each row $t_1 \in \mathsf{T}_1$. If there was not a row $t_2 \in \mathsf{T}_2$ such that $t_1 = t_2$, then the row t_1 was added to the resulting table T_R and continue to the next row of the table T_1. As a variation, they added a feature and create the second algorithm. Here, the tuple t_1 is still added to the resulting relation T_R when there was no equal row t_2 to it in T_1, but also, if there existed that tuple, then the tuple t_2 from T_2 was removed from T_2. In the next variation, the result is only being updated in the same table T_1. Here, all tuples from both relations are compared, and if for a tuple $t_1 \in \mathsf{T}_1$ exists an identical tuple $t_2 \in \mathsf{T}_1$, the tuple t_1 is removed from the left relation. In the next variation, they remove not only t_1 from left relation but also t_2 from the right relation when t_1 and t_2 are identical. Finally, the fifth and best algorithm proposed by the authors for the difference was Algorithm 6. In this last algorithm, we can see in lines 1-3 that if the relation T_1 has a greater size than T_2, it returns the union of both relations transposed. If not, then in lines 4 to 11, it iterates for each tuple t_1 in the left relation, and inside, for each tuple t_2 in the right relation. When $t_1 = t_2$ then t_1 is removed from T_2 and we can continue with the next tuple in T_2. Finally, in line 13, the tuples remaining in T_1 compose the result.

Summarizing, the authors proposed four algorithms for intersection, three algorithms for union and five algorithms for difference. To find the best algorithms for each set operation, worst-case and average-case complexity values were calculated for each algorithm. As results, for the intersection operation, the first algorithm has the smallest complexity in the worst case, but the last algorithm had the smallest average-case complexity. For the union operation, the third algorithm had the smallest worst-case and average-case complexity. Among the difference algorithms, the third and fifth algorithms had the smallest worst-case complexity, but only the fifth algorithm has the smallest average-case complexity. Thus, among the proposed algorithms

Algorithm 6 – Difference(T_1, T_2)

Input: T_1: Left Relation, T_2: Right Relation
Output: The difference of both relations.

```
1: if |T1| > |T2| then
2:     return Difference(T2, T1)
3: end if
4: for each t1 ∈ T1 do
5:     for each t2 ∈ T2 do
6:         if t1 = t2 then
7:             remove t1 from T1
8:             continue with next tuple of T2
9:         end if
10:    end for
11: end for
12: return T1
```

by the authors, Algorithm 4, Algorithm 5, and Algorithm 6 are the fastest. Also, the authors experimentally demonstrated their theoretical results by developing a software system that computed the actual number of executed actions for each of the proposed algorithms and comparing them with the obtained theoretical estimates, differing from design values by no more than 0.1%.

3.2 Extensions of Relational Algebra Operators

Following another approach, many modern applications require not only to store numeric and short character strings but also videos, photos, large text, and several other types of "complex" data elements. Here, comparing complex elements by identity, in contrast to the traditional simple elements, is usually senseless because an exact match almost never happens(POLA *et al.*, 2015; POLA *et al.*, 2013). For example, let's imagine we are working with images, and we take two pictures of the same object, it will be almost impossible for both images to be identical. Instead, it is more significant to evaluate their similarity. This is why there exist several works that focus on extending the Relational Algebra to support queries with complex data.

To support similarity-aware comparisons, it is common to represent the dataset in a metric space. A metric space $< \mathbb{S}, d >$ is a combination of a data domain $\mathbb{S}$ and a distance function $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$ that meets the following properties, for any s_1, s_2 and s_3 in $\mathbb{S}$.

- Identity: $d(s_1, s_2) = 0 \iff s_1 = s_2$
- Non-Negativity: $0 < d(s_1, s_2) < \infty$; $\forall s_1 \neq s_2$
- Symmetry: $d(s_1, s_2) = d(s_2, s_1)$
- Triangle inequality: $d(s_1, s_2) < d(s_1, s_3) + d(s_3, s_2)$

Another fact distinguishes complex data even more from traditional data: ordering properties does not hold among complex data elements. One can say only that two elements are equal or different, as in a general case there is no rule that allows sorting the elements. As a consequence, the relational operators $<$, $\leq$, $>$, and $\geq$ cannot be used for comparisons. Moreover, since exact match rarely occurs or makes sense here, the identity-based operators $=$ and $\neq$ are also almost useless. Therefore, only similarity-based operations can be performed over complex data. The main similarity-based comparison operations are the similarity range and k-nearest neighbor. Both operations receive a query center s_q to perform the selection. The similarity range also receives a threshold ξ and retrieves the elements $s_i \in S$, such that $d(s_i, s_q) \leq \xi$. Likewise, the k-nearest neighbor selection receives the parameter k and retrieves the k elements in S that are nearest to s_q.

There are many works that study the similarity-aware Selection (LU *et al.*, 2018; LU *et al.*, 2017; SILVA *et al.*, 2013; SILVA *et al.*, 2010; BARIONI *et al.*, 2009; SANTOS *et al.*, 2013; BUDíKOVá; BATKO; ZEZULA, 2012; BELOHLAVEK; VYCHODIL, 2010), in which similarity awareness is achieved by using range queries, nearest neighbor queries, and their variants.

Other proposals explore the similarity-aware Join (RONG *et al.*, 2017; YU *et al.*, 2016; SILVA *et al.*, 2015; DENG *et al.*, 2015; JIANG *et al.*, 2014; SILVA *et al.*, 2013; KALASHNIKOV, 2013; SILVA; PEARSON, 2012; SILVA; AREF; ALI, 2010; JACOX; SAMET, 2008) implementing range queries and nearest neighbor queries as well. The main approaches are: (a) Range Distance Join that retrieves every pair of tuples with distance lower or equal than the threshold; (b) kNN Join, which for each tuple on the left relation, retrieves the k most similar tuples from the right one, and; (c) k-Distance Join that retrieves the overall k most similar pairs of tuples.

Other contributions we can mention are the extensions of the relational division to work with complex data, proposing the definitions, algorithms and utility of the similarity-aware division (GONZAGA; CORDEIRO, 2019; GONZAGA; CORDEIRO, 2017) and also the inclusion of these algorithms to the database management systems (VASCONCELOS; KASTER; CORDEIRO, 2018; VASCONCELOS *et al.*, 2018).

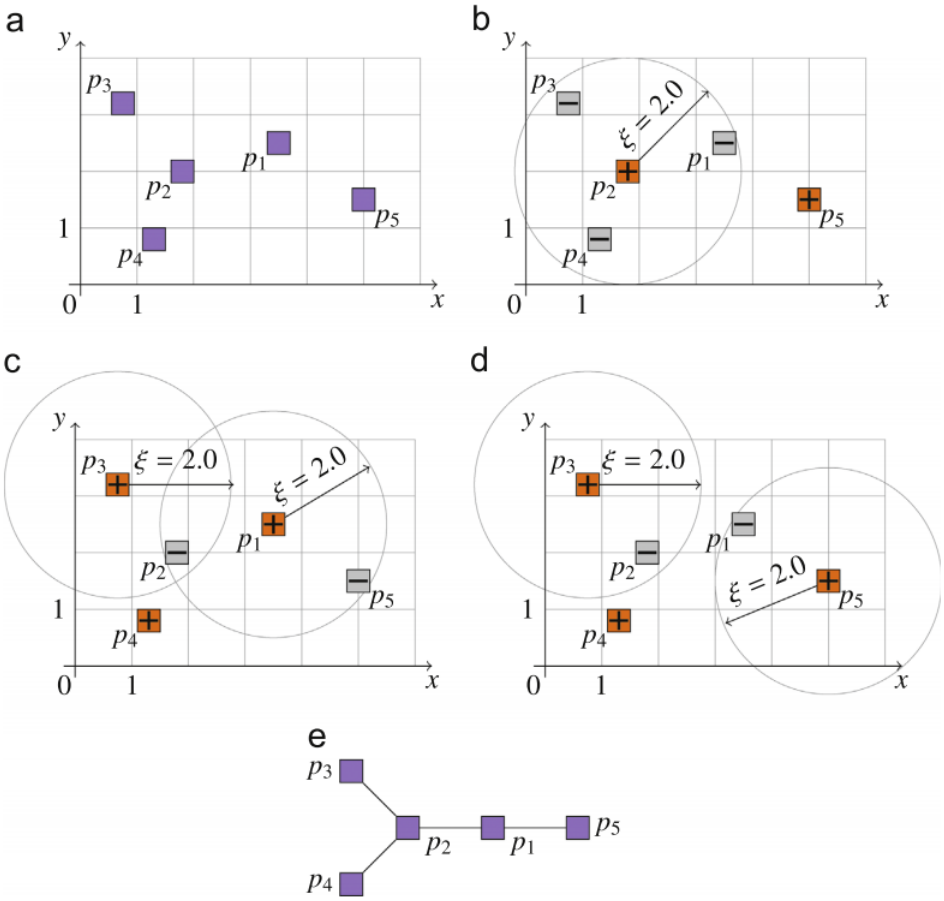
Also, there exists another branch of research that discusses the Relational Algebra operations with an approach in fuzzy logic (GALINDO; URRUTIA; PIATTINI, 2005). Following this approach, many works (BOSC; PIVERT, 2013; ZHAO *et al.*, 2007; TANG; CHEN, 2004; SHARMA *et al.*, 2004) attempt to extend the Relational Algebra semantics to a fuzzy domain, which implies having fuzzy relations with vague values using linguistic labels, weighted tuples, and grades with different meanings for the attributes. However, none of them is concerned in extending the relational set operations for conditional sets.

3.2.1 Extensions of Relational Set Operators

We have previously discussed the works that extend different operators from the Relational Algebra, such as selection, join and division. Now, let's explore the related works aimed to extend the databases set operators.

There are many works that focus on adapting the set operators to work with complex data (Al Marri *et al.*, 2016; POLA *et al.*, 2015; MARRI *et al.*, 2014; POLA *et al.*, 2013). Remember the traditional concept tell us that a set cannot include the same element twice (there can't be two identical elements in the same set). Now, let's imagine we are working with complex data. As exact match on pairs of complex elements seldom occurs or makes sense, the usual concept of "set" also blurs for these data. Instead, the concept equivalent to "sets" for complex data: the "similarity sets" or just *SimSets* can be defined as a set of complex elements without any two elements "too similar" to each other (POLA *et al.*, 2015; POLA *et al.*, 2013).

Figure 16 – (a) Example of a 2-dimensional dataset; (b)–(d) examples of 2-simsets produced from our toy dataset using the Euclidean distance; (e) corresponding ξ-similarity graph for $\xi = 2$.



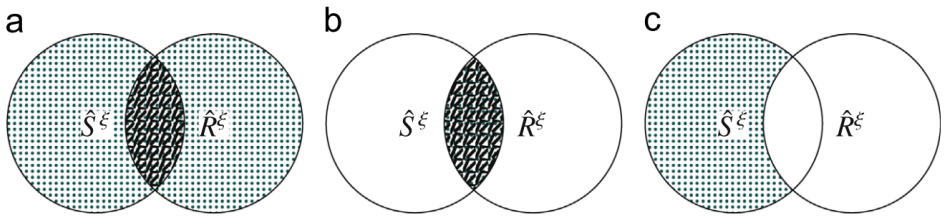
Source: Pola *et al.* (2015).

Two fundamental concepts defined by the authors are the similarity-set and the similarity graph. As mentioned before, a similarity-set is a concept aimed to work with sets of complex objects, where we want to ensure that two "too similar" elements do not be in the same set. So in

the same way that we use distance functions to measure similarity, where larger values represent less similar items; we call as SimSet to a set where there are no elements that are too similar. Formally, A ξ-similarity-set $\hat{S}^{\xi}$ is a set of elements from a metric domain $<\mathbb{S}, d>, \hat{S}^{\xi} \subset \mathbb{S}$, such that there is no pair of elements $s_i, s_j \in \hat{S}^{\xi}$ where $s_i \hat{=}^{\xi} s_j$. Two elements s_1, s_2 are ξ-similar if $d(s_1, s_2) \leq \xi$. It is important to note that there may exist many distinct ξ-simsets within a set S. For example, Figure 16(a) illustrates a set of points associated with distance $\xi = 2$ which produces many distinct 2-simsets. Examples are $\{p_2, p_5\}$ from Figure 16(b), $\{p_1, p_3, p_4\}$ from Figure 16(c) and $\{p_3, p_4, p_5\}$ from Figure 16(d), since the distance between p_2 and p_5 is greater than 2, and it also happens for p_1, p_3 and p_4, and for p_3, p_4 and p_5. The other fundamental concept is the corresponding ξ-similarity graph for of a set: $\hat{G}^{\xi}(S) = \{V, E\}$, where each node $v_i \in V$ corresponds to an element $s_i \in S$, and there is an edge $< v_i, v_j > \in E$ if and only if $s_i \hat{=}^{\xi} s_j$. Figure 16(e) illustrates the 2-similarity graph of Figure 16(a).

As traditional union, intersection and difference operations are not well suited to work with complex data, these concepts were also adapted to handle these operations by similarity. The ξ-similarity Union of two ξ-simsets $\hat{S}^{\xi}$ and $\hat{R}^{\xi}$ is defined as the set $\hat{T}^{\xi}$ of all elements that are either in $\hat{S}^{\xi}$ or in $\hat{R}^{\xi}$, and do not exist any two elements $t_i, t_j \in \hat{T}^{\xi}$ such that $t_i \hat{=}^{\xi} t_j$. The similarity union is illustrated in Figure 17(a). The ξ-similarity Intersection of two ξ-simsets $\hat{S}^{\xi}$ and $\hat{R}^{\xi}$ is defined as the set $\hat{T}^{\xi}$ of all elements in the traditional union $\hat{S}^{\xi} \cup \hat{R}^{\xi}$ such that $t_i \in \hat{S}^{\xi} \wedge \exists r_j \in \hat{R}^{\xi} | t_1 \hat{=}^{\xi} r_j$ or $t_i \in \hat{R}^{\xi} \wedge \exists s_k \in \hat{S}^{\xi} | t_i \hat{=}^{\xi} s_k$. The similarity intersection is illustrated in Figure 17(b). The ξ-similarity Difference of two ξ-simsets $\hat{S}^{\xi}$ and $\hat{R}^{\xi}$ is the set of the elements $s_i \in \hat{S}^{\xi}$ such that there is no element $r_i \in \hat{R}^{\xi}$ such that $s_i \hat{=}^{\xi} r_j$. The similarity difference is illustrated in Figure 17(c).

Figure 17 – Representation of similarity binary operations: (a) similarity union, (b) similarity intersection, (c) similarity difference.



Source: Pola *et al.* (2015).

Naturally, the authors also contributed with the algorithms to support their new concepts for selecting similarity sets from a set (a distinct algorithm), and one single algorithm called "BinOp" that supports querying the three binary operations (similarity union, similarity intersection, and similarity difference). Finally, they validated their proposal by performing experiments with synthetic and real data, proving that the proposed algorithms are scalable and accurate.

We've studied different works that include the Relational Algebra operators from the Theory of Sets. Some of those works even extended the operations to answer more complex

queries. However, none of them allows answering "conditional" queries with custom predicates.

3.3 Concluding Remarks

This chapter presented related works in traditional Relational Algebra operators and also on extensions of them, with more emphasis in works that study the relational operators from the Theory of Sets. In the first section, we discussed works that focus generally on Relational Algebra, but also in some specific operands such as joins and division. In the second section, we studied the extensions of these operations to support complex data, defining first the new type of set, in this case the similarity set, and extending the binary operations of union, intersection and difference. Although all these works contribute with our research, as we may see in Table 2, none of them allows answering "conditional" queries with custom predicates. Thus, we present our proposal in the next chapter.

Table 2 – Summary of the State of the Art in Relational Algebra Set Operations

	Present Scalable Algorithms	Support Identity Predicates	Support Similarity-aware Predicates	Support Custom Predicates
Al Marri *et al.* (2016)	✓	✓	✓	
Marri *et al.* (2014)	✓	✓	✓	
Pola *et al.* (2013)	✓	✓	✓	
Red'ko *et al.* (2017)	✓	✓		
Bille *et al.* (2007)	✓	✓		
Our Contribution	✓	✓		✓

CHAPTER

4

RELATIONAL CONDITIONAL SET OPERATIONS

This chapter presents the main contribution of this MSc work: the Relational Conditional Set Operations. First of all, we present here some previous definitions to then present the formal definitions for all of our operators.

4.1 Formal Definitions

Definition 1. An **arithmetic operator between values** ($\otimes$) is represented by $a_1 \otimes a_2$, in which $a_1 \in \mathbb{R}$ and $a_2 \in \mathbb{R}$, and the result $a_1 \otimes a_2 \in \mathbb{R}$. By enumeration:

$$\otimes \in \{+, -, *, /\} \tag{4.1}$$

Definition 2. A **logical operator between attribute values** ($\odot$) is represented by $a_1 \odot a_2$, in which $a_1 \in A_1$ and $a_2 \in A_2$ are attribute values and the result is either $true$ or $false$. A_1 and A_2 are attributes that must have the same domain, $Dom(A_1) = Dom(A_2)$. By enumeration, the operator is expressed as:

$$\odot \in \{<, \leq, >, \geq, =, \neq\} \tag{4.2}$$

The logical operator $\odot$ may not be commutative. Thus, $a_1 \odot a_2$ and $a_2 \odot a_1$ do not necessarily produce the same logic result.

Definition 3. The **logical negation** ($\neg$) is represented as $\neg b$. It reverses the state of truth of one boolean value b. That is:

$$\neg b = \begin{cases} true, & \text{if } b \text{ is } false \\ false, & \text{if } b \text{ is } true \end{cases} \tag{4.3}$$

Definition 4. A **logical connector between boolean values** ($\diamond$) is represented by $b_1 \diamond b_2$, in which b_1 and b_2 are logical states of $true$ or $false$, and the result of the connection can only be $true$ or $false$ as well. By enumeration:

$$\diamond \in \{\wedge, \vee\} \tag{4.4}$$

The logical connector $\diamond$ is commutative. Therefore, $b_1 \diamond b_2 = b_2 \diamond b_1$.

Definition 5. A **predicate between tuples** (c) is represented by $c(t_1, t_2)$, in which t_1 and t_2 are tuples from relations T_1 and T_2, respectively. Relations T_1 and T_2 must be union compatible. The predicate is a logical expression that can group different operands with logical connectors, negations, and parentheses; its result is always $true$ or $false$. Formally, we have:

$$c(t_1, t_2) = \begin{cases} t_1.a_1 \, [\otimes p] \odot t_2.a_2 \, [\otimes q] \\ (c(t_1, t_2) \diamond c(t_1, t_2)) \\ \neg c(t_1, t_2) \end{cases} \tag{4.5}$$

Here, [ ] indicate optional operands, p and q are arithmetic expressions involving only constant values, and $t.a$ is an attribute value of tuple t. $c(t_1, t_2)$ may not be commutative. Thus, $c(t_1, t_2)$ and $c(t_2, t_1)$ are not necessarily equivalent.

Definition 6. A **relational conditional set** ($_c\mathsf{T}$) is a relation T in which all of its tuples are conditionally different. That is, given a predicate c, there are no pair of tuples in T that satisfies the predicate. Formally, we have:

$$\mathsf{T} \text{ is } {}_c\mathsf{T} \Leftrightarrow \nexists t_i \in \mathsf{T}, t_j \in \mathsf{T} : \; c(t_i, t_j) \text{ is } true \; \wedge \; t_i \neq t_j \tag{4.6}$$

Definition 7. Two relational conditional sets $_c\mathsf{T}_1$ and $_c\mathsf{T}_2$ are **union compatible** if they both have the same number of attributes and each attribute from $_c\mathsf{T}_1$ has the same domain of its counterpart in $_c\mathsf{T}_2$. Let us consider A_i to be the i^{th} attribute in the schema $Sch(_c\mathsf{T})$ of a set $_c\mathsf{T}$. The domain of A_i is $Dom(A_i)$. In this setting, $_c\mathsf{T}_1$ and $_c\mathsf{T}_2$ are union compatible if and only if:

$$\begin{aligned} &(|Sch(_c\mathsf{T}_1)| = |Sch(_c\mathsf{T}_2)|) \wedge \\ &(\forall A_i \in Sch(_c\mathsf{T}_1), \forall A_j \in Sch(_c\mathsf{T}_2), i = j : Dom(A_i) = Dom(A_j)) \end{aligned} \tag{4.7}$$

Definition 8. The **relational conditional set membership** ($\in_c$) is represented as $t \in_c {}_c\mathsf{T}$, in which ${}_c\mathsf{T}$ is a relational conditional set and t is a tuple from any relation T. Relations ${}_c\mathsf{T}$ and T must be union compatible. Tuple t is a conditional element of ${}_c\mathsf{T}$ if and only if there exists one tuple $t_j \in {}_c\mathsf{T}$ that satisfies the predicate $c(t,t_j)$. Formally, we have:

$$t \in_c {}_c\mathsf{T} \Leftrightarrow \exists t_j \in {}_c\mathsf{T} : c(t,t_j) \tag{4.8}$$

Following the same idea, t is not a conditional element of ${}_c\mathsf{T}$ if and only if there is no tuple $t_j \in {}_c\mathsf{T}$ that satisfies predicate $c(t,t_j)$. Formally, it is given by:

$$t \notin_c {}_c\mathsf{T} \Leftrightarrow \nexists t_j \in {}_c\mathsf{T} : c(t,t_j) \tag{4.9}$$

Definition 9. The **relational conditional subset** ($\subseteq_c$) is given by ${}_c\mathsf{T}_1 \subseteq_c {}_c\mathsf{T}_2$, where ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$ are relational conditional sets, and the result is either $true$ or $false$. Relations ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$ must be union compatible. ${}_c\mathsf{T}_1$ is a conditional subset of ${}_c\mathsf{T}_2$ if and only if every tuple $t_i \in {}_c\mathsf{T}_1$ is also a conditional element of ${}_c\mathsf{T}_2$. Formally, its is defined by:

$${}_c\mathsf{T}_1 \subseteq_c {}_c\mathsf{T}_2 \Leftrightarrow \forall t_i \in {}_c\mathsf{T}_1 : t_i \in_c {}_c\mathsf{T}_2 \tag{4.10}$$

The conditional subset operation may not be commutative, so if ${}_c\mathsf{T}_1 \subseteq_c {}_c\mathsf{T}_2$ is valid we cannot affirm that ${}_c\mathsf{T}_2 \subseteq_c {}_c\mathsf{T}_1$ is also valid. However, one interesting property is that ${}_c\mathsf{T}_1 \subseteq_c {}_c\mathsf{T}_2$ can be valid even when the cardinality of ${}_c\mathsf{T}_1$ is larger than that of ${}_c\mathsf{T}_2$, since a single tuple of ${}_c\mathsf{T}_2$ can satisfy many tuples of ${}_c\mathsf{T}_1$.

Definition 10. The **relational conditional intersection operation** ($\cap_c$) is a binary operation represented as ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2 = {}_c\mathsf{T}_R$, in which ${}_c\mathsf{T}_R$ has the result of the conditional intersection between ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$. Relations ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$ must be union compatible. The resulting relation ${}_c\mathsf{T}_R$ has all tuples of ${}_c\mathsf{T}_1$ that are also conditional members of ${}_c\mathsf{T}_2$. Formally, we have:

$${}_c\mathsf{T}_R = \{t_i \,:\, t_i \in {}_c\mathsf{T}_1 \wedge t_i \in_c {}_c\mathsf{T}_2\} \tag{4.11}$$

The relational conditional intersection may not be commutative. Thus, queries ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$ and ${}_c\mathsf{T}_2 \cap_c {}_c\mathsf{T}_1$ do not necessarily produce the same results.

Definition 11. The **relational conditional difference operation** ($-_c$) is a binary operation represented as ${}_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2 = {}_c\mathsf{T}_R$, in which ${}_c\mathsf{T}_R$ has the result of the conditional difference between ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$. Relations ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$ must be union compatible. The resulting relation ${}_c\mathsf{T}_R$ has all tuples of ${}_c\mathsf{T}_1$ that are not conditional members of ${}_c\mathsf{T}_2$. Formally, we have:

$${}_c\mathsf{T}_R = \{t_i \,:\, t_i \in {}_c\mathsf{T}_1 \wedge t_i \notin_c {}_c\mathsf{T}_2\} \tag{4.12}$$

The relational conditional difference may not be commutative. Thus, queries ${}_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2$ and ${}_c\mathsf{T}_2 -_c {}_c\mathsf{T}_1$ do not necessarily produce the same results.

4.2 Conclusion

This chapter presented the formal definitions for our Relational Conditional Set Operations. For this, we first presented some base definitions: the arithmetic operator between tuples ($\otimes$), the logical operator between attribute values ($\odot$), the logical negation ($\neg$), the logical connector between boolean values ($\diamond$), the predicate between tuples (c), the relational conditional set (${}_c\mathsf{T}$), and the union compatibility between two conditional sets. Subsequently, we formally define our four relational conditional set operations: the relational conditional set membership ($\in_c$), the relational conditional subset ($\subseteq_c$), the relational conditional intersection ($\cap_c$), and the relational conditional difference ($-_c$).

CHAPTER

5

ALGORITHMS FOR THE RELCOND SET OPERATIONS

In order to allow the execution of our conditional set operations, we developed algorithms to support them. Here, the following considerations must be made.

Relations: The algorithms read both relations from disk. One relation is iterated in a full table scan; the other is read through indexes for each column. In our motivational examples, the right relation is the largest one, i.e., *SP*. However, in general, one of the two relations must contain index structures.

Arithmetic Expressions: We assume that all expressions in the predicate refer to the table without indexes. Note that it is always possible, e.g., instead of writing $DP.Price \geq SP.Price * 2$ one may write $DP.Price/2 \geq SP.Price$.

5.1 Proposed Algorithms

Let us now focus on the algorithms. We present in this section the Algorithm 7, which converts an expression in infix notation to one in postfix notation. This algorithm is important in order to process efficiently evaluate the expression later. Starting the algorithm, in Lines 1-2, we need to create a stack of pending operations and a vector to contain the postfix expression to be returned. Also, in the initialization, Line 3 pushes an open bracket "(" onto the stack of pending operations, as well as Line 4, adds the closing bracket ")" to the end of our input expression. Lines 5-22 are the main loop, which will get every single token extracted from the original expression in infix notation ($infixExp$). The current token being read will be processed according to its type. If the token is an operand (any constant or reference to a column table), it will be added directly to the postfix expression vector. In case the token is an opening bracket "(", it will be added to the stack of pending operations. But if the token is a closing bracket ")", we will pop tokens from the pending operations and add them to the postfix expression until

Algorithm 7 – ToPostfix(*infixExp*)

Input: *infixExp*: Expression in infix notation
Output: *postfixExp*: Expression in postfix notation

```
1: create stack pendingOps;
2: create vector postfixExp;
3: push "(" onto pendingOps;
4: add ")" to the end of infixExp;
5: for each token extracted from infixExp do
6:     if token is an operand then
7:         add token to postfixExp;
8:     else if token is "(" then
9:         add token to pendingOps;
10:    else if token is ")" then
11:        lastToken = pop from pendingOps;
12:        while lastToken != "(" do
13:            add lastToken to postfixExp;
14:            lastToken = pop from pendingOps;
15:        end while
16:    else if token is an operator then
17:        while pendingOps.top is an operator with equal or higher precedence than token do
18:            lastToken = pop from pendingOps;
19:            add lastToken to postfixExp;
20:        end while
21:        add token to pendingOps;
22:    end if
23: end for
24: return postfixExp;
```

an opening bracket is found, and the bracket will just be discarded. As the last option, if the token is an operator of any type (logical negation, arithmetic operator, logical operator, or logical connector), we will add to the postfix expression all top operators from the pending operations while the top is an operator with higher or equal precedence than the current token; and then, add the token to the stack of pending operations. The next operators are ordered by higher to lower precedence:

1. $\neg$
2. *, / both with same precedence
3. +, - both with same precedence
4. $<, \leq, >, \geq, =, \neq$ both with same precedence
5. $\wedge$
6. $\vee$

Finally, we will return our postfix expression vector.

Algorithm 8 – EvaluateRelCond($_c\mathsf{T}$, t, c)

Input: $_c\mathsf{T}$: relational conditional set, t: tuple of interest, c: predicate
Output: *result*: array of bits indicating each tuple of $_c\mathsf{T}$ that satisfies c for t

```
1:  create stack operands;
2:  create stack subresults;
3:  exp = ToPostfix(c);                                   ▷ from Algorithm 7
4:  for each token in exp do
5:      switch token do
6:          case relation's column name _cT.A do
7:              push token into operands;
8:          case tuple's column value t.A do
9:              push value of t.A into operands;
10:         case constant value val do
11:             push token into operands;
12:         case arithmetic operator ⊗ do
13:             valR = pop from operands;
14:             valL = pop from operands;
15:             push value of valL ⊗ valR into operands;
16:         case logical operator ⊙ do
17:             val = pop from operands;
18:             A = pop from operands;
19:             subresult = IndexQuery(_cT, A, val, ⊙);
20:             push subresult into subresults;
21:         case negation ¬ do
22:             R = pop from subresults;
23:             push ¬R into subresults;
24:         case logical connector ⋄ do
25:             R = pop from subresults;
26:             L = pop from subresults;
27:             push result of L ⋄ R into subresults;
28:     end switch;
29: end for;
30: return top from subresults;
```

Algorithm 8 identifies which tuples of a relational conditional set $_c\mathsf{T}$ satisfy a predicate c for a tuple of interest t. As it was discussed before, c is stored in a stack following the postfix notation, and there exist index structures for each column of $_c\mathsf{T}$. For instance, this algorithm could take one tuple t from relation *DP* of our motivational example shown in Figure 2, such as $t = (FarmAnimals, Horse, 4, 16)$, and return the tuples from relation *SP* that conditionally satisfy t using predicate c. As a result, we would have one array of bits 10000, where a 1 in an i^{th} position would mean that the i^{th} tuple of relation *SP* satisfies the condition, while zeros would have the opposite meaning. Therefore, only the first tuple of *SP* would satisfy c for t. As it is shown in Lines 1-2 of Algorithm 8, the first step is to create two stacks: one to store the next

operations to be performed and another stack to store the subresults of the previous operands. Then, as shown in Line 3, we need to get the postfix notation of the expression and save it into *exp*. Now, for each token read from the expression *exp*, we execute an action according to the type of the token: a) if the token refers to an attribute from relation ${}_c\mathsf{T}$ (Lines 6-7), the token is pushed into the operands stack; b) if the token refers to an attribute from tuple t (Lines 8-9), the attribute value is pushed into the operands stack; c) if the token is a constant value (Lines 10-11), that value is pushed into the operands stack; d) if the token is an arithmetic operator $\otimes$ (Lines 12-15), we pop the last two operands from the operands stack and push into the same stack the result of the arithmetic operation; e) if the token is a logical operator $\odot$ (Lines 16-20), we pop the last two operands from the operands stack. One will always be a constant value *val*, and the other will refer to an attribute A of relation ${}_c\mathsf{T}$. Then, we perform an indexed query according to the logical operator. That is, the query retrieves each tuple $t_i \in {}_c\mathsf{T}$ such that $t_i.A \odot val$ is true. The result is represented as an array of bits, where the i^{th} bit is 1 if the i^{th} tuple of ${}_c\mathsf{T}$ is returned by the query, and it is 0 otherwise. Then, the array is pushed into the subresults stack; f) if the token is a negation $\neg$ (Lines 21-23), we negate the top of stack subresults, and; g) as the last option, if the token is a logical connector $\diamond$ (Lines 24-27), we pop the last two arrays of bits from stack subresults, perform the logical operation and push the resulting array into the same stack. Finally, once the loop ends, we have only one array of bits in the subresults stack; it is returned as the final result of the whole operation.

Algorithm 9 implements the relational conditional set membership $\in_c$ from Definition 8. Thus, it identifies whether or not a tuple t is a conditional member of a set ${}_c\mathsf{T}$ according to a predicate c. The algorithm is twofold: a) use Algorithm 8 to get an array of bits representing the tuples of ${}_c\mathsf{T}$ that satisfy the condition, and; b) return $true$ if there is any bit 1 in the array; return $false$, otherwise.

Algorithm 9 – IsCondMember(${}_c\mathsf{T}$, t, c)

Input: ${}_c\mathsf{T}$: relational conditional set, t: tuple of interest, c: predicate
Output: $true$ if $t \in_c {}_c\mathsf{T}$; $false$, otherwise

```
1: result = EvaluateRelCond(cT, t, c);                ▷ from Algorithm 8
2: if result has any bit 1 then
3:     return true;
4: end if;
5: return false;
```

Algorithm 10 implements both the conditional intersection $\cap_c$ and the conditional difference $-_c$ from Definitions 10 and 11, respectively. It receives as parameters the left relation ${}_c\mathsf{T}_1$, the right relation ${}_c\mathsf{T}_2$, the predicate c, and an indicator $SetOp \in \{\cap_c, -_c\}$ of the operation of interest. The result ${}_c\mathsf{T}_R$ is either ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$ or ${}_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2$, according to $SetOp$. As it is shown in Lines 1-6, the algorithm begins by creating an array of bits R to be used latter to indicate the tuples of ${}_c\mathsf{T}_1$ that should be in ${}_c\mathsf{T}_R$. If $SetOp = \cap_c$, R is initialized with 0s; otherwise, it receives

1s. Now, as a first possibility as it is shown in lines 7-16, is that the right relation is the one with index structures. Here, there is a loop in lines 8-16 iterating for each tuple $t_i \in {}_c\mathsf{T}_1$. The loop updates array R only when the current tuple t_i is a conditional member of ${}_c\mathsf{T}_2$. In this case, $R[i]$ receives 1 if $SetOp = \cap_c$; otherwise, $R[i]$ is set to 0. As the second possibility, we could have the index structures not in the right relation but in the left one, as it is shown in lines 17-26. Here, there is a loop in lines 18-26 iterating for each tuple $t_j \in {}_c\mathsf{T}_2$. The first step in this loop is to compute the sub-result S as a bits vector indicating for each tuple of ${}_c\mathsf{T}_1$ if it satisfies the predicate with t_j. In this case, we will update our result R with a bit-wise operation of $R \vee S$ if $SetOp = \cap_c$; otherwise, R is set to $R \wedge S$. Finally, ${}_c\mathsf{T}_R$ is obtained as being the tuples of ${}_c\mathsf{T}_1$ that refer to each bit 1 in R.

Algorithm 10 – RelCondSetOp(${}_c\mathsf{T}_1$, ${}_c\mathsf{T}_2$, c, $SetOp$)

Input: ${}_c\mathsf{T}_1$, ${}_c\mathsf{T}_2$: relational conditional sets, c: predicate, $SetOp$: $\cap_c$ or $-_c$
Output: the result ${}_c\mathsf{T}_R$ from ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$ or from ${}_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2$, according to $SetOp$

```
1: create an array of bits R of size |_cT_1|;
2: if SetOp is ∩_c then
3:     initialize R with 0s;
4: else
5:     initialize R with 1s;
6: end if;
7: if _cT_2 is the table with index structures then
8:     for each tuple t_i ∈ _cT_1 do
9:         if IsCondMember(_cT_2, t_i, c) then                ▷ from Algorithm 9
10:            if SetOp is ∩_c then
11:                set R[i] as 1;
12:            else
13:                set R[i] as 0;
14:            end if;
15:        end if;
16:    end for;
17: else if _cT_1 is the table with index structures then
18:    for each tuple t_j in _cT_2 do
19:        S = EvalRelCond(_cT_1, t_j, c);                    ▷ from Algorithm 8
20:        if SetOp is ∩_c then
21:            R = R ∨ S;
22:        else
23:            R = R ∧ ¬S;
24:        end if
25:    end for
26: end if
27: _cT_R = get tuples from _cT_1 that refer to each bit 1 in R;
28: return _cT_R;
```

Algorithm 11 implements the relational conditional subset operator $\subseteq_c$ from Definition 9. Thus, it receives as parameters the left relation ${}_c\mathsf{T}_1$, the right relation ${}_c\mathsf{T}_2$, and the predicate c.

This short algorithm simply executes the conditional intersection ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$, and verifies it the result of this operation is equal to relation ${}_c\mathsf{T}_1$. If so, the algorithm returns *true*; otherwise, it returns $false$.

Algorithm 11 – IsCondSubset(${}_c\mathsf{T}_1$, ${}_c\mathsf{T}_2$, c)

Input: ${}_c\mathsf{T}_1$, ${}_c\mathsf{T}_2$: relational conditional sets, c: predicate
Output: *true* if ${}_c\mathsf{T}_1 \subseteq_c {}_c\mathsf{T}_2$; $false$, otherwise

1: **if** RelCondSetOp(${}_c\mathsf{T}_1$, ${}_c\mathsf{T}_2$, c, $\cap_c$) = ${}_c\mathsf{T}_1$ **then** ▷ from Algorithm 10
2: **return** *true*;
3: **end if**;
4: **return** $false$;

5.1.1 Complexity Analysis:

For the analysis, let us consider p to be the number of tokens that a predicate c has, while m and n are the cardinalities of relations ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$, respectively. Algorithm 7 only iterates the size of the predicate that is passed as parameter, using simple push and pop operations of the stack. As all process is executed accessing memory, the time is negligible and the complexity would be only $\mathcal{O}(1)$. Algorithm 8 iterates each predicate token. Each iteration runs an indexed query over a given relation; let us assume it to be ${}_c\mathsf{T}_2$. The algorithm takes advantage of existing index structures to perform the queries, so the use of state-of-the-art indexes allows each execution of function IndexQuery in Line 17 to cost $\mathcal{O}(\log n + s)$ time, where $s \leq n$ is the number of tuples selected by the query. Thus, the overall time complexity of Algorithm 8 is $\mathcal{O}(p(\log n + s))$. Algorithm 9 executes Algorithm 8 using as parameters a tuple t, a predicate c and a given relation; again, let us assume it to be ${}_c\mathsf{T}_2$. Then, the algorithm looks at negligible cost for a bit 1 in the resulting array of bits. Thus, the overall time complexity of Algorithm 9 is $\mathcal{O}(p(\log n + s))$. Algorithm 10 initializes at negligible cost an array of bits. Then, according to where the index structures are located. If index is in right relation ${}_c\mathsf{T}_2$, the loop in Lines 8-13 executes Algorithm 9 for each of the m tuples in ${}_c\mathsf{T}_1$, using ${}_c\mathsf{T}_2$ as a parameter with cost $\mathcal{O}(mp(\log n + s))$. If index is in left relation ${}_c\mathsf{T}_2$ the loop in Lines 18-25 executes Algorithm 8 for each of the n tuples in ${}_c\mathsf{T}_2$, using ${}_c\mathsf{T}_1$ as a parameter with cost $\mathcal{O}(np(\log m + s))$. Then, the whole function costs $\mathcal{O}(p(m(\log n + s) + n(\log m + s)))$ in total. At last, in Line 14, relation ${}_c\mathsf{T}_1$ is scanned at cost $\mathcal{O}(m)$. Thus, the overall cost of Algorithm 10 is $\mathcal{O}(mp(\log n + s) + m)$. Algorithm 11 runs Algorithm 10 and validates at cost $\mathcal{O}(m)$ if the result is equal to ${}_c\mathsf{T}_1$. Thus, the overall complexity of Algorithm 11 is $\mathcal{O}(p(m(\log n + s) + n(\log m + s)) + m)$. Finally, let us emphasize that p tends to be small in practice because large predicates are rare, so our algorithms are fast and scalable as long as the query selectivity s is also small, just like it happens with any index-based access method.

5.2 Conclusion

We have presented in this chapter all necessary algorithms to support our relational conditional set operations. In this case, we started with Algorithm 7 which converts an infix expression into postfix notation, allowing us to read a user formatted query, and after its conversion to postfix, to be able to evaluate it efficiently with Algorithm 8. This algorithm for evaluation will require the relational conditional set, a tuple, and a predicate; and identifies which tuples of the relational conditional set satisfy a predicate for the tuple of interest. Then, we present our Algorithm 9, which answers if a certain tuple is a conditional member of a relational conditional set for a given predicate. Also, this algorithm is used by Algorithm 10 that performs the conditional intersection or conditional difference, according to a parameter, returning the resulting conditional set for the desired operation. Finally, we also presented Algorithm 11, which verifies if one conditional subset is a conditional member of another. Additionally, we discussed the complexity analysis of all our algorithms.

CHAPTER

6

EXPERIMENTS

We evaluated the semantics and the usability of our operators, as well as the scalability of our algorithms by following the motivational example of product sales. The experiments were performed to answer two main Research Questions:

- **RQ1**: How accurate are the new relational conditional set operations in the sense of returning what the users expect to receive?
- **RQ2**: How effective and scalable are the algorithms that we propose?

Algorithms 7-11 were implemented in `C++` with page buffer management. Library `Arboretum`[1] was used to run indexed queries. All experiments were performed with an `Intel Core i7` processor working at 3.4GHz and 8GB of RAM.

Observation: for the purpose of reproducibility, all codes, results, and datasets studied in this work are freely available for download online[2].

6.1 Amazon Toys

Following our motivational example of product sales, we studied a dataset[3] of toy products from Amazon. The dataset was preprocessed to suit the purpose of our case study. Originally, it had 10,000 tuples and 16 attributes. However, each tuple representing a product contained information about a collection of suppliers and their respective prices. Thus, we preprocessed the data to have one tuple per supplier of each product. Unfortunately, the number of units available of each product was the stock of all suppliers combined, and not the stock of each individual seller. Thus, we were forced to generate random numbers of units that are

[1] <https://bitbucket.org/gbdi/arboretum/>
[2] <https://github.com/alivasples/RCSetOp>
[3] <https://www.kaggle.com/PromptCloudHQ/toy-products-on-amazon>

lower than or equal to the corresponding total stock of each product. We also removed tuples with missing information. The preprocessed dataset is freely available for download online [2]. It has 25,457 tuples referring to 4,277 unique toy products, and 5 columns: supplier, category, product, units, and price.

6.2 Semantic Validation

This section investigates Research Question **RQ1** by validating the semantics of our operators in the case study of sales of products. To make it possible, we generated 200 conditional sets by sampling at random the Amazon Toys dataset; they represented 100 pairs of relations *DP* and *SP* to be given as input for the queries. As this experiment intended to validate semantics only, the numbers of tuples, i.e., products, in the relations are small, varying from 3 to 6, so that we could manually verify if the results are meaningful. For each pair *DP* and *SP*, we ran the 4 motivational queries that were described previously in Chapter 1, that is: Query **Q1** – "Can I buy a certain product X in the store?"; Query **Q2** – "Can I buy all the desired products in the store?"; Query **Q3** – "Which desired products can be bought in the store?", and; Query **Q4** – "Which desired products won't be found in the store?". Note that, for Query **Q1**, each pair led to several executions by verifying if one can buy each product in *DP* individually. In summary, correct results were obtained for all 100 pairs of relations; thus, we argue that they validate the semantics of our proposals. From the pairs of relations studied, one specific case shown in Tables 3 and 4 was the one that we took as inspiration for our motivational example of the introductory Figure 2. Table 5 shows the results obtained from our algorithms for our query **Q3**, this is, the intersection of $DP \cap_c SP$: the list of products that can be bought in the store. The tuples listed in the result are also the tuples that satisfy are true in the result of query **Q1**, the tuples that are conditional members of *SP*. In contrast, Table 6 shows the results obtained from our algorithms for our query **Q4**, this is, the difference of $DP -_c SP$: the list of products that can't be bought in the store. Here, the listed tuples, are those which are not conditional members of *SP*. As there were tuples in the left relation *DP* that were not conditional members of the right relation *DP*, consequently, our algorithm's answer for Query **Q2** is also $false$.

Table 3 – Relational conditional set Desired Products (*DP*) and query results

CATEGORY	PRODUCT	UNITS	PRICE
Fantasy	Robot	17	30.32
Farm Animals	Horse	4	15.57
Educational	Think. Game	4	52.8
Vehicles	Lexus	33	16.05

Table 4 – Relational conditional set Store Products (*SP*)

CATEGORY	PRODUCT	UNITS	PRICE
Farm Animals	Horse	2	6.24
Educational	Think. Game	4	52.8
Fantasy	Robot	9	32.51
Star Wars	Wicket	13	16.26
Accessories	Playmobil	20	12.99

Table 5 – Relational conditional intersection ($DP \cap_c SP$) results

CATEGORY	PRODUCT	UNITS	PRICE
Farm Animals	Horse	4	15.57
Educational	Think. Game	4	52.8

Table 6 – Relational conditional difference ($DP -_c SP$) results

CATEGORY	PRODUCT	UNITS	PRICE
Fantasy	Robot	17	30.32
Vehicles	Lexus	33	16.05

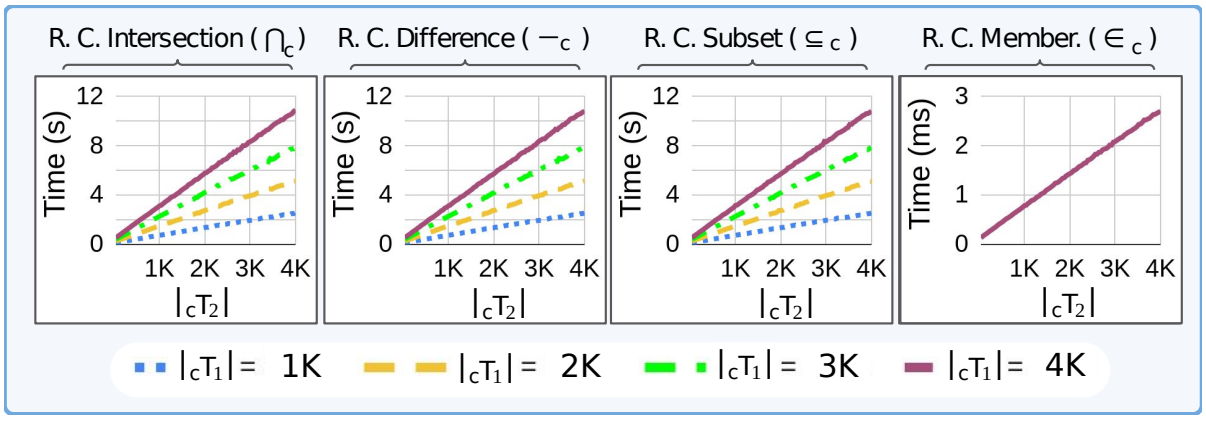


Figure 18 – Scalability of our conditional set operators in dataset Amazon Toys.

6.3 Scalability

This section investigates Research Question **RQ2** by evaluating the scalability of our algorithms. To make it possible, we generated random samples of varying sizes from dataset Amazon Toys to represent pairs of conditional sets ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$. Specifically, we created 100 pairs by varying the cardinality of ${}_c\mathsf{T}_1$ from 40 to 4,000 and using a fixed cardinality of 1,000 for ${}_c\mathsf{T}_2$. Other 300 pairs were created in a similar way: 100 pairs with $|{}_c\mathsf{T}_2| = 2{,}000$, 100 pairs with $|{}_c\mathsf{T}_2| = 3{,}000$ and 100 pairs with $|{}_c\mathsf{T}_2| = 4{,}000$. Then, operators $\cap_c$, $-_c$ and $\subseteq_c$ were executed 10 times for each of these pairs of relations to obtain average runtime results. A distinct procedure was necessary for operator $\in_c$, since it receives a tuple t and a single conditional set ${}_c\mathsf{T}$ as input; not, two sets. Thus, we took each of the 4,000 tuples from our largest relation ${}_c\mathsf{T}_1$ to be tuple t and executed $t \in_c {}_c\mathsf{T}_2$ to obtain the average runtime. The distinct versions of relation ${}_c\mathsf{T}_2$ were

considered, whose cardinalities go from $1,000$ to $4,000$. Figure 18 reports the results obtained from the aforementioned procedure. Each individual plot reports average runtime versus the cardinality of ${}_c\mathsf{T}_2$; distinct lines refer to distinct cardinalities of ${}_c\mathsf{T}_1$. As it can be seen, the results corroborate our theoretical analysis of complexity from our motivational example, by indicating that all of our algorithms are fast and scalable.

6.4 Conclusion

We discussed in this chapter the experiments performed so as to semantically validate our proposal and also achieve demonstrating the scalability of our algorithms. For this, we followed our motivational example of sales of amazon toys, and generated several short tests and big tests. From the short tests, we observed that all return the expected results, and picked up one of them to explain it in the results. With this, we semantically validated our proposal. With the big tests, we ran our algorithms many times and demonstrated the scalability of our algorithms.

CHAPTER

7

DISCUSSION

In this chapter, we will discuss two important points. Firstly, we discuss the generality and usability of our relational conditional set operations, showing other real-life applications where our operator could be used. Secondly, we discuss the reason why a relational conditional union operator was not defined in previous chapters as well as present how it would be its formal definition and algorithm.

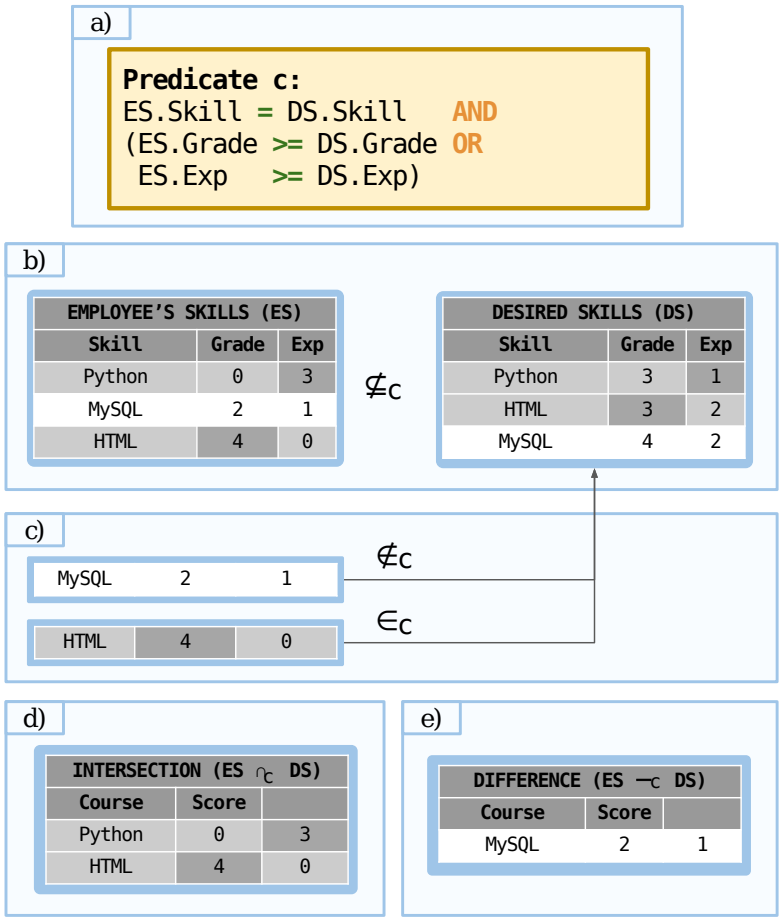
7.1 Usability and Generality

This section presents other applications where our operators are well suitable, thus corroborating their general usability.

7.1.1 Job Promotion

Let us consider a call for a job promotion supported by data from the Desired Skills (DS) for the job position and one candidate Employee's Skills (ES). Each skill can be quantified by a certification grade on a scale from 1 (beginner) to 5 (expert), with 0 for none; or, by the number of years of experience. Thus, we have relations with schemas Sch(DS) = Sch(ES) = ($Skill$, $Grade$, Exp). The traditional set-based operators would be helpless here, as they would not allow to verify if the employee has the minimal certification grade or the minimal experience for each desired skill. However, this condition can be easily treated by our operators as illustrated in Figure 19. Here, we only have to consider a predicate shown in Figure 19.a: $c : DS.Skill = ES.Skill \wedge (DS.Grade \leq ES.Grade \vee DS.Exp \leq ES.Exp)$. Now, one may design and execute queries like: a) "Does one skill $t \in ES$ satisfy any desired skill?" with $t \in_c DS$; b) "Does the employee satisfy all the desired skills?" with $DS \subseteq_c ES$; c) "Which desired skills does the employee satisfy?" with $DS \cap_c ES$, and; d) "Which are the desired skills that the employee does not satisfy?" with $DS -_c ES$.

Figure 19 – Example of RelCond Set Operations: Job Promotion.

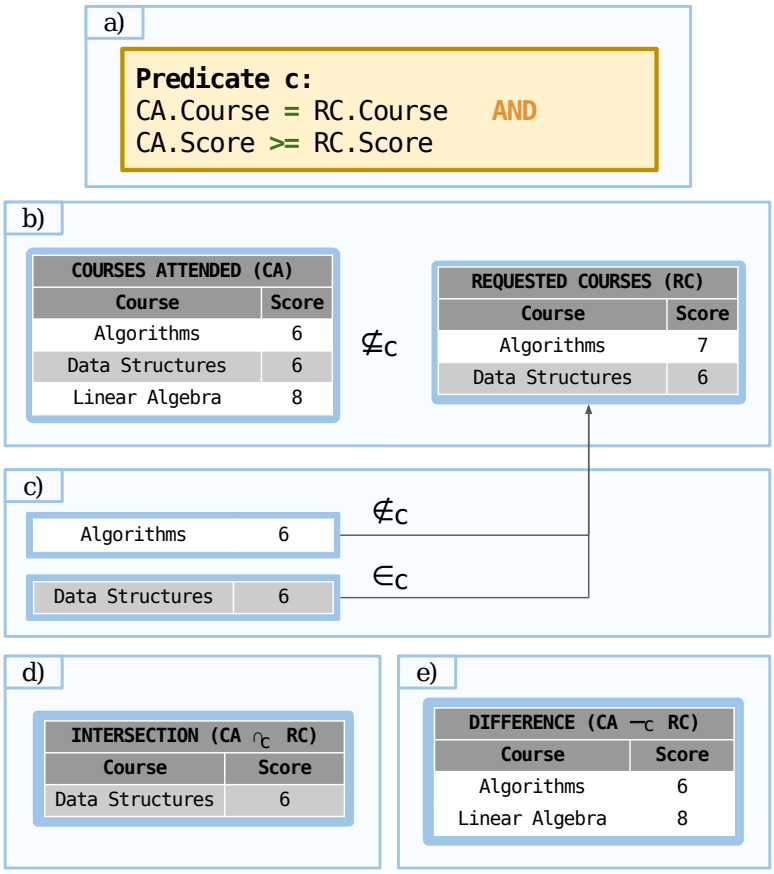


Source: Elaborated by the author.

7.1.2 Internship

For this case let us consider an organization that wants to recruit an intern. Naturally, the organization would like to analyze the best options among the applicants, and the first step would be to inspect their grades in the Courses Attended (CA) in college. Here, the organization would list the Requested Courses (RC) with minimal grades, and contact an applicant for an interview only if he/she satisfies those requisites. Therefore, the schemas $\text{Sch}(RC) = \text{Sch}(CA) = (Course, Score)$. The traditional set-based operators would be helpless here, as they would not allow one to verify if an applicant has the minimal grades for the requested courses. Fortunatelly, our operators are promptly applicable as illustrated in Figure 19. Here, we only have to consider a predicate shown in Figure 20.a: $c : RC.Course = CA.Course \wedge RC.Score \leq CA.Score$. Now, one may design and execute queries like: a) "Does a certain applicant's course $t \in CA$ satisfy any of the requested courses?" with $t \in_c RC$; b) "Does the applicant satisfy all the requested courses?" with $RC \subseteq_c CA$; c) "Which requested courses does the applicant satisfy?" with $RC \cap_c CA$, and; d) "Which are the requested courses that the applicant do not satisfy?" with $RC -_c CA$.

Figure 20 – Example of RelCond Set Operations: Internship selection.



Source: Elaborated by the author.

7.2 Conditional Union

In this work, we have expanded different concepts from the Theory of Sets. However, we did not discuss a potential conditional union operation because we could not identify practical utility for it. For example, let us consider once again the motivational case study on sales of products that is illustrated in Figure 2. The union of relations *DP* and *SP* would return the products that are either desired by the client or available in the store. In the job promotion case from Section 7.1.1, the union of *DS* and *ES* would be the skills that are either desired by the employer or present in the employee. Also, in the internship example of Section 7.1.2, the union of *RC* and *CA* would be the courses that are either requested to recruit applicants or in the applicant's curriculum. In our humble opinion, none of these results seem to be meaningful for practical use. However, it would be straightforward to define a relational conditional union operator $\cup_c$, if it is required in any future work. Thus, we present its formal definition in Definition 12.

Definition 12. The **relational conditional union operation** ($\cup_c$) is a binary operation represented as ${}_c\mathsf{T}_1 \cup_c {}_c\mathsf{T}_2 = {}_c\mathsf{T}_R$, in which ${}_c\mathsf{T}_R$ has the result of the conditional union between ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$. Relations ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$ must be union compatible. The resulting relation ${}_c\mathsf{T}_R$ has all tuples of

${}_c\mathsf{T}_1$ in addition to all tuples of ${}_c\mathsf{T}_2$ that are not conditional members of ${}_c\mathsf{T}_1$. Formally, we have:

$$ {}_c\mathsf{T}_R = {}_c\mathsf{T}_1 \cup \{t_i : t_i \in {}_c\mathsf{T}_2 \wedge t_i \notin_c {}_c\mathsf{T}_1\} \tag{7.1}$$

The relational conditional intersection may not be commutative. Thus, queries ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$ and ${}_c\mathsf{T}_2 \cap_c {}_c\mathsf{T}_1$ do not necessarily produce the same results.

Algorithm 12 – RelCondSetOp(${}_c\mathsf{T}_1$, ${}_c\mathsf{T}_2$, c, $SetOp$)

Input: ${}_c\mathsf{T}_1$, ${}_c\mathsf{T}_2$: relational conditional sets, c: predicate, $SetOp$: $\cap_c$ or $-_c$ or $\cup_c$
Output: the result ${}_c\mathsf{T}_R$ from ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$ or from ${}_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2$, according to $SetOp$

```
1: create an array of bits R1 of size |cT1|;
2: create an array of bits R2 of size |cT2|;
3: if SetOp is ∩c then
4:     initialize R1 with 0s;
5: else
6:     initialize R1 with 1s;
7: end if;
8: initialize R2 with 0s;
9: for each tuple ti ∈ cT1 do
10:    S = Index_TupleQuery(cT2, ti, c);
11:    if SetOp is ∩c and S has any bit 1 then
12:        set R1[i] as 1;
13:    else if SetOp is −c and S has any bit 1 then
14:        set R1[i] as 0;
15:    else
16:        R2 = R2 ∨ S
17:    end if;
18: end for;
19: cTR1 = get tuples from cT1 that refer to each bit 1 in R1;
20: cTR2 = get tuples from cT2 that refer to each bit 1 in R2;
21: return cTR1 ∪c cTR2;
```

Also, it would be easy to adapt our Algorithm 10 to support the conditional union. This adaptation is studied in Algorithm 12. Algorithm 12 receives as parameters the left relation ${}_c\mathsf{T}_1$, the right relation ${}_c\mathsf{T}_2$, the predicate c, and an indicator $SetOp \in \{\cap_c, -_c, \cup_c\}$ of the operation of interest. The result ${}_c\mathsf{T}_R$ is either ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$, ${}_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2$, or ${}_c\mathsf{T}_1 \cup_c {}_c\mathsf{T}_2$, according to $SetOp$. As it is shown in Lines 1-8, the algorithm begins by creating the arrys of bits R_1 and R_2 to be used latter to indicate the tuples of ${}_c\mathsf{T}_1$ that should be in ${}_c\mathsf{T}_{R1}$ and the tuples of ${}_c\mathsf{T}_2$ that should be in ${}_c\mathsf{T}_{R2}$ respectively. If $SetOp = \cap_c$, R_1 is initialized with 0s; otherwise, it receives 1s. In all cases, R_2 will be initialized with 0s. The main loop in Lines 9-18 iterates for each tuple $t_i \in {}_c\mathsf{T}_1$. The first step in this loop is to compute the sub-result S as a bits vector indicating for each tuple of ${}_c\mathsf{T}_2$ if it satisfies the predicate with t_i. Then, if $SetOp = \cap_c$ and any bit of S is 1, $R[i]$ receives 1. If instead $SetOp = -_c$ and any bit of S is 1, $R[i]$ receives 1. Otherwise, R_2 is set to $R_2 \vee S$. Finally,

${}_c\mathsf{T}_{R1}$ is obtained as being the tuples of ${}_c\mathsf{T}_1$ that refer to each bit 1 in R_1, ${}_c\mathsf{T}_{R2}$ is obtained as being the tuples of ${}_c\mathsf{T}_2$ that refer to each bit 1 in R_2 and the union of both ${}_c\mathsf{T}_{R1} \cup {}_c\mathsf{T}_{R2}$ is returned.

7.3 Conclusion

We discussed in this chapter two important points. As first point, we discuss the generality and usability of our relational conditional set operations. Here, we presented an example of job promotion and one of internship. We discussed how the left and right relations would be constructed and which queries would be answered by the conditional membership, subset, intersection, and difference. Also, we illustrated both examples. The other important point was to justify why we didn't present the relational conditional union operation as a main part of our proposal. Still, we discuss how it would be its definition and the algorithm to support it.

CHAPTER

8

CONCLUSIONS AND FUTURE WORK

In this chapter we present our final conclusions as well as discussing future work.

8.1 Conclusions

In this work, we demonstrated that the set-based operations of the Relational Algebra have severe limitations that prevent their use in several real-world applications. Thus, we tackled the problem by means of four main contributions:

C1 Operators Design and Usability – We first discussed about the traditional set operators and point that they are very useful when elements are compared by identity. That is, when it makes sense to compare elements by the implicit identity predicate that says that a tuple is a member of a relation if it is identical to any tuple in the set. Also, we explored the related works on the relational set operations, many of them even extend the operators to work with other type of data, such as complex data. When we are working with data types as images, videos, long texts, and other complex elements, we can use the similarity-aware sets and the similarity-aware binary operations. However, even when we found several works that extend the set operations, none of them allow us to compare tuples by custom predicate predicates. Therefore, we tackled the problem by presenting the new **Relational Conditional Set Operators** that are naturally well suited to answer queries using custom predicates.

C2 Formal Definition and Algorithms – we formally defined the new conditional set operators as the relational conditional Set Membership $\in_c$, relational conditional Subset $\subseteq_c$, relational conditional Intersection $\cap_c$, and relational conditional Difference $-_c$, thus enabling their usage in queries along with the existing algebraic operators. Additionally, we designed novel algorithms that execute these operators in a fast and scalable manner.

C3 Semantic Validation – we performed a case study by analyzing real data from thousands of toy products available for sale at Amazon. We generated several datasets from the original. Then, we ran our relational conditional set algorithms with them. The results corroborate the practical usability of our operators in real-life applications.

C4 Generality and Usability – we exemplified other cases of use where our operators are well suitable, thus corroborating their generality and usability. Here, we presented an example of job promotion with desired skills and grades or certifications, and the other example of students internships, considering graduation scores. We presented schemas for both cases constructed and the queries that can be answered with our relational conditional set membership, subset, intersection, and difference operations. Thus, corroborating that our operators can be used in more real-life applications.

C5 Basis for other operators – our new operators can also be used as the basis for other relational algebra operations. For example, in Appendix A, we discuss the utility of generalizing the relational division to a new operator that can also support custom predicates, using as a basis our relational conditional set operations.

In addition, as part of this MSc work, we generated two publications, the first one (LIMA *et al.*, 2020) as collaborator together with other authors, and the second one (LESCANO; CORDEIRO, 2021) is the publication that shares our hypothesis results.

1. Afonso Lima, Alexander Florez, Alexis Aspauza, João Novaes, Natalia Martins, Caetano Traina, Elaine P. M. de Sousa, José F. Rodrigues Júnior, Robson L. F. Cordeiro: Analysis of ENEM's attendants between 2012 and 2017 using a clustering approach. In: Journal of Information and Data Management 2020. Pages 115-130. Available at: <https://periodicos.ufmg.br/index.php/jidm/article/view/24835>.

2. Alexis Iván Aspauza Lescano, Robson Leonardo Ferreira Cordeiro: Relational Conditional Set Operations. In: 25th European Conference on Advances in Databases and Information Systems. Pages 38-49. Available at: <https://link.springer.com/chapter/10.1007%2F978-3-030-85082-1_4>.

8.2 Future Work

This MSc work achieved all proposed objectives and yet, it is just a tiny part of science. It opens then, path for future works such as follows:

- We formally defined our relational conditional set operators and proposed scalable algorithms to support them. Thus, one of the closest future works would be to implement these algorithms into Database Management Systems.

- In parallel, we could also propose SQL sentences to express our relational conditional set operations in DBMS. Then, we can even compare the results and running measurements against our base algorithms.

- Also, in contrast to the traditional set operations where the tuples in the intersection's result are member of both relations, here they do not necessary belong to the right relation but always do it to the left one. Consequently, another future work can explore the utility of allowing the user to specify the default's relation result.

- When we work with SQL, we actually work with bags or multisets. This is, a relation can actually have duplicate elements. Thus, future works can generalize the use of our relational conditional set operations to work with multisets.

- We presented scalable algorithms to support our novel operators. However, these algorithms are only a starting point. Then, there is a whole new path to explore the optimization of these novel operators.

- So far we only used simple data in our proposal. However, we could also extend the concept to support similarity in future works. For example, in our motivation example of sales of toys, we could compare images of the toys, but keeping the other conditional elements, such as quantity and price.

- Finally, the algebra of sets is such a fundamental concept that we believe that future works can explore the usability of our novel operators as basis of other operations. As an example, we present in Appendix A how the relational division could be generalized to also support custom predicates.

BIBLIOGRAPHY

Al Marri, W. J. *et al.* The similarity-aware relational database set operators. **Information Systems**, Elsevier Ltd, v. 59, p. 79–93, 2016. ISSN 03064379. Citations on pages 54 and 56.

ALSALLAKH, B.; MICALLEF, L.; AIGNER, W.; HAUSER, H.; MIKSCH, S.; RODGERS, P. The State-of-the-Art of Set Visualization. **COMPUT GRAPH FORUM**, Blackwell Publishing Ltd, v. 35, n. 1, p. 234–260, 2016. ISSN 01677055. Citation on page 33.

BARIONI, M. C. N.; RAZENTE, H. L.; TRAINA, A. J. M.; Traina Jr., C. Seamlessly integrating similarity queries in {sql}. **Softw., Pract. Exper.**, v. 39, n. 4, p. 355–384, 2009. Citation on page 53.

BELOHLAVEK, R.; VYCHODIL, V. Query systems in similarity-based databases: Logical foundations, expressive power, and completeness. In: **Proc. of the 2010 ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2010. (SAC '10), p. 1648–1655. ISBN 978-1-60558-639-7. Citation on page 53.

BILLE, P. *et al.* Fast Evaluation of Union-Intersection Expressions. In: TOKUYAMA, T. (Ed.). **Algorithms and Computation**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 739–750. ISBN 978-3-540-77120-3. Citations on pages 49 and 56.

BOSC, P.; PIVERT, O. On a fuzzy bipolar relational algebra. **Information Sciences**, Elsevier, v. 219, p. 1–16, 2013. ISSN 00200255. Citation on page 53.

BUDíKOVá, P.; BATKO, M.; ZEZULA, P. Query language for complex similarity queries. In: **ADBIS**. [S.l.: s.n.], 2012. p. 85–98. Citation on page 53.

CAMPS, D. High Performance Relational Division in SQL Server. **Simple-Talk**, Red Gate, 2014. Citations on pages 48 and 95.

CAO, B.; BADIA, A. SQL query optimization through nested relational algebra. **ACM Transactions on Database Systems**, v. 32, n. 3, 2007. ISSN 03625915. Available: <http://dx.doi.org/10.1145/1272743.1272748>. Citations on pages 47 and 49.

CELKO, J. Divided We Stand: The SQL of Relational Division. **Simple-Talk**, Red Gate, 2009. Citations on pages 48 and 95.

CODD, E. F. Relational completeness of data base sublanguages. **IBM Research Report**, San Jose, California, RJ987, 1972. Citations on pages 27, 35, 36, 39, 89, 93, and 95.

____. **The Relational Model for Database Management: Version 2**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. 538 pages. ISBN 0-201-14192-2. Citations on pages 27, 35, 36, 39, 89, and 93.

DENG, D.; LI, G.; WEN, H.; FENG, J. An efficient partition based method for exact set similarity joins. **Proc. VLDB Endow.**, VLDB Endowment, v. 9, n. 4, p. 360–371, 2015. ISSN 2150-8097. Citation on page 53.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems**. 7th. ed. [S.l.]: Pearson, 2015. ISBN 0133970779, 9780133970777. Citations on pages 27, 35, 36, 48, and 95.

GALINDO, J.; URRUTIA, A.; PIATTINI, M. **Fuzzy Databases: Modeling, Design and Implementation: Modeling, Design and Implementation**. [S.l.]: Idea Group, 2005. ISBN 1-59140-324-3. Citation on page 53.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database Systems: The Complete Book**. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. ISBN 9780131873254. Citations on pages 35 and 36.

GONZAGA, A.; CORDEIRO, R. L. The similarity-aware relational division database operator with case studies in agriculture and genetics. **Information Systems**, Elsevier Ltd, v. 82, p. 71–87, 2019. ISSN 03064379. Citations on pages 53 and 95.

GONZAGA, A. S.; CORDEIRO, R. L. F. Fast and scalable relational division on database systems. In: **Brazilian Symposium on Databases - SBBD**. [S.l.]: SBC, 2016. Citations on pages 48 and 95.

____. A new division operator to handle complex objects in very large relational datasets. In: **International Conference on Extending Database Technology - EDBT**. [S.l.]: Open Proceedings, 2017. Citations on pages 53 and 95.

GORMAN, J. *et al.* Learning relational algebra by snapping blocks. In: **SIGCSE 2014 - Proceedings of the 45th ACM Technical Symposium on Computer Science Education**. Atlanta, GA, United states: [s.n.], 2014. p. 73–78. Available: <http://dx.doi.org/10.1145/2538862.2538961>. Citations on pages 47 and 49.

IMAMUDDIN, A.; NAHAR, I.; CHANDRA, S. {TransJoin}: An Algorithm to Implement Division Operator of Relational Algebra in Structured Query Language. **Journal of Physics: Conference Series**, {IOP} Publishing, v. 1477, p. 32003, 2020. Citations on pages 48 and 95.

JACOX, E. H.; SAMET, H. Metric space similarity joins. **ACM Trans. Database Syst.**, ACM, New York, NY, USA, v. 33, n. 2, p. 7:1—-7:38, 2008. ISSN 0362-5915. Citation on page 53.

JIANG, Y.; LI, G.; FENG, J.; LI, W.-S. String similarity joins: An experimental evaluation. **Proc. VLDB Endow.**, VLDB Endowment, v. 7, n. 8, p. 625–636, 2014. ISSN 2150-8097. Citation on page 53.

KALASHNIKOV, D. V. {Super-EGO}: Fast multi-dimensional similarity join. **The VLDB Journal**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 22, n. 4, p. 561–585, 2013. ISSN 1066-8888. Citation on page 53.

KESSLER, J. *et al.* RelaX: A webbased execution and learning tool for relational algebra. In: **Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)**. Rostock, Germany: [s.n.], 2019. P-289, p. 503–506. ISSN 16175468. Available: <http://dx.doi.org/10.18420/btw2019-32>. Citations on pages 47 and 49.

KVET, M.; MATIASKO, K. Flower Master Index for Relational Database Selection and Joining. In: **DISA 2020**. [S.l.: s.n.], 2021. p. 181–202. ISBN 978-3-030-63871-9. Citation on page 47.

LESCANO, A. I. A.; CORDEIRO, R. L. F. Relational Conditional Set Operations. p. 38–49, 2021. Available: <https://link.springer.com/10.1007/978-3-030-85082-1_4>. Citation on page 80.

LIMA, A.; FLOREZ, A.; LESCANO, A.; NOVAES, J.; MARTINS, N.; JUNIOR, C. T.; SOUSA, E. P. M. de; JÚNIOR, J. F. R.; CORDEIRO, R. L. F. Analysis of ENEM's attendants between 2012 and 2017 using a clustering approach. **J. Inf. Data Manag.**, v. 11, n. 2, 2020. Available: <https://periodicos.ufmg.br/index.php/jidm/article/view/24835>. Citation on page 80.

LITORIYA, R.; RANJAN, A. Implementation of Relational Algebra Interpreter Using Another Query Language. In: **2010 International Conference on Data Storage and Data Engineering**. [S.l.: s.n.], 2010. p. 24–28. Citations on pages 47 and 49.

LU, W. *et al.* MSQL: efficient similarity search in metric spaces using SQL. **VLDB Journal**, v. 26, n. 6, p. 829–854, 2017. ISSN 10668888. Available: <http://dx.doi.org/10.1007/s00778-017-0481-6>. Citation on page 53.

LU, W.; ZHANG, X.; SHUI, Z.; PENG, Z.; ZHANG, X.; DU, X.; HUANG, H.; WANG, X.; PAN, A.; LI, H. MSQL+: A plugin toolkit for similarity search under metric spaces in distributed relational database systems. In: **Proceedings of the VLDB Endowment**. [s.n.], 2018. v. 11, n. 12, p. 1970–1973. ISSN 21508097. Available: <http://dx.doi.org/10.14778/3229863.3236237>. Citation on page 53.

MARRI, W. J. A. *et al.* The Similarity-Aware Relational Intersect Database Operator. In: TRAINA, A. J. M. *et al.* (Ed.). **7th SISAP**. Cham: Springer International Publishing, 2014. p. 164–175. ISBN 978-3-319-11988-5. Citations on pages 54 and 56.

MATOS, V.; GRASSER, R. Teaching tip: A simpler (and better) sql approach to relational division. **Journal of Information Systems Education**, v. 13(2), p. 85–88, 2002. Citations on pages 48 and 95.

MCMASTER, K. *et al.* A relational algebra query language for programming relational databases. In: **Proceedings of ISECON**. Nashville, TN, United states: [s.n.], 2010. p. Educ. Spec. Interest Group Assoc. Inf. Technol. Pr. ISSN 15427382. Citations on pages 47 and 49.

____. Relational algebra programming with Microsoft access databases. **Interdisciplinary Journal of Information, Knowledge, and Management**, v. 6, p. 73–83, 2011. ISSN 15551229. Available: <http://dx.doi.org/10.28945/1365>. Citations on pages 47 and 49.

MCMASTER, K.; SAMBASIVAM, S.; HADFIELD, S. Relational algebra and SQL: Better together. **Proceedings of the Information Systems Education Conference, ISECON**, Association of Information Technology Professionals, v. 29, 2012. ISSN 21671435. Citations on pages 47 and 49.

MILLER, B. N.; RANUM, D. L. **Problem Solving with Algorithms and Data Structures Using Python SECOND EDITION**. 2nd. ed. USA: Franklin, Beedle & Associates Inc., 2011. ISBN 1590282574. Citation on page 41.

NGO, H. Q.; PORAT, E.; RE, C.; RUDRA, A. Worst-case Optimal Join Algorithms. **Journal of the ACM**, v. 65, n. 3, 2018. ISSN 00045411. Available: <http://dx.doi.org/10.1145/3180143>. Citation on page 47.

PINTER, C. **Set theory**. Reading MA: Addison-Wesley Pub. Co., 1971. Citation on page 34.

POLA, I. R. *et al.* A new concept of sets to handle similarity in databases: The SimSets. In: **LNCS**. [S.l.]: Springer, Berlin, Heidelberg, 2013. v. 8199, p. 30–42. ISBN 9783642410611. ISSN 03029743. Citations on pages 52, 54, and 56.

____. Similarity sets: A new concept of sets to seamlessly handle similarity in database management systems. **Information Systems**, Elsevier Ltd, v. 52, p. 130–148, 2015. ISSN 03064379. Citations on pages 52, 54, and 55.

RAMAKRISHNAN, R.; GEHRKE, J. **Database Mangement System**. 2nd. ed. [S.l.]: McGraw-Hill Higher Education, 2000. ISBN 0072465352. Citations on pages 35, 36, and 93.

RAMSDEN, J. J. **Sets and combinatorics**. [S.l.]: Springer, Dordrecht, 2004. 47–52 p. Citation on page 33.

RASTOGI, R.; MONDAL, P.; AGARWAL, K. An exhaustive review for infix to postfix conversion with applications and benefits. **2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)**, p. 95–100, 2015. Citation on page 42.

RED'KO, V. N.; BUY, D. B.; KANARSKAYA, I. S.; SENCHENKO, A. S. Precise Estimates for the Time Complexity of Implementing Algorithms of Set-Theoretic Operations in Table Algebras. **Cybernetics and Systems Analysis**, Springer New York LLC, v. 53, n. 1, p. 1–11, 2017. ISSN 15738337. Citations on pages 49 and 56.

RONG, C.-T. *et al.* Fine-Granular Parallel Set Similarity Join Based on Multicores. **Jisuanji Xuebao/Chinese Journal of Computers**, v. 40, n. 10, p. 2320–2337, 2017. ISSN 02544164. Available: <http://dx.doi.org/10.11897/SP.J.1016.2017.02320>. Citation on page 53.

SANTOS, L. F. D.; OLIVEIRA, W. D.; FERREIRA, M. R. P.; TRAINA, A. J. M.; Traina Jr., C. Parameter-free and domain-independent similarity search with diversity. In: **Proceedings of the 25th International Conference on Scientific and Statistical Database Management**. New York, NY, USA: ACM, 2013. (SSDBM), p. 5:1—-5:12. ISBN 978-1-4503-1921-8. Citation on page 53.

SHARMA, A. K. *et al.* An extended relational algebra for fuzzy multidatabases. In: **7th I-SPAN**. [S.l.: s.n.], 2004. p. 445–450. Citation on page 53.

SHIN, H.; ON, B.-W.; LEE, I.; CHOI, G. S. Bucket-sorted hash join. **Journal of Information Science and Engineering**, v. 36, n. 1, p. 171–190, 2020. ISSN 10162364. Available: <http://dx.doi.org/10.6688/JISE.20200136(1).0010>. Citation on page 47.

SHIN, Y.; TEMUUJIN, O.; JEON, M.; AHN, J.; IM, D.-H. Implementation and Experiment of Join Optimization Algorithm for Inverted Index in an RDBMS. In: PARK, J. J.; FONG, S. J.; PAN, Y.; SUNG, Y. (Ed.). **Advances in Computer Science and Ubiquitous Computing**. [S.l.]: Springer Singapore, 2021. p. 79–85. ISBN 978-981-15-9343-7. Citation on page 47.

SILVA, Y.; AREF, W.; LARSON, P.-A.; PEARSON, S.; ALI, M. Similarity queries: their conceptual evaluation, transformations, and processing. **The VLDB Journal**, Springer-Verlag, v. 22, n. 3, p. 395–420, 2013. ISSN 1066-8888. Citation on page 53.

SILVA, Y. N.; ALY, A. M.; AREF, W. G.; LARSON, P. A. Simdb: A similarity-aware database system. In: **Proceedings of the ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 2010. p. 1243–1246. ISBN 9781450300322. ISSN 07308078. Citation on page 53.

SILVA, Y. N.; AREF, W. G.; ALI, M. H. The similarity join database operator. In: IEEE. **Data Engineering (ICDE), 2010 IEEE 26th International Conference on**. [S.l.], 2010. p. 892–903. Citation on page 53.

SILVA, Y. N.; PEARSON, S. Exploiting database similarity joins for metric spaces. **Proc. VLDB Endow.**, VLDB Endowment, v. 5, n. 12, p. 1922–1925, 2012. ISSN 2150-8097. Citation on page 53.

SILVA, Y. N.; PEARSON, S. S.; CHON, J.; ROBERTS, R. Similarity joins: Their implementation and interactions with other database operators. **Inf. Syst.**, v. 52, p. 149–162, 2015. Citation on page 53.

STOLL, R. R. **Set Theory and Logic**. [S.l.]: W.H. Freeman, 1963. Citations on pages 27 and 34.

TANG, X.; CHEN, G. A complete set of fuzzy relational algebraic operators in fuzzy relational databases. In: **FUZZ-IEEE**. [S.l.: s.n.], 2004. v. 1, p. 565–569. ISBN 0780383532. ISSN 10987584. Citation on page 53.

ULLMAN, J. D. **A first course in database systems**. [S.l.]: Pearson, 1997. 470 p. ISSN 1545-679X. ISBN 013600637X. Citations on pages 35, 36, 37, 38, and 39.

UNDE, M. D.; KURHE, P. S. Web based control and data acquisition system for industrial application monitoring. In: **ICECDS 2017**. [S.l.]: IEEE, 2018. p. 246–249. ISBN 9781538618868. Citation on page 33.

VASCONCELOS, G. Q.; KASTER, D. S.; CORDEIRO, R. L. On the Support of the Similarity-Aware Division Operator in a Commercial RDBMS. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [S.l.]: Springer Verlag, 2018. v. 11019 LNCS, p. 142–155. ISBN 9783319983974. ISSN 16113349. Citations on pages 53 and 96.

VASCONCELOS, G. Q.; ZABOT, G. F.; De Lima, D. M.; RODRIGUES, J. F.; TRAINA, C.; KASTER, D. D. S.; CORDEIRO, R. L. Tender-sims: Similarity retrieval system for public tenders. In: **ICEIS 2018 - Proceedings of the 20th International Conference on Enterprise Information Systems**. [S.l.]: SciTePress, 2018. v. 1, p. 143–150. ISBN 9789897582981. Citations on pages 53 and 96.

VINAYAKUMAR, R. *et al.* DB-Learn: Studying Relational Algebra Concepts by Snapping Blocks. In: **2018 9th International Conference on Computing, Communication and Networking Technologies, ICCCNT 2018**. Bengaluru, India: [s.n.], 2018. Available: <http://dx.doi.org/10.1109/ICCCNT.2018.8494181>. Citations on pages 47 and 49.

XUE, M.-T. *et al.* FPGA-Accelerated Hash Join Operation for Relational Databases. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 67, n. 10, p. 1919–1923, 2020. ISSN 15497747. Available: <http://dx.doi.org/10.1109/TCSII.2019.2959661>. Citation on page 47.

YU, C. T.; MENG, W. **Principles of database query processing for advanced applications**. San Francisco, California: Morgan Kaufmann Publishers Inc., 1998. 4–5 p. ISBN 1-55860-434-0. Citations on pages 35 and 36.

YU, M.; LI, G.; DENG, D.; FENG, J. String similarity search and join: a survey. **Frontiers of Computer Science**, v. 10, n. 3, p. 399–417, 2016. ISSN 2095-2236. Citation on page 53.

ZHAO, F. *et al.* A vague relational model and algebra. In: **4th FSKD 2007**. [S.l.: s.n.], 2007. v. 1, p. 81–85. ISBN 0769528740. Citation on page 53.

ZHOU, Z.; YU, C.; NUTANONG, S.; CUI, Y.; FU, C.; XUE, C. J. A Hardware-Accelerated Solution for Hierarchical Index-Based Merge-Join(Extended Abstract). In: **2019 IEEE 35th International Conference on Data Engineering (ICDE)**. [S.l.: s.n.], 2019. p. 2137–2138. Citation on page 47.

APPENDIX

A

RELATIONAL CONDITIONAL FOR ALL AND FOR ANY OPERATIONS

In Relational Algebra, the operator of division (÷) is an intuitive way to answer queries with the concept of "for all", and thus, it is constantly required in real applications. The dividend's tuples are grouped into sets and each set that contains the whole divisor as a subset is part of the quotient. However, these operations have limitations because of the implicit use of the identity predicate. That is, a tuple is a member of a set if it is identical to any tuple in the set. Still, many applications need other comparison predicates different than identity. Also, with a simple variation, we could also answer queries with the concept of "for any". This appendix presents the new **Relational Conditional For All (** $\forall_c$ **)** and **Relational Conditional For Any (** $\exists_c$ **)** operators. These novel operators are naturally well suited to answer queries with an idea of "candidate elements and requirements" to be performed with custom predicates. The first one is the generalization of the relational division, which allows us to answer queries when all requirements are needed. In contrast, the second one will allow us to answer queries when at least one requirement is needed. For example, they are potentially useful to support products' imports, detect allergen products, filter candidates for job promotions, and filter candidates for internships. We semantically validate our proposals by studying the first two of these applications. Also, we propose scalable algorithms to support our novel operators.

A.1 Introduction

The Relational algebra(CODD, 1972; CODD, 1990) is a set of operations performed on relations such as: Union ($\cup$), Intersection ($\cap$), Difference ($-$), Selection (σ), Projection (π), Join ($\bowtie$), Cartesian Product ($\times$) and Division ($\div$). One of these operations, the relational division is defined for convenience for dealing with queries that involve universal quantification or the **"for all"** condition. Most Relational DataBase Management Systems (RDBMS) do not directly

implement division, and still, the relational division is required in real-world applications. For example, it is useful to answer the next queries:

- "*What students approved **all** courses required for an internship?*"
- "*What employees have **all** skills for a job promotion?*"
- "*What food products have **all** ingredients of a list in their composition?*"
- "*Which suppliers can send **all** requested products in just one shipment?*"

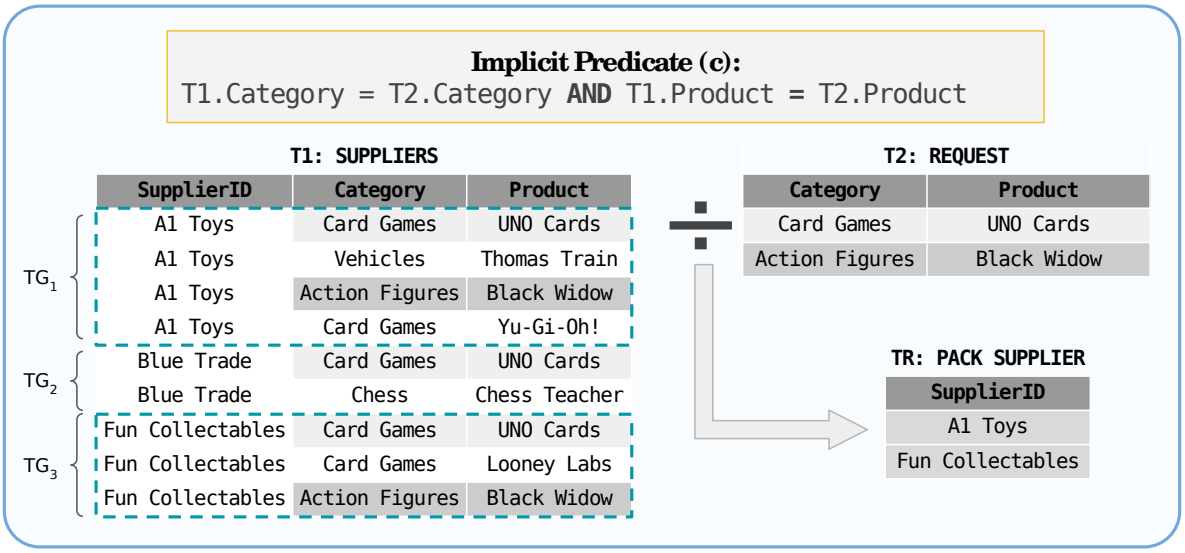


Figure 21 – Example of relational division. Querying all suppliers who can send all requested products in a single shipment.

In order to understand better the relational division, let's focus on the last example. Suppose that a client wants to import a list of products urgently and he is willing to pay for an express shipment. Then, all the required products should come in just one shipment. Given a list of suppliers with all the products they offer, the relational division can answer the query: *Q1: which suppliers can send **all** requested products in just one shipment?* This example is presented in Figure 21. Here, the dividend (T_1) is the list of suppliers and their products, having as attributes: the supplier identifier, the product category, and the product name. Also, the dividend is split into different products sets, grouped by the supplier, in our example, the products from the stores "A1 Toys", "Blue Trade", and "Fun Collectables". The divisor (T_2) is composed by the list of requirements, also containing category and name of each product. Finally, each group that contains all divisor requirements is part of the result or quotient (T_R). In our example, the result is formed by the stores that can deliver all desired products. Note that the traditional division uses an implicit identity predicate to compare the tuples between dividend and divisor. This is, tuples from dividend and divisor must be identical in all pairs of comparison attributes' values. This predicate is also illustrated in Figure 21.

Now, let's expand our example by adding other two attributes to both relations: units and price. This is illustrated in Figure 22. In T_1 these attributes refer to each store's available units and tag price. In T_2 they refer to the client's desired units and maximum affordable price. Additionally, let us assume that the client can accept fewer units than the desired ones if the seller has a low price, let's say it is at most half of the client's budget. Now, to compare tuples by identity would not be useful. We need to employ a custom predicate to match suppliers' products with the client's preferences, i.e., having enough units and acceptable price, or simply low price. The predicate c is also shown in Figure 22.

Remember that we need to answer the query "Q1: which suppliers can send **all** requested products in just one shipment?". However, we need to compare tuples with a custom predicate that the relational division doesn't support. Thus, it could be answered with the **Relational Conditional For All** ($\forall_c$) operator. This new operator is a generalization of the relational division with a new feature to support custom predicates. Also, this new operator loses the property of being "the opposite operation of the Cartesian Product". Thus, we also need to adapt the concepts of dividend, divisor and quotient, and rename them as "candidates", "requirements", and "result" relations respectively. Here, the candidates' relation (T_1) needs to be split into groups, being grouped by those attributes that are not part of the predicate. In our example from Figure 22, T_1 is grouped by supplier. Then, if all requirements' relation (T_2) tuples satisfy the predicate to any tuple of one group, this group is part of the result. In our example, each supplier who has all required products with enough units and affordable price, or low price, is part of the result.

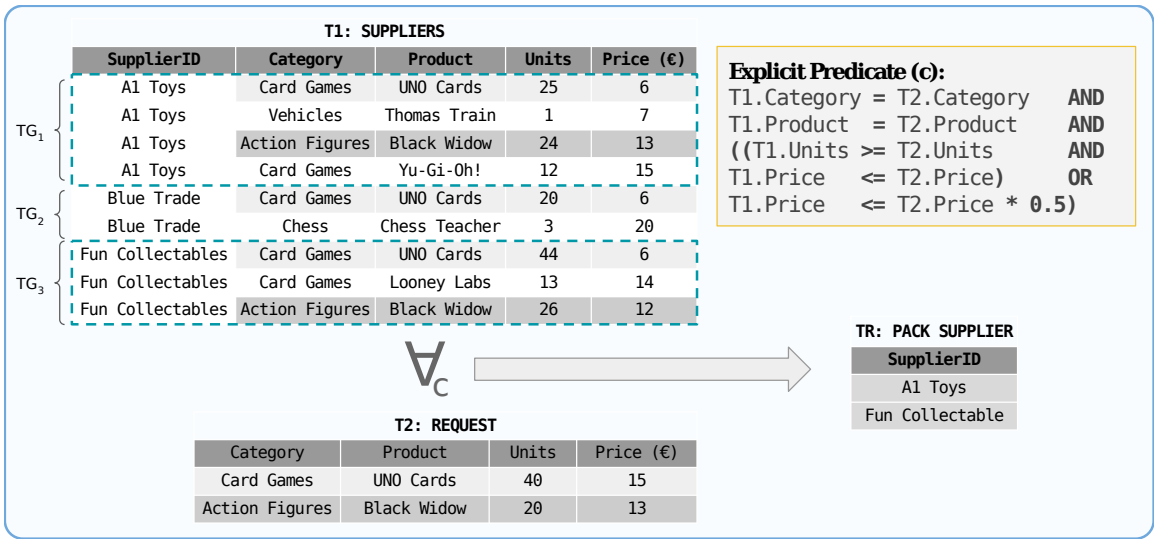


Figure 22 – Example of relational conditional For All operation: Suppliers which can send **all** requested products in just one ship, "having enough" units and price "up to" the client's budget, or a low price for that product.

Now, let's assume that the client doesn't need the products with urgency, and consequently, he can receive the products from different suppliers. This case is too similar to the previous one, same data and same predicate, but instead of answering the "for all" query, we

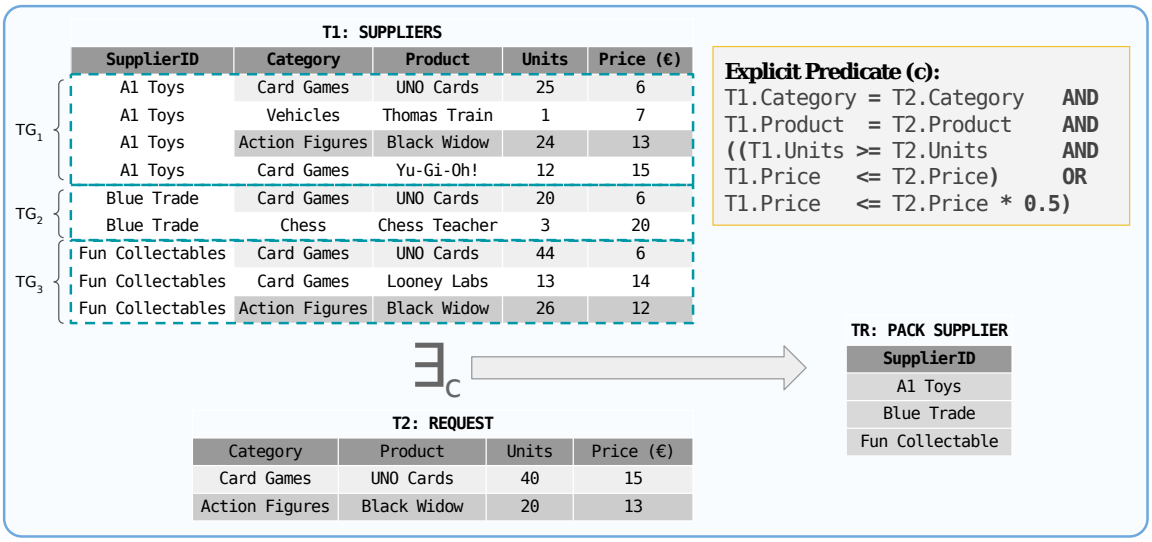


Figure 23 – Example of relational conditional For Any operation: Suppliers which can send **any** require product, "having enough" units and price "up to" the client's budget, or a low price for that product.

need to answer a *"for any"* query. This new operator will allow us to answer the query: "Q2: which suppliers can send **any** desired product?". Let's call this a **Relational Conditional For Any** ($\exists_c$) operation. Here, if any requirement's relation (T_2) tuple satisfy the predicate to any tuple of one group in the candidates' relation (T_1), this group is part of the result. This case is illustrated in Figure 23. In our example, each supplier who has any required product with enough units and affordable price, or low price, is part of the result.

As it was described before, the relational division is well suited to answer queries with the **"for all"** concept. However, it has severe limitations when we need to compare tuples by any predicate different than identity.

This proposal tackles the problem by presenting the new **Relational Conditional For All** ($\forall_c$) and **Relational Conditional For Any** ($\exists_c$) database operators. Note that if we use identity predicates with our For All operator, we would be performing a relational division. In general, our main contributions are:

C1 **Operator Design and Usability –** we identified severe limitations on the usability of the Relational Division to process queries of the "For All" type with custom predicates. We also notice that with a simple variation, we could answer queries of the "For Any" type. Thus, we extended these operators into a new one to tackle the problem.

Our Relational Conditional For All ($\forall_c$) and Relational Conditional For Any ($\exists_c$) operators are naturally well suited to answer queries with the idea of "candidate elements and requirements" to be performed when *all* requirements are requested or just **any** of them. Also, the predicates to compare tuples can be customized by the user and are not limited to identity.

C2 **Formal Definition and Algorithms –** we formally defined the Relational Conditional For All and For Any operators, enabling their usage in queries along with the existing algebraic operators. Also, we carefully designed novel algorithms to execute our new operators in a fast and scalable manner.

C3 **Semantic Validation –** we analyzed a case of products' importation, following our motivational example with toys, units, and price. Also, we proposed a case for the detection of allergen products. The results of these case studies corroborate the practical usability of our operators in real applications.

C4 **Generality and Usability –** we also exemplified other cases where our operators are well suitable, corroborating their general usability.

Observation: for the purpose of reproducibility, all codes, detailed results, parameter values tested and datasets studied in this chapter are freely available for download online [1].

A.2 Background

A.2.1 Relational Algebra

The Relational Algebra (CODD, 1972; CODD, 1990) is defined as a set of operations, not necessarily binary, that are performed on relations and the result is another relation. Some relational algebra operations are: the renaming operation (ρ), selection (σ), projection (π), union ($\cup$), intersection ($\cap$), difference ($-$), Cartesian product ($\times$), join ($\bowtie$), and division ($\div$). In this section, we will focus on relational division.

A.2.1.1 Relational Division ($\div$)

The relational division (CODD, 1972; CODD, 1990) is very similar to integers division (RAMAKRISHNAN; GEHRKE, 2000). In both cases, there exists a dividend, divisor, quotient, and residual. Remember that for two integers D and d, $D \div d$ gives us as quotient q, which is the largest integer such that $q \times d \leq D$. Also, the missing units for D, this is $D - q \times d$, is the residual. Very similar, when we work with relations T_1 and T_2, $\mathsf{T}_1 \div \mathsf{T}_2$ gives us as quotient T_R, which is the largest relation such that $\mathsf{T}_R \times \mathsf{T}_2 \subseteq \mathsf{T}_1$. Here, the missing tuples to T_1, this is $\mathsf{T}_1 - \mathsf{T}_R \times \mathsf{T}_2$, make up the residual.

Some considerations when performing a relational division between dividend T_1 and divisor T_2 are: (1) the lists of attributes to be compared from both relations, $L_1 \in Sch(\mathsf{T}_1)$ and $L_2 \in Sch(\mathsf{T}_2)$, can be expressed in the operation $\mathsf{T}_1[L_1 \div L_2]\mathsf{T}_2$; (2) both lists of attributes L_1 and L_2 must be union-compatible, i.e. both lists must contain same number of attributes and each pair of $L_1[i], L_2[i]$ must have the same domain; (3) if L_1 and L_2 are not explicitly expressed in the

[1] https://github.com/alivasples/ConditionalBinOps

operation $\mathsf{T}_1 \div \mathsf{T}_2$, they will be composed by the attributes with same name and domain in both relations.

The schema of the quotient T_R has the columns of T_1 that are not present in $L_1, Sch(T_R) = \overline{L_1}$, and the values it takes are the subset of $\pi_{(\overline{L_1})}(\mathsf{T}_1)$ with the greatest possible cardinality, such that $T_R \times \mathsf{T}_2 \subset \mathsf{T}_1$. Thus, T_1 is partitioned into $k \geq 0$ different groups of tuples such that each T_{G_k} group $\subset \mathsf{T}_1$ represents a candidate tuple for the quotient T_R if it manages to satisfy the conditions of $T_R \times \mathsf{T}_2 \subset \mathsf{T}_1$ and $\mathsf{T}_1 = \cup_{k=1}^{k} T_{G_k}$. Finally the residual is given by $\mathsf{T}_1 - (T_R \times \mathsf{T}_2)$. Equation A.1 defines the Relational Division operation.

$$\mathsf{T}_1[L_1 \div L_2]\mathsf{T}_2 = \pi_{(\overline{L_1})}(\mathsf{T}_1) - \pi_{(\overline{L_1})}\left(\left(\pi_{(\overline{L_1})}(\mathsf{T}_1) \times \pi_{(L_2)}(\mathsf{T}_2)\right)\right) - \mathsf{T}_1 \tag{A.1}$$

An example of the relational division was presented in Figure 21, which aims to solve the query "*Which suppliers can send **all** requested products in just one ship?*". Here, T_1 is the Suppliers relation and T_2 is the Request relation. Each group T_{G_i} is formed by tuples that have the same Seller, the attributes $[L_1 \div L_2]$ under which the division is being executed are $(Category, Product) \div (Category, Product)$ and the result is the list of suppliers that meet all the requirements.

A.2.2 Relational Conditional Set Operations

The Relational Conditional "For All" and "For Any" operations will also have as basis many of the concepts we previously defined in Chapter 4. Remember that the Relational Conditional Set Operations extend the traditional concepts of set membership ($\in$), subset ($\subseteq$), intersection ($\cap$), and difference ($-$) from the relational algebra set operations. These extensions are well suited to work with custom predicates, this is when we want to compare tuples by any user custom condition and not necessarily compare them by identity. As an example of the relational conditional set operations, let's take a look to Figure 22. Let's have as left relation the group T_{G_1}, getting only the projection of category, product, units, and price. In other words, we have a list of products from a single store. As the right relation, let's use T_2. This is the list of desired products by a client. In the end, with a conditional set membership operator ($\in_c$) we could query if any desired product can be bought at that store. With the conditional subset operator ($\subseteq_c$) we could know if all desired products can be bought at the store. The relational conditional intersection operator ($\cap_c$) would tell us all products that can be bought in the store. Finally, in contrast, the relational conditional difference operator ($-_c$) tells us the products not available in the store.

A.3 Related Work

In this section we discuss the main works related to our proposal. As we previously discussed most of these related works in Chapter 3, in this section we will only summarize the ones most related to the relational division. First, we discuss the works that focus on implementing operators able to answer queries of "for all" and "for any" in the Relational Algebra. Then, we discuss the works that generalize the relational division.

A.3.1 For All and For Any operations in the Relational Algebra

We consider the "For All" ($\forall_c$) and "For Any" ($\exists_c$) operations as binary operations that receive one relation of "candidates" and another relation of "requirements". The "For All" operation will retrieve the groups of candidates that satisfy *all* the requirements. Unlike, the "For Any" operation will retrieve the groups that satisfy *any* requirement. Also, note that the meaning of our operators is different to the relational calculus quantifiers: $\forall_c \neq \forall$ and $\exists_c \neq \exists$. This is because the quantifiers are not binary operators that receive two relations. They are declarations asserting that given a formula $f(x)$, it is true for all or any values of x. Under these concepts, in the Relational Algebra, there is already one operation that performs a an specific case of "For All" query, when we need to compare tuples by identity. This operation is the relational division (CODD, 1972), which retrieves the groups that satisfy by identity to all requirements. However, there is no operation that implements a "For Any" query. Thus, in this section, we discuss the related works to the relational division.

Most implementations in SQL do not directly implement the division as in the case of the join operator (ELMASRI; NAVATHE, 2015). Thus, some authors implement the relational division as queries in SQL (CAMPS, 2014; MATOS; GRASSER, 2002; CELKO, 2009). However, recent works (IMAMUDDIN; NAHAR; CHANDRA, 2020; GONZAGA; CORDEIRO, 2016) have demonstrated that there are more efficient ways to implement the operation. The first one (IMAMUDDIN; NAHAR; CHANDRA, 2020), implements a division algorithm in external source code, in this case, a c file. Also, the authors modify the SQLite library in order to make the engine recognize the "DIVIDE" token with the grammar-compliant syntax and produces output that matches the division operation. The second one (GONZAGA; CORDEIRO, 2016) implements the operation through a stored procedure. Also, the proposed algorithm makes use of indexes to improve the performance of the implementation. However, none of them is concerned in handling custom predicates for cases of candidates and requirements, allowing answering queries of the "for all" and the "for any" types.

A.3.2 Extensions of the relational division

An expansion of the relational division is the similarity-aware division (GONZAGA; CORDEIRO, 2017; GONZAGA; CORDEIRO, 2019), which answers the "For All" query when

the attributes to compare are not simple but complex. The authors proposed two algorithms for their operator, one based on Index structures to improve the performance of the query, and the other one is Full Table Scan, which is used when there are no indexes for the attributes.

Also, another work (VASCONCELOS; KASTER; CORDEIRO, 2018) incorporates and studies the behavior of several similarity aware division algorithms in a commercial RDBMS. The authors compare the two aforementioned algorithms for the similarity-aware division against several SQL statements that they adapted from the relational division to the similarity-aware division and test them with synthetic data. The conclusion was that for a long number of tuples or groups in the divisor, the best approach is to perform the Indexed Algorithm.

An application of the similarity-division is a system called Tender-Sims (VASCONCELOS *et al.*, 2018). This system can help with a public tendering process. The initial step of a public tendering process is to publish a Request For Tender (RFT) document outlining the agency's needs, requirements, criteria, and instructions. When the RFT is designed to acquire goods, the latter are usually grouped in lots, which can have the requirement that a bidding company should provide all the goods listed. RFTs are semi-structured documents, as their structure may vary for each specific request and agency, and product/service descriptions are usually written in natural language. Moreover, these documents are commonly organized by grouping products and/or services in lots to which a business must be able to supply all of them to qualify for the tender. Finally, Tender-Sims focuses on answering the question of selecting lots where a company has an item similar enough for *all* the required items.

We've studied different works that extended the relational division to support complex data. However, none of them pretended to answer queries of the "For All" and "For Any" types with conditions not limited to identity or similarity.

A.4 Proposal

In this section, we propose a formal definition for the novel operators, as well as the algorithms to support them.

A.4.1 Formal Definition

This section presents the formal definition of the **Relational Conditional For All** ($\forall_c$) and **Relational Conditional For Any** ($\exists_c$) database operators. These are presented in Definitions 13 and 14 respectively. Also, our new definitions use as basis the definitions we discussed in Chapter 4.

Definition 13. The **Relational Conditional For All** ($\forall_c$) operation is a binary operation represented as ${}_b\mathsf{T}_1 \ \forall_c \ {}_c\mathsf{T}_2 = {}_d\mathsf{T}_R$, in which ${}_b\mathsf{T}_1$, ${}_c\mathsf{T}_2$ and ${}_d\mathsf{T}_R$ are relations that respectively correspond to the candidates' relation, requirements' relation and the result, with c as the custom predicate.

L_1 and L_2 are respectively the lists of attributes of T_1 and ${}_c\mathsf{T}_2$ that are internally used in c. The schema of ${}_d\mathsf{T}_R$ is defined as $Sch({}_d\mathsf{T}_R) = \overline{\mathsf{L}_1} = Sch(\mathsf{T}_1) - \mathsf{L}_1$. The candidates' relation is grouped by $\overline{\mathsf{L}_1}$, each group T_{G_k} contains a key $\pi_{(\overline{\mathsf{L}_1})}\mathsf{T}_{G_k}$ and a conditional set ${}_c\mathsf{T}_{1k} = \pi_{(\mathsf{L}_1)}\mathsf{T}_{G_k}$. The instance of ${}_c\mathsf{T}_R$ is the union of the groups keys $\pi_{(\overline{\mathsf{L}_1})}\mathsf{T}_{G_k}$ where the requirements' relation is a conditional subset of ${}_c\mathsf{T}_{1k}$. Formally, the result ${}_c\mathsf{T}_R$ is defined as:

$$ {}_d\mathsf{T}_R = \bigcup_{k=1}^{\kappa} \begin{cases} \pi_{(\overline{\mathsf{L}_1})}\mathsf{T}_{G_k}, & if\ {}_c\mathsf{T}_2 \subseteq_c {}_c\mathsf{T}_{1k} \\ \varnothing, & \text{otherwise.} \end{cases} \tag{A.2} $$

Definition 14. The **Relational Conditional For Any** ($\exists_c$) operation is a binary operation represented as ${}_b\mathsf{T}_1\ \exists_c\ {}_c\mathsf{T}_2 = {}_d\mathsf{T}_R$, in which ${}_b\mathsf{T}_1$, ${}_c\mathsf{T}_2$ and ${}_d\mathsf{T}_R$ are relations that respectively correspond to the candidates' relation, requirements' relation and the result, with c as the custom predicate. L_1 and L_2 are respectively the lists of attributes of T_1 and ${}_c\mathsf{T}_2$ that are internally used in c. The schema of ${}_d\mathsf{T}_R$ is defined as $Sch({}_d\mathsf{T}_R) = \overline{\mathsf{L}_1} = Sch(\mathsf{T}_1) - \mathsf{L}_1$. The candidates' relation is grouped by $\overline{\mathsf{L}_1}$, each group T_{G_k} contains a key $\pi_{(\overline{\mathsf{L}_1})}\mathsf{T}_{G_k}$ and a conditional set ${}_c\mathsf{T}_{1k} = \pi_{(\mathsf{L}_1)}\mathsf{T}_{G_k}$. The instance of ${}_c\mathsf{T}_R$ is the union of the groups keys $\pi_{(\overline{\mathsf{L}_1})}\mathsf{T}_{G_k}$ where the result of the conditional intersection ${}_c\mathsf{T}_{1k} \cap_c {}_c\mathsf{T}_2$ is not null. Formally, the result ${}_c\mathsf{T}_R$ is defined as:

$$ {}_d\mathsf{T}_R = \bigcup_{k=1}^{\kappa} \begin{cases} \pi_{(\overline{\mathsf{L}_1})}\mathsf{T}_{G_k}, & if\ {}_c\mathsf{T}_2 \cap_c {}_c\mathsf{T}_{1k} \neq \varnothing \\ \varnothing, & \text{otherwise.} \end{cases} \tag{A.3} $$

A.4.2 Algorithms

In this section, we present the algorithm to support our new operators. Our algorithm is well suitable to support the relational conditional for all ($\exists_c$) and for any ($\forall_c$) operations. Also, our algorithm includes the option to perform an index-based approach if the candidates' relation has index structures for each attribute referenced in the predicate, or a full table scan (FTS) approach if not.

Algorithm 13 receives as parameters the candidates' relation (${}_b\mathsf{T}_1$), the requirements' relation (${}_c\mathsf{T}_2$), the candidates' relation's groups (T_G), the custom predicate c and the operation type ($op \in \{\ \forall_c\ ,\ \exists_c\ \}$). As result, it will return the list of the identifiers of the groups that satisfy all or any requirements, according to the operation type. Let's divide our algorithm into four sections: (1) the initialization in lines 1-2; (2) the index-based approach in lines 4 - 12; (3) the full table scan (FTS) approach in lines 14-20, and; (4) the finalization in lines 21-27. In the initialization, we only need to create a matrix M of boolean values. This matrix will be useful to store the requirements that the groups satisfy, each row being a group id, and each column being a requirement id. Then, in the best case, if the candidates' relation ${}_b\mathsf{T}_1$ contains index structures for all the columns referenced in the predicate, we will be taking advantage of these indexes

Algorithm 13 – RelCondForAllAndForAny($_b\mathsf{T}_1$, $_c\mathsf{T}_2$, T_G, c, op)

Input: $_b\mathsf{T}_1$: candidates' relation, $_c\mathsf{T}_2$: requirements' relation, T_G: candidates' relation groups, c: predicate, op: operator $\in \{ \exists_c , \forall_c \}$

Output: G: vector of resulting groups ids

```
# Create a matrix to mark the requirements satisfied by each group
M = |T_G| × |T_2| matrix of booleans with all values set to false;
# Algorithm based on index
if _bT_1 has index structures for all attributes used in the predicate then
    for each tuple t_j ∈ T_2 do
        # Bits vector indicating for each tuple t_i ∈ _bT_1 the result of c(t_i, t_j)
        S = IndexTupleQuery(_bT_1, t_j, c)
        # Mark as true all groups that satisfy the current requirement t_j
        for each s_i ∈ S : t_i ∈ T_{G_k} do
            if s_i = 1 then: M[k][j] = 1
        end for
    end for
# Algorithm based on Full Table Scan
else
    for each tuple t_i ∈ T_1 : t_i ∈ T_{G_k} do
        for each tuple t_j ∈ T_2 do
            if c(t_i, t_j) = True then: M[k][j] = 1
        end for
    end for
end if
# Sending all valid groups to the result
for each group T_{G_k} ∈ T_G do
    if (op = ∀_c and M[k] contains only 1's) or
       (op = ∃_c and M[k] contains at least one 1) then
        Add k to G
    end if
end for
return G
```

to speed up the queries. Here, we will execute for each requirement t_j, an index-tuple-query, which using index based-queries will return a vector of bits indicating for all tuples in $_b\mathsf{T}_1$ if they satisfy the current requirement. For each tuple marked with 1, i.e. tuple that satisfies the current requirement, we will mark in our matrix M its group id and current requirement id as 1. In contrast, if the candidates' relation does not have index structures, we will be using a full table scan approach. In other words, we will be iterating for each tuple t_i in $_b\mathsf{T}_1$, each tuple t_j in $_c\mathsf{T}_2$ and evaluating the predicate in both tuples. When the result of the predicate evaluation $c(t_i,t_j)$ is true, then we will mark in our matrix, the group of t_i and the requirement t_j with 1. In the end, in the finalization of our algorithm, we will have our matrix M fulled with 0's or 1's, indicating for each group, which requirements are satisfied. Then, for each row of the matrix, we will verify if all or any (according to the operation type) requirements are marked with 1, if yes, then the current group will be sent to the list of valid groups. Finally, the list of valid groups is returned.

A.5 Experiments

We evaluated the semantics and the usability of our proposals by studying two distinct applications: (1) Amazon Toys import, following the motivational example, and; (2) Food allergies, to find allergen products from a list. Their features are summarized in Table 8. Also, we simulated several synthetic datasets with different characteristics and run our algorithms to test the scalability of them. They are summarized in Table 7. Specifically, we performed experiments aimed at answering two main questions:

RQ1 How accurate are the relational conditional For All and For Any operations in the sense of returning what the users expect to receive?

RQ2 How effective and scalable are the algorithms that we propose?

The experiments were performed in a machine with an `Intel® Core i7` processor of $3.40GHz$ and $8GB$ of RAM. Our Algorithm 13 was implemented in `C++` with page buffer management. Library `Arboretum`[2] was used for the indexed-based queries.

Observation: for the purpose of reproducibility, all codes, detailed results, parameter values tested and datasets studied in this chapter are freely available for download online [3].

Dataset	$\|T_1\|$	$\|T_2\|$	$\|T_G\|$	Valid Groups	$\|Sch(T_2)\|$	Groups Distribution
Var $\|T_1\|$	$[1K;10K]$	10	10	-	3	Normal
Var $\|T_2\|$	$1K$	$[10;100]$	10	-	3	Normal
Var $\|T_G\|$	$1K$	10	$[10;100]$	-	3	Normal
Var Valid Groups	$1K$	10	10	$[0;100]\%$	3	Normal
Var $\|Sch(T_2)\|$	$1K$	10	10	-	$[2;20]$	Normal
Var Groups Dist	$1K$	10	10	-	3	{ Normal, Uniform, Exponential }

Table 7 – Summary of the Synthetic datasets

[2] https://bitbucket.org/gbdi/arboretum/
[3] https://github.com/alivasples/ConditionalBinOps

Dataset	$\|T_1\|$	$\|T_2\|$	$\|T_G\|$	$\|Sch(T_1)\|$	$\|Sch(T_2)\|$
Toys Import	25,457	[2;10]	3,523	7	4
Food Allergies	93,414	[2;10]	8,990	5	1

Table 8 – Summary of the real datasets

A.5.1 Amazon Toys

Aimed to answer our first research question RQ1, we validated the semantics of the new operators in a case study with suppliers and toys products. We obtained a dataset[4] of amazon toy products and preprocess it in order to give it the desired format to perform our operation. One fundamental step was to separate the suppliers' groups of each product in different lines. Each supplier had its own price for a product. Unfortunately, the units were part of the original rows and not part of the suppliers' list, unlike the prices, and we considered for our execution random units lower or equal than the originals for each product. Consequently, the processed dataset[5] contains supplier, category, product, units, and, price. This dataset is already our candidates' relation.

As it can be seen in Table 8, our dataset contains 25,457 candidates tuples split in 3,523 groups, 4 attributes for comparison and 3 additional attributes in T_1. Also, in order to test our queries of "For All" and "For Any", we've generated 100 test cases varying the number of requirements between 2 and 10 tuples. We first found all couples of suppliers with at least 4 common products, each group with the same couple received the name "pack". Then, we randomly selected one of those packs and randomly select between 2 and 10 tuples to compose the test case requirements' table. We ran the two queries for each test case, taking the averages for each test case and computing the total average and standard deviation of all test cases. In our three queries, we compared categories and products' names by identity and the other attributes according to our first case study. For our query Q1: "which suppliers can send **all** requested products in just one shipment?", we obtained 8.75 sec. of average and 3.55 sec. of standard deviation for the FTS approach; and, 151 ms. of average and 66 ms. of standard deviation for the index-based approach. For our query Q2: "which suppliers can send **any** desired product?", we also obtained 8.75 sec. of average and 3.55 sec. of standard deviation for the FTS approach; and, 161 ms. of average and 66 ms. of standard deviation for the index-based approach.

For example, one of the test cases is the one we took as example in Figures 22 and 23. Naturally, in our illustration we only shown few groups in the candidates' relation. Still, the requirements' relation is totally adapted from our real test case, having as requirements: (1) a set of "UNO Cards" from category "cards games" with a minimum stock of 40 units and a

[4] https://www.kaggle.com/PromptCloudHQ/toy-products-on-amazon

[5] https://github.com/alivasples/ConditionalBinOps/blob/master/Experiments/ForAllAndForAny/Real-AmazonToys/T1.data

maximum price of 15 euros each, and; (2) a "black widow funko" from category "figures and playsets" with a minimum stock of 20 units and maximum price of 13 euros each. For this case, our query Q1 retrieved only two groups. The first one represented by the supplier "a1 Toys", having for the first toy, only 25 units but with a price of 6.13 euros, and for the second one, 24 units and price of 12.99 euros. The second one represented by the supplier "Fun Collectables", having for the first toy, 44 units and price of 5.99 euros, and for the second one, 26 units and a price of 11.78 euros. The averages of 10 executions' runtime measurements were 3.97 sec. and 45 ms. for FTS and index-based approaches respectively.

Finally, our query Q2 retrieved the two previous groups in addition to 12 others that only satisfy one requirement. One of them was represented by the supplier "Blue Trade", which had only the first toy, 20 units, and a price of 6.29 euros. The averages of 10 executions' runtime measurements were 3.93 sec. and 45 ms. for FTS and index-based approaches respectively.

The results of the three queries are summarized in Table 9.

Suppliers	R.C. For All	R.C. For Any
"a1 Toys"	✓	✓
"Fun Collectable"	✓	✓
"Blue Trade"		✓
...	...	...

Table 9 – Toys Import queries results.

A.5.2 Food Allergies

We also obtained another dataset[6] for testing a "For Any" query. This one, exemplified in Figure 24, consisted in selecting products that are allergens. A product is considered allergen if contains an allergen component. Thus, given a list of products and its components, and a list of the components that a person is allergic to, all the allergen products are retrieved. Then, to find the allergen products we need to answer the query **Q3**: *"Which products contain **any** of the allergen components?*" For example, in the requirements' relation, we can see the allergen components are "orange peel" and "vanilla bean", and cookies contain both of them, but marmalade and chai tea only contain one each. Still, the three of them are allergen since they contain at least one allergen component.

In our dataset[7] we separated the components in different rows since the original dataset contained a list of products and components in a single row. After pre-processing it, we had for each row, a product and a component.

As it can be seen in Table 8, our dataset contain $93,414$ candidates tuples divided in $8,990$ groups, only 1 attribute for comparison and 4 additional attributes in T_1. Also, we've

[6] https://www.kaggle.com/datafiniti/food-ingredient-lists

[7] https://github.com/alivasples/ConditionalBinOps/blob/master/Experiments/ForAllAndForAny/Real-ProductsAllergies/T1.data

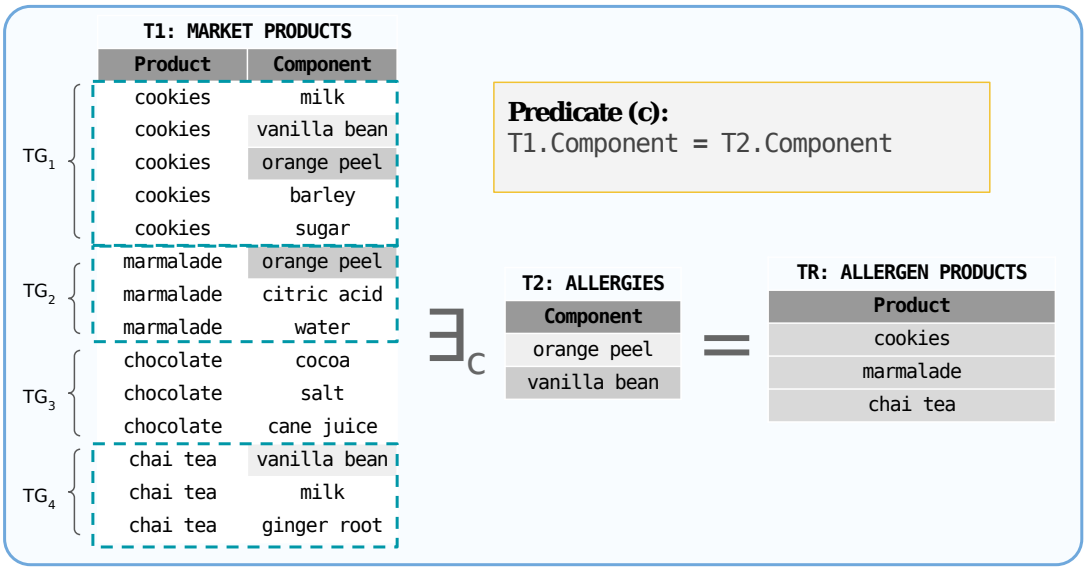


Figure 24 – Example of **For Any** relational condition. The allergen products are those which have as a component any of the allergen components.

ALLERGEN PRODUCTS
"Celestial Seasonings Sugar **Cookie** Sleigh Ride Holiday Herb Tea 20 Tea Bags"
"Polaner Sugar Free Orange **Marmalade**, 135 Oz"
"India **Chai Tea** Spice 220 oz"
...

Table 10 – Result of the query "Which products contain any allergen component?"

generated 100 test cases varying the number of requirements between 2 and 10 tuples. We ran each test 10 times and find its average, after that, we computed the average and standard deviation of all test cases. We obtained 4.94 sec. of average and 2.11 sec. of standard deviation for the FTS approach; and, 211 ms. of average and 97 ms. of standard deviation for the index-based approach.

For example, one of the test cases had as allergen products "orange peel" and "vanilla bean" and the query retrieved 42 groups that contained at least one of them in their composition. As presented in Table 10, three of these groups are the representations of "cookies", containing both allergen components, "marmalade", containing only the orange peel, and the "chai tea", containing only the vanilla bean. The other 39 groups only contain orange peel. The averages of 10 executions' runtime measurements were 1.995 sec. and 75 ms. for FTS and index-based approaches respectively.

A.5.3 Synthetic Data

In order to answer our research question RQ2: *How effective and scalable are the algorithms that we propose?*, we've generated 6 different features' families, each one varying different features in the families. Table 7 summarizes them. We defined as default the following features for all datasets: $1,000$ tuples in T_1 normally distributed in 10 groups, 10 tuples in T_2, and 3 attributes in T_2. In a general way, each family contains several datasets varying the corresponding features between the parameter limits. The predicate always take all attributes of T_2 and compares with their compatible attributes in T_1. Also, the predicate follows the form "$T_1.A = T_2.A$ *and* $(T_1.B < T_2.B$ *or* $(T_1.C > T_2.C$ *and* $(...)))$", always swapping the logical connectors between "or" and "and", and the logical operators between $=, <, >, \leq, and \geq$. Each dataset inside a family was executed 10 times for both queries, "For All" and "For Any", and for both approaches, index-based when T_1 contains index structures, and full table scan (FTS) when it does not. Then, the average of the runtime measurements was computed. Finally, for each features family, we present in

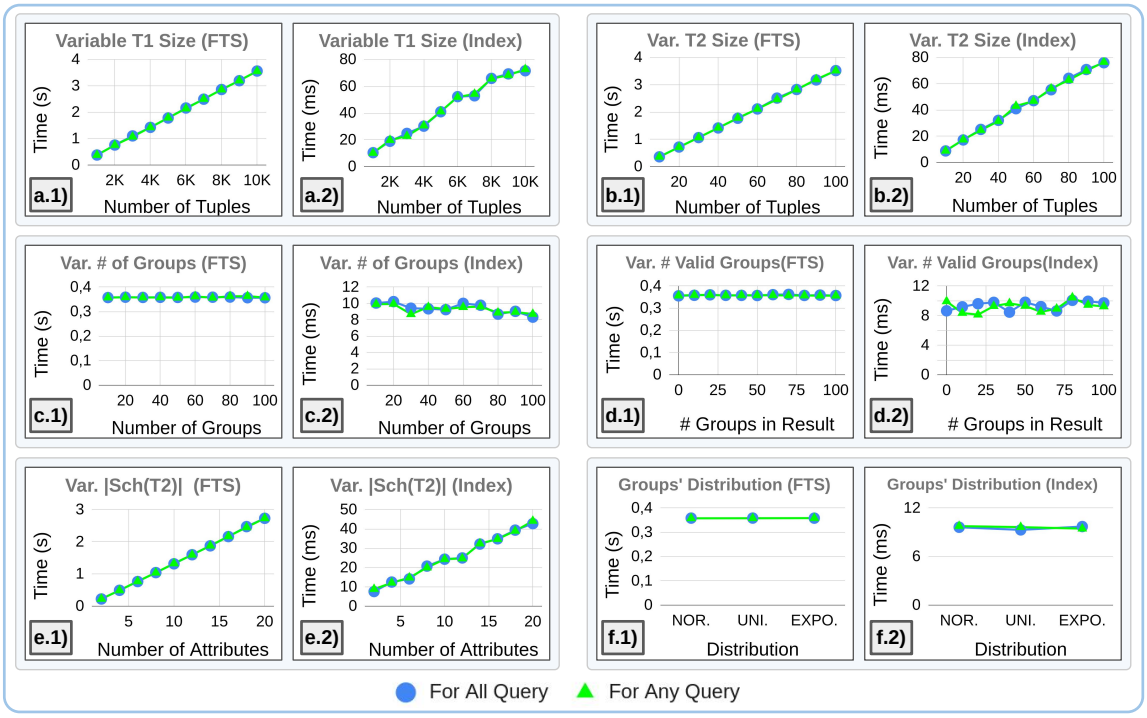


Figure 25 – Results of the Relational Condition varying distinct features in different datasets families.

Figure 25 the results of all instances measurements averages.

For the "Var $|T_1|$" Family we varied the number of tuples in the candidates' relation from 1000 to 10000. Figure 25.a.1 shows the running time for the FTS approach with both queries. Figure 25.a.2 shows the running time for the Index approach with both queries. Thus, for this case, the linear scalability seems to be present.

For the "Var $|T_2|$" Family we varied the number of tuples in the requirements' relation

from 10 to 100. Figure 25.b.1 shows the running time for the FTS approach with both queries. Figure 25.b.2 shows the running time for the index-based approach with both queries. Also here, the linear scalability seems to be present.

For the "Var $|\mathsf{T}_G|$" Family we varied the number of groups from 10 to 100. Figure 25.c.1 shows the running time for the FTS approach with both queries. Figure 25.c.2 shows the running time for the index-based approach with both queries. In both cases, it can be observed that the running time for all the cases tends to be uniform. That is, the number of groups apparently does not affect much the runtime for our algorithms.

For the "Var Valid Groups" we varied the expected valid groups in result from 0% to 100%. Figure 25.d.1 shows the running time for the FTS approach with both queries. Figure 25.d.2 shows the running time for the index-based approach with both queries. Also here, we can observe that the running time for all the cases tends to be uniform. That is, the number of valid groups in the result apparently does not affect much the runtime for our algorithms.

For the "Var |Sch(T_2)|" Family we varied the number of attributes in $Sch(\mathsf{T}_2)$ from 2 to 20. By doing this, we also varied the attributes in $|Sch(\mathsf{T}_1)| = |Sch(\mathsf{T}_2)| + 1$ and the predicate to compare all attributes by equality or inequality. Figure 25.e.1 shows the running time for the FTS approach with both queries. Figure 25.e.2 shows the running time for the index-based approach with both queries. In both cases, the linear scalability seems to be present.

Finally, for the "Var Groups Dist" Family we varied the distributions in the groups to be either exponential, uniform, or normal. Figure 25.f.1 shows the running time for the FTS approach with both queries. Figure 25.f.2 shows the running time for the index-based approach with both queries. We can observe that the running time for all the cases tends to be uniform. Thus, the groups' distribution apparently does not affect much the runtime of our algorithms.

A.6 Generality and Usability

The usability of our novel operators are not restricted only to our case study. There are several examples in real life that can use it. Basically, whenever exist queries with the idea of "candidate elements and requirements", our operator is well suited to answer them. In this section, we will show other applications.

A.6.1 Job Promotion

Let us consider a call for a job promotion supported by data from the Desired Skills (*DS*) for the job position and one candidate Employee's Skills (*ES*). In this case, *ES* and *DS* will be the candidates' relation and requirements' relation respectively. This is illustrated in Figure 26. Here, each skill can be quantified by a certification grade on a scale from 1 (beginner) to 5 (expert), with 0 for none; or, by the number of years of experience. Thus, our candidates'

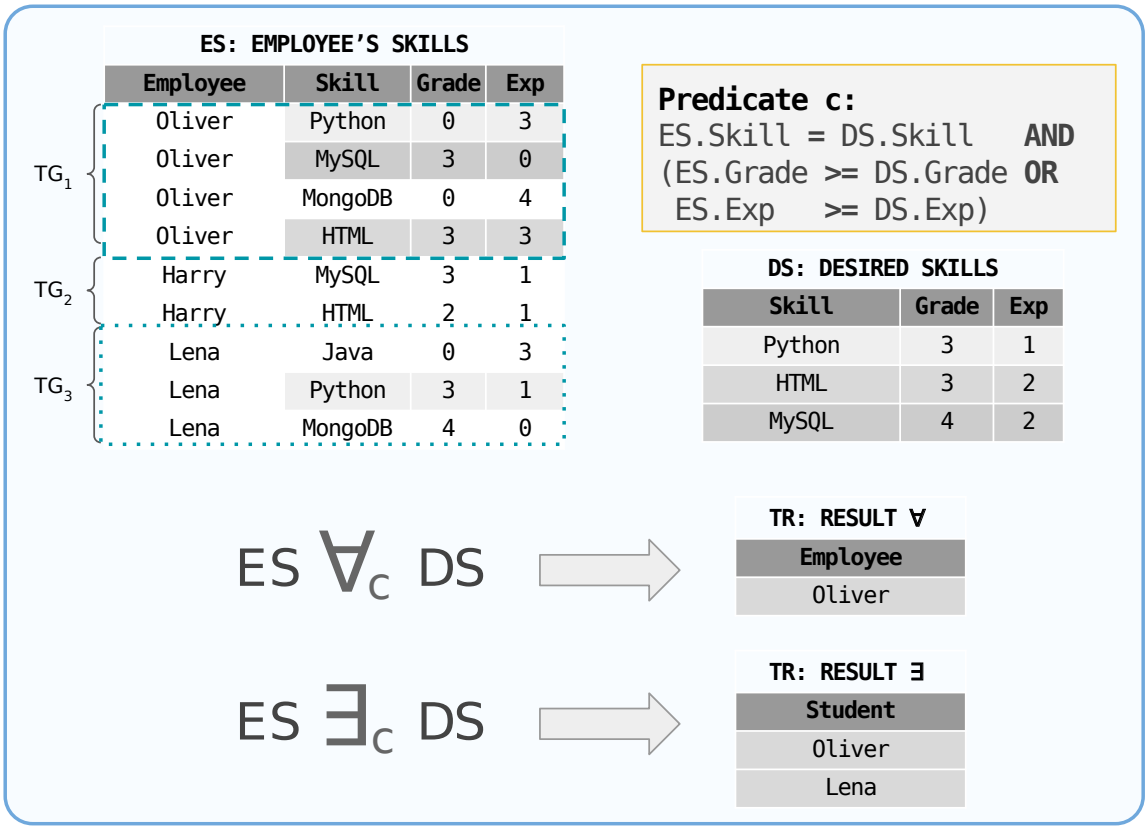


Figure 26 – Example of relational conditional For All and For Any operations: Employees that have all or any desired skills for a new job position.

relation will have the schema Sch(ES) = ($Employee$, $Skill$, $Grade$, Exp) and the requirements' relation will have Sch(DS) = ($Skill$, $Grade$, Exp). The traditional division operator would be helpless here, as it would not allow to verify if the employee has the minimal certification grade or the minimal experience for each desired skill. However, this condition can be easily treated by our new operators; we only have to consider a custom predicate $c : ES.Skill = DS.Skill \wedge (ES.Grade \geq DS.Grade \vee ES.Exp \geq DS.Exp)$. Now, with our relational conditional *For All* operation we can query: "Find all employees that satisfy all requirements" with $ES \; \forall_c \; DS$. Also, with our relational conditional *For Any* operation, we can query: "Find all employees that satisfy any requirement" with $ES \; \exists_c \; DS$.

A.6.2 Internship

For this case let us consider an organization that wants to recruit interns. Naturally, the organization would have minimum expectations about their grades in the Courses Attended in college. This is illustrated in Figure 27. Here, the organization would list the Requested Courses (RC) with minimal grades, and will receive a list of candidates (CA). In this case, CA and RC will be the candidates' relation and requirements' relation respectively. Therefore, the candidates' relation will have the schema Sch(RC) = ($Candidate$,

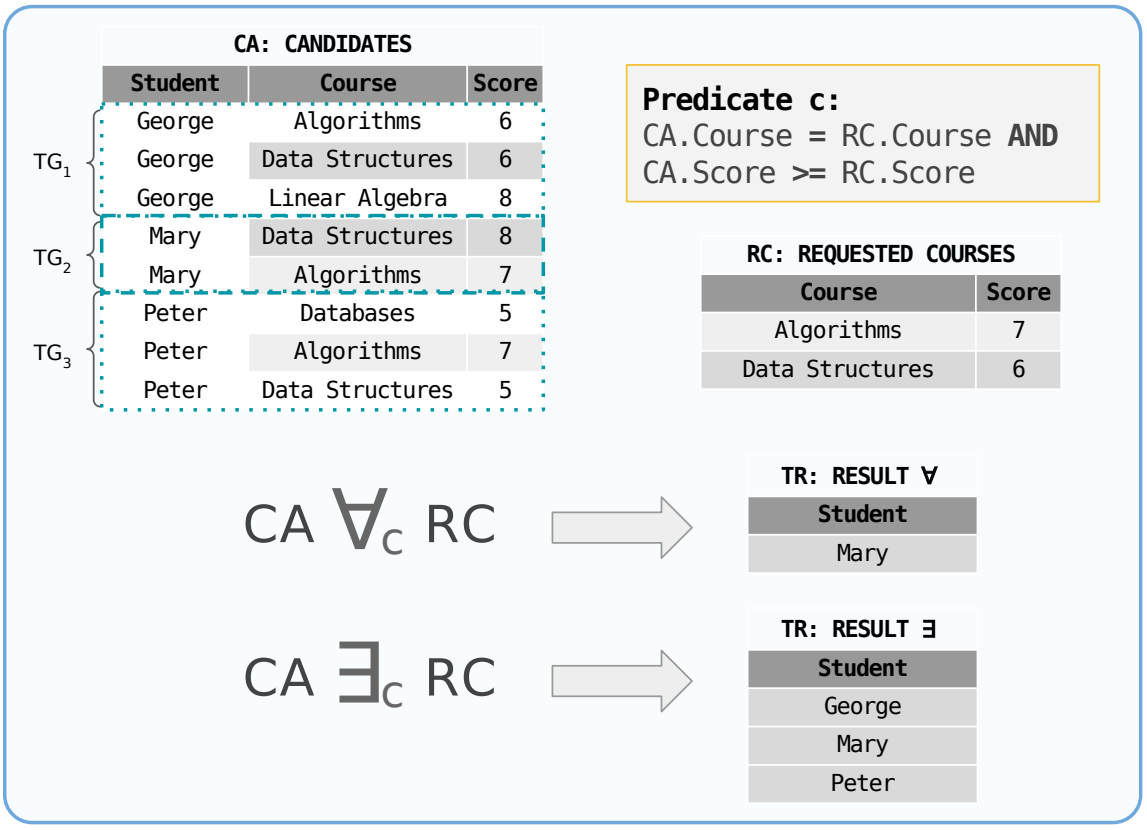


Figure 27 – Example of relational conditional For All and For Any operations: Candidates that have all or any desired grades for an internship.

Course, *Score*) and the requirements' relation Sch(*CA*) = (*Course*, *Score*). The traditional division operator would be helpless here, as they would not allow one to verify if an applicant has the minimal grades for the requested courses. Fortunatelly, our operators are promptly applicable with a predicate $c : CA.Course = RC.Course \wedge CA.Score \geq RC.Score$. Now, with our relational conditional *For All* operation we can query: "Find all candidates that passed the requested courses with enough scores" with $CA \ \forall_c \ RC$. Also, with our relational conditional *For Any* operation, we can query: "Find all candidates that passed any requested course with enough score" with $CA \ \exists_c \ RC$.

A.7 Conclusion

In this work, we identified severe limitations on the usability of the Relational Division to process queries of the "For All" type with custom predicates. We also noticed that with a simple variation, we could answer queries of the "For Any" type. Thus, we extended these operators into new ones to tackle the problem. Our main contributions were:

C1 **Operator Design and Usability –** We designed our Relational Conditional For All ($\forall_c$) and Relational Conditional For Any ($\exists_c$) operators. They are naturally well suited to answer queries with the idea of "candidate elements and requirements" to be performed

when *all* requirements are requested or just **any** of them. Also, the predicates to compare tuples can be customized by the user and are not limited to identity.

C2 **Formal Definition and Algorithms –** we formally defined the Relational Conditional For All and For Any operators, enabling their usage in queries along with the existing algebraic operators. Also, we carefully designed novel algorithms to execute our new operators in a fast and scalable manner.

C3 **Semantic Validation –** we analyzed a case of products' importation, following our motivational example with toys, units, and price. Also, we proposed a case for the detection of allergen products. The results of these case studies corroborate the practical usability of our operators in real applications.

C4 **Generality and Usability –** we also exemplified other cases where our operators are well suitable, corroborating their general usability.


ICMC
USP
SÃO CARLOS