UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

On the support of the similarity-aware division operator in a relational database management system

Guilherme Queiroz Vasconcelos

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

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Guilherme Queiroz Vasconcelos

On the support of the similarity-aware division operator in a relational database management system

Dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Robson Leonardo Ferreira Cordeiro

USP – São Carlos
June 2019

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

V958o Vasconcelos, Guilherme Queiroz
On the support of the similarity-aware division
operator in a relational database management system
/ Guilherme Queiroz Vasconcelos; orientador Robson
Leonardo Ferreira Cordeiro; coorientador Daniel dos
Santos Kaster. -- São Carlos, 2019.
89 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, 2019.

1. Relational Division. 2. Similarity Queries.
3. Relational Database Management Systems. 4. SQL.
5. Complex Data. I. Ferreira Cordeiro, Robson
Leonardo, orient. II. dos Santos Kaster, Daniel,
coorient. III. Título.

Guilherme Queiroz Vasconcelos

**Suporte à divisão por similaridade em um sistema
gerenciador de banco de dados relacional**

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
Junho de 2019

*This work is dedicated to anyone who can benefit of it
and to everyone who made this work possible.*

ACKNOWLEDGEMENTS

First and foremost I would like to thank my family for their unconditional love and for always being there for me. My parents Vivianny and Marcos and my brother Felipe who always supported me to follow my dreams. The blessings of my late great grandmother Anésia and my grandmother Elza gave me strength to go on. Thanks to my aunt Vanessa and uncle Daniel for all their motivational conversation and support. Without any of them, this work would not be possible.

I would like to thank my supervisors Professor Robson and Professor Daniel for guidance, support and patience. I am grateful for being supervised by these incredible Professors who taught me so much and encouraged me even when I doubted myself and this work. I would like to extend this thanks to Professor Elaine with whom I worked as teacher's assistance. It was a great opportunity and I learned a lot. And to Professor Marcos Bedo and my friends Daniel Mário and Guilherme Zabet who actively contributed in this research.

I am also grateful for the support of my friends. My long time friends Alan, Rafael and Walacy and the new friends I made here. Thanks Jorge, Larissa, Leonardo S., Takata, William, Carl, Daniel Mário, Eugênio, Gabriel, Guilherme Zabet, Jadson, Jaqueline, Jéssica, Jonathan, Leonardo M., Lucas, Lucas, Lucas, Mirela, Nathan, Nesso, Patrícia, P.H., Thábata and Vivian. Thanks for not letting this become dull and boring but also thanks for giving me reality check when I needed.

Finally, I would like to thank CNPq, CAPES project 10357907/M and FAPESP projects 2016/170780 and 2018/05714-5 for financial support and Pexels¹ for the pictures used in this work. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) -Finance Code 001.

¹ <http://www.pexels.com>

*“In olden days, a glimpse of stocking
Was looked on as something shocking
But now, God knows,
Anything goes
Good authors too who once knew better words
Now only use four-letter words
Writing prose
Anything goes
”*
(Anything Goes)

RESUMO

VASCONCELOS, G. Q. **Suporte à divisão por similaridade em um sistema gerenciador de banco de dados relacional**. 2019. 95 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, 2019.

O operador de Divisão ($\div$) da Álgebra Relacional permite a representação de consultas com o conceito de “para todos” de forma simples e intuitiva, e por isso, é empregado em várias aplicações do dia a dia. Entretanto, a Divisão Relacional é incapaz de atender as necessidades de aplicações modernas que manipulam dados complexos como imagens, áudios, textos longos, sequência genéticas, etc. Esses tipos de dados são melhor comparados por similaridade, porém, a Divisão Relacional sempre compara valores por igualdade. Estudos recentes focaram-se em estender a Álgebra Relacional e operadores de banco de dados para suportar comparações por similaridade. Esse projeto incorporou a Divisão Por Similaridade a um Sistema Gerenciador de Banco de Dados Relacional (SGBDR) e estudou seu relacionamento com outros operadores de consulta. Para isso, foi realizada a extensão de um SQL com operadores de similaridade para representar o operador de Divisão Por Similaridade de forma simples e intuitiva e a implementação de algoritmos do estado-da-arte, consultas internas ao banco e recursos para manipulação de dados por similaridade dentro do SGBD. Esta solução apresenta estratégias para execução eficiente de consultas envolvendo este operador. Para avaliação da qualidade de resultados, foi realizado um estudo de caso para encontrar empresas em potencial capazes de participar de licitações públicas através de comparações por similaridade dos documentos de licitação e da lista de produtos das empresas. Nós avaliamos o caso de uso com conjuntos de dados reais de licitações e empresas brasileiras da indústria alimentícia. Nos experimentos, a Divisão por Similaridade foi capaz de indentificar quais licitações cada empresa pode concorrer com uma revocação de 90%.

Palavras-chave: Divisão Relacional, Consultas por Similaridade, Sistemas Gerenciadores de Bancos de Dados Relacionais, SQL, Dados Complexos.

ABSTRACT

VASCONCELOS, G. Q. **On the support of the similarity-aware division operator in a relational database management system**. 2019. 95 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, 2019.

The Division operator ($\div$) from the Relational Algebra allows simple and intuitive representation of queries with the concept of “for all”, and thus it is required by many real applications. However, the Relational Division is unable to support the needs of modern applications that manipulate complex data, such as images, audio, long texts, genetic sequences, etc. These data are better compared for similarity, whereas the Division always compares values for equality. Recent works focused on extending the Relational Algebra and database operators to support similarity comparison. This project incorporated the Similarity-Aware Division Operator in a Relational Database Management System (RDBMS) and studied its relationship with other query operators. We extended a similarity-oriented SQL to represent the Similarity-Aware Division Operator in a simple and intuitive manner and implemented state-of-art algorithms, internal database queries and resources for similarity data manipulation all inside the RDBMS. This solution presents strategies for efficient and improved performance queries. For semantical validation, it was performed a case study of an application that finds prospective companies able to bid in public request for tenders (RFT) using similarity comparison on RFTs documents and companies’s catalogs. We evaluated the quality of results in a case study with real datasets from request for tenders from public brazilian food companies. In the experiments, the Similarity-Aware Division Operator was able to identify which RFT which company can participate in with 90% recall.

Keywords: Relational Division, Similarity Queries, Relational Database Management System, SQL, Complex Data.

LIST OF FIGURES

Figure 1 – Example of Relational Division on image contents represented as keywords.	27
Figure 2 – Example of Similarity Division on user images.	28
Figure 3 – The main contribution of this Master’s thesis	29
Figure 4 – Example of Relational Division on image contents as keywords.	32
Figure 5 – The Cartesian Product of UserIDs and Contents is in UserImages.	33
Figure 6 – The remainder is the difference of UserImages and the Cartesian product between Contents and UserIDs	33
Figure 7 – Illustration of Similarity Selection.	34
Figure 8 – Illustration of Similarity Join.	35
Figure 9 – Illustration of Similarity Set Operators.	36
Figure 10 – Illustration of the Similarity Group By.	37
Figure 11 – Illustration of the Slim-Tree.	41
Figure 12 – Illustration of the Slim-Tree’s optimization process.	41
Figure 13 – Illustration of the Onion-Tree features.	42
Figure 14 – The SIREN architecture.	47
Figure 15 – The FMI-SiR architecture.	50
Figure 16 – Our proposed method	52
Figure 17 – The Extended Similarity-Aware Division schema on images.	57
Figure 18 – Example of similarity division on image pictures.	60
Figure 19 – Pipeline of the ODCITable Interface.	65
Figure 20 – Runtime of Similarity-Aware Division Algorithms.	71
Figure 21 – Optimizing the Extended SQL Interpreter.	72
Figure 22 – Overview of our ontology.	78
Figure 23 – Illustration of the similarity comparison.	79
Figure 24 – Sequence of operations to execute Similarity-Aware Division on TendeR-Sims.	80
Figure 25 – Evaluation results.	82

LIST OF ALGORITHMS

Algorithm 1 – Similarity-aware division based on full table scan.	39
Algorithm 2 – Similarity-aware division based on index.	40
Algorithm 3 – Similarity-Aware Except	66
Algorithm 4 – Pseudo-code of the OntologySimilarity algorithm.	80
Algorithm 5 – Pseudo-code of the similarity evaluation algorithm.	81

LIST OF SOURCE CODES

Statement 1 – Division query using literal translation from Relational Algebra.	44
Statement 2 – Division query using sorting and counting.	44
Statement 3 – Division query based on double negation.	45
Statement 4 – Division based on Set operations.	45
Statement 5 – A new syntax for Relational Division by Draken, Gao and Alhajj (2001).	46
Statement 6 – Creating a metric using SIREN’s SQL.	48
Statement 7 – Creating a table and associating a metric to a complex attribute.	48
Statement 8 – User input to perform range query to select images from UserImages that are similar to image identified by ROWID 1.	48
Statement 9 – Translation from Statement 8 that is executed by the RDBMS after pro- cessing similarity operators externally.	48
Statement 10 – Example of how to associate a cluster algorithm to the attribute Image from table UserImage.	49
Statement 11 – Query using the result Cluster table to find the relevant tuples used as cluster centers.	49
Statement 12 – Query using the result Clustering table to find groups that each tuple from UserImages belongs to.	49
Statement 13 – User input to perform range query to select images from UserImages that are similar to image identified by ROWID 1.	50
Statement 14 – User input to perform range join query to join images from UserImages that are similar to each other up to a threshold of 2.	51
Statement 15 – Similarity Division query using literal translation from Extended Rela- tional Algebra.	58
Statement 16 – Similarity Division query based on the double negation.	58
Statement 17 – Similarity Division based on Set operations.	59
Statement 18 – A new syntax for Similarity Division	61
Statement 19 – Creating the Output type for the interface.	64
Statement 20 – Creating a collection of the output type.	64
Statement 21 – The interface header definition.	64
Statement 22 – The table function associated to the header	64
Statement 23 – SQL command that uses the table function.	65
Statement 24 – FMI-SiR Statement for Similarity Division query using literal translation from Extended Relational Algebra.	67

Statement 25 – FMI-SiR Statement for Similarity Division query based on the double negation.	67
Statement 26 – FMI-SiR Statement for Similarity Division based on Set operations . . .	68
Statement 27 – SQL command to invoke Similarity Division using the Full Table Scan Algorithm.	68
Statement 28 – SQL command to invoke the Similarity Division using the Indexed Algorithm.	69
Statement 29 – Statement to create a call to our ontology function.	79
Statement 30 – Statement to perform Similarity Division on Tender-Sims.	80

LIST OF SYMBOLS

π — Relational Projection

σ — Relational Selection

$\bowtie$ — Relational Join

$\div$ — Relational Division

$\cup$ — Relational Union

$\cap$ — Relational Intersection

$-$ — Relational Difference

$\times$ — Relational Cartesian Product

$\mathbb{S}$ — a data domain

$\hat{\cap}$ — Intersection by Similarity

$\hat{-}$ — Difference by Similarity

$\hat{\div}$ — Similarity-Aware Division

T_1 — A relation

L_1 — A list of attributes from T_1

t_i — The i_{th} tuple of a relation projecting all attributes

$t_i[L_1]$ — The i_{th} tuple of a relation projecting only the attributes presents in L_1 .

a_j — The j_{th} attribute of L_1

Gv_{id} — A set of integers containing id of valid groups

Gq_{id} — A temporary set of integers containing id of valid groups

T_{G_k} — Group of similar tuples from a relation

T_G — Set of groups of similar tuples from a relation

$threshold[i]$ — The i_{th} threshold of an array of similarity thresholds

ε — Similarity threshold

CONTENTS

1	INTRODUCTION	25
1.1	Problem and Motivation	26
1.2	Contributions	28
1.3	Organization	29
2	BACKGROUND	31
2.1	Relational Division	31
2.2	Similarity Queries in Metric Spaces	33
2.2.1	<i>Similarity-Aware Division</i>	38
2.3	Metric Access Methods	40
2.4	Concluding Remarks	42
3	RELATED WORK	43
3.1	Algorithms for Relational Division	43
3.1.1	<i>Plain SQL approaches</i>	43
3.1.2	<i>Other approaches</i>	45
3.2	Similarity Awareness in RDBMS	46
3.2.1	<i>SIREN</i>	47
3.2.2	<i>FMISIR</i>	50
3.2.3	<i>Other Relevant Approaches</i>	51
3.3	Discussion	52
3.4	Concluding Remarks	54
4	PROPOSED SQL STATEMENTS FOR THE SIMILARITY-AWARE DIVISION	55
4.1	SQL Statements for the Similarity-Aware Division	56
4.1.1	<i>Similarity Division using literal translation from Extended Relational Algebra</i>	57
4.1.2	<i>Similarity Division based on the double negation</i>	58
4.1.3	<i>Similarity Division based on Set operations</i>	58
4.1.4	<i>Similarity Division using sorting and counting</i>	59
4.1.5	<i>A new keyword to represent the Similarity Division in SQL</i>	60
4.2	Conclusion	61

5	ALGORITHMS TO EXECUTE THE SIMILARITY-AWARE DIVISION	63
5.1	Algorithm Implementations	63
5.2	Statement's Translation	66
5.2.1	<i>Similarity Division using literal translation from Extended Relational Algebra</i>	67
5.2.2	<i>Similarity Division based on the Double Negation</i>	67
5.2.3	<i>Similarity Division based on Set Operations</i>	68
5.2.4	<i>Similarity Division using the new keyword</i>	68
5.3	Experiments and discussion	69
5.3.1	<i>Discussion</i>	70
5.4	Conclusion	72
6	TENDER-SIMS - SIMILARITY RETRIEVAL SYSTEM FOR PUBLIC TENDERS	75
6.1	Code the System	76
6.1.1	<i>Data Acquisition</i>	77
6.1.2	<i>Building the Ontology</i>	77
6.1.3	<i>The Similarity Comparison</i>	78
6.1.4	<i>Implementation</i>	79
6.2	Experiments	82
6.3	Conclusion	83
7	CONCLUSION	85
	BIBLIOGRAPHY	89

INTRODUCTION

The Relational Database Management Systems (RDBMS) were initially developed to deal with queries using data types such as numbers and small texts. These systems manipulate data through efficient algorithms ensuring data security, redundancy control, prevention of inconsistency and application independence and so it is widely employed in information systems (ELMASRI; NAVATHE, 2010; STONEBRAKER; ROWE, 1986; DATABASE, 2010; SILBERSCHATZ; KORTH; SUDARSHAN, 2006; Kim *et al.*, 2017; WORBOYS; DUCKHAM, 2004). In these RDBMSs, data manipulation is performed following the principles of Relational Algebra (CODD, 1972), which define a number of operators to express queries on relations, including: Projection (π), Selection (σ), Join ($\bowtie$), Division ($\div$), Union ($\cup$), Intersection ($\cap$), Difference ($-$) and Cartesian Product ($\times$). Traditionally, these algebraic operators retrieve data by performing two types of comparisons: identity (using operators $=$ and $\neq$) and total order relation (using the inequality operators $<$, $\leq$, $>$, $\geq$). The later type of comparison is only applicable to ordered sets (SILBERSCHATZ; KORTH; SUDARSHAN, 2006; GANTER; WILLE, 2012). Since numbers, dates and small characters strings have a established sorting criteria, the identity and total order relation comparisons are well-defined in an RDBMS environment (CODD, 1972; KANELAKIS, 1989).

Nowadays, modern applications demand efficient manipulation of complex data such as audio, video, images, genetic sequences, long texts, etc (TRAINA *et al.*, 2003; IQBAL; ODETAYO; JAMES, 2012; TROJACANEC; DIMITROVSKI; LOSKOVSKA, 2009; OLIVEIRA *et al.*, 2016; VASCONCELOS *et al.*, 2018; CORDEIRO; FALOUTSOS; TRAINA-JR., 2013). These types of data do not present a total order relationship – For example, one cannot say that “an image precedes other one” or that “an image is larger than another” without follow-up questions such as “in what context? time? dimension?” – such ambiguity does not happen when saying that “a number is larger than one other”. Therefore comparison operators such as $<$, $\leq$, $>$, $\geq$ cannot be employed and the identity comparison may not be useful for complex data (CHÁVEZ *et al.*, 2001; SAMET, 2005; ZEZULA *et al.*, 2010). This makes data manipulation in

RDBMSs rather challenging.

A widely employed solution is to compare complex data for similarity. In similarity queries, data are usually represented by their intrinsic features and evaluated using a distance function. These queries return elements that match a similarity criteria, mostly based on the distance between the elements in a dataset and one or many query elements (or query centers). The two most common types of similarity queries are: a) the *range query*, in which a query returns all elements that are similar - or close - to the query center up to a threshold; and b) the *k-nn query*, in which a query returns the *k* nearest elements to the query center. (CHÁVEZ *et al.*, 2001; SAMET, 2005; ZEZULA *et al.*, 2010). Several researchers have been extending relational algebra to support similarity queries based on the presented approaches. Operators such as **Selection**, **Join**, **Intersect**, **Difference**, **Division**, **Group By** and **Aggregation**, as well as indexes, have all been extended to incorporate similarity comparison (ZEZULA *et al.*, 2010; JACOX; SAMET, 2008; MARRI *et al.*, 2014; SILVA; AREF; ALI, 2009; TRAINA-JR.; TRAINA, 2002; GONZAGA; CORDEIRO, 2017a).

However, despite the fact that similarity operators are well-established in the state-of-art, current RDBMSs provide very little support for them. This Master's thesis focuses on the inclusion of the new Similarity-Aware Division operator in a RDBMS. This operator proved to be useful to answer queries with the context of “for all” in real collections of complex objects coming from applications such as finding relevant requests for tenders that a company is able to satisfy by comparing their product catalog with the products required (VASCONCELOS *et al.*, 2018); identifying cities able to produce a type of crop by comparing satellite images of cities against the requirements' images (GONZAGA; CORDEIRO, 2017a); and identifying animals with all genetic sequences similar to animals that have predefined genetic traits for selective breeding, to develop high quality derivative, such as milk (GONZAGA; CORDEIRO, 2017b).

1.1 Problem and Motivation

To illustrate the Similarity-Aware Division operator's usability and motivate the importance of this work, let us consider the scenario as follows: An analyst using an image-sharing social network wants to find users that post images about a specific set of contents. Figure 1 illustrates a toy database for this query. Relation *UserImages* associates users to the content of their posted images, which is represented by keywords. Each tuple represents one content that can relate to one or many pictures of its associated user. For example, user José posted pictures about flowers, sunset and dogs. Relation *Contents* describes a set of required contents. In the example, we are looking for users that posted about both dogs and flowers. Relation *UserIDs* is the result of the Division operation; it indicates that users José and Sérgio meet the search criteria.

	(a) UserImages	$\div$	(b) Contents	$=$	(c) UserIDs
	UserID Content		Content		UserID
T_{G_1}	José Flowers		Dog		José
	José Sunset		Flowers		Sérgio
	José Dog				
T_{G_2}	Sérgio Crops				
	Sérgio Dog				
	Sérgio Flowers				
T_{G_3}	Luis Sunset				
	Luis Flowers				

Figure 1 – Example of Relational Division on image contents represented as keywords.

The Relational Division provides a simple and intuitive manner to answer this query using a single operator with traditional data. However, it cannot properly process complex data types that have been employed in nowadays applications. Therefore, the operator was extended in the literature to manipulate these types of data using similarity queries. Let us reconsider our example about users and their image categories, but now in a more realistic variation in which relation *UserImages* contains the users and the actual images they posted instead of keywords, while *Contents* is a set of sample images. Figure 2 illustrates the data. In this scenario, the query and the semantics are the same – an analyst wants to find users that post images about dogs and flowers – however, the data type has changed. Because of that, the Relational Division is unable to answer the query because it will compare data by equality ($=$), therefore the result will include only users that have posted exactly the same images as the sample set – every single pixel must be equal. Thus, the Similarity-Aware Division is the appropriate operator to answer this query. It returns users that have posted similar images, according to a given similarity criteria, such as users José and Sérgio – who, in Relational Division, would be excluded from the output set despite having pictures with similar content as what is listed in Figure 2b (dogs and flowers).

Because RDBMSs do not integrate this new operator, they are unable to attend the demands of modern applications that need to perform Division on complex data. Therefore, by extending RDBMSs to support this new operator, we extend their expressiveness power making it possible for users to find answers to a variety of queries. In this regard, this Master's thesis focuses on providing support for the Similarity-Aware Division operator, exploring the hypothesis as follows:

Hypothesis: The Similarity-Aware Division Operator can be inserted in a query processor environment of a RDBMS, thus extending its usability to deal with complex data in modern applications.

Figure 2 – Example of Similarity Division on user images.

1.2 Contributions

Our main contribution in this Master's thesis is the inclusion of the Similarity-Aware Division operator in one RDBMS providing a simple and intuitive syntax using a customized SQL language with reserved words for similarity operators and algorithms through user-defined-functions and appropriate interfaces provided by the RDBMS for internal execution of similarity operators. Figure 3 illustrates our approach: the application submits a SQL-like query using the Similarity-Aware Division; the query has its similarity-based operators pre-processed and then submitted to the RDBMS for proper execution. More details are given in the following:

1. **Incorporation to an Extended SQL:** we incorporated the Similarity-Aware Division Operator in an extended SQL with keywords for similarity operators providing intuitive syntax to represent the queries;

Figure 3 – The main contribution of this Master’s thesis: Simple and intuitive form to represent Similarity-Aware Division in SQL commands and efficient execution.

2. **Implementation of physical algorithms and SQL Statements:** we implemented the state-of-art algorithms for the Similarity Division Operator as well as all resources needed to perform this operation using SQL Statements and Similarity Operators inside an RDBMS.
3. **Performance Evaluation:** we evaluated the algorithms and Statements in order to identify properties and strategies to improve execution.
4. **Case Study - the TendeR-Sims:** we employed the proposed algorithms in an application to identify prospective clients for Requests For Tender (RFT). Our results indicate that the system is able to identify 90% of the relevant RFTs documents, and significantly reduce the manual work by 76%.

1.3 Organization

The rest of this monograph follows a traditional organization: Chapter 2 presents the background work and concepts, Chapter 3 discusses the related work, Chapters 4 and 5 detail our proposals and experiments, Chapter 6 presents a case study and Chapter 7 concludes this work.

BACKGROUND

This Chapter presents the background for this work. First, we present the Relational Division operator and important definitions that will be used during this thesis. Then, we review similarity queries and present relevant algorithms that were employed in this work.

2.1 Relational Division

The Relational Division (CODD, 1972; CODD, 1990) allows writing queries with the concept of “for all” in a simple and intuitive manner. It is the only algebraic operator that directly corresponds to the Universal Quantification ($\forall$) from Relational Calculus (CODD, 1972). Similarly to the arithmetic division, the Relational Division has a **dividend**, a **divisor**, a **quotient** and a **remainder**. For example, let us consider the following expression using integers: $21 \div 5 = 4$. In this expression, 21 is the dividend; 5 is the divisor; 4 is the quotient; and 1 is the remainder. The quotient is the greatest integer that can be multiplied by the divisor whose result is equal or less than the dividend, i.e., $4 * 5 \leq 21$; the quotient cannot be 5 in this example because the operation’s result would be greater than 21 and it cannot be 3 either, because 3 is not the greatest integer that satisfies the condition. The remainder is the difference between the dividend and the product between the quotient and the divisor, i.e., $21 - (4 * 5) = 1$.

In the Relational Division, such operations are defined between relations. The operator is defined as $T_1 [L_1 \div L_2] T_2 = T_R$ where T_1 , T_2 and T_R are relations corresponding to the **dividend**, the **divisor** and the **quotient**. L_1 and L_2 are lists of attributes of T_1 and T_2 respectively. These lists must be union compatible. For instance, let us consider the relations T_1 and T_2 defined by the schemas $Sch(T_1) = (A_1, A_2, A_3, A_4, A_5)$ and $Sch(T_2) = (A_6, A_7, A_8)$ respectively. In the division $T_1 [(A_2, A_4, A_5) \div (A_6, A_7, A_8)] T_2 = T_R$ the number of attributes listed in $L_1 = (A_2, A_4, A_5)$ from T_1 must be equal to the number of attributes listed in $L_2 = (A_6, A_7, A_8)$ from T_2 and the pairs $\langle A_2, A_6 \rangle$, $\langle A_4, A_7 \rangle$ and $\langle A_5, A_8 \rangle$ must be comparable. The quotient relation T_R is an instance of the schema $Sch(T_R) = \overline{L_1}$, which means it has all attributes from T_1 that are not present in L_1 ,

in the example, attributes (A_1, A_3) . Formally (CODD, 1972; CODD, 1990), the quotient T_R is the subset of $\pi_{(\overline{L_1})}(T_1)$ with the largest possible cardinality, such that $T_R \times T_2 \subseteq T_1$. That is, T_1 is partitioned into $\kappa \geq 0$ distinct groups of tuples so that each group $T_{G_k} \subseteq T_1$ represents one candidate tuple for T_R and the following conditions apply: $T_R \times T_2 \subseteq T_1$ and $T_1 = \bigcup_{k=1}^{\kappa} T_{G_k}$. Finally, the remainder is $T_1 - (T_R \times T_2)$. Note that in terms of Relational Division, the Cartesian Product ($\times$), the Difference ($-$) and the inclusion ($\subseteq$) operator from the Set Theory correspond to the algebraic operators of multiplication ($*$), difference ($-$) and less than or equal ($\leq$) used in arithmetic division of integers. Equation 2.1 defines the Relational Division operation.

$$T_1 [L_1 \div L_2] T_2 = \pi_{(\overline{L_1})}(T_1) - \pi_{(\overline{L_1})}\left(\left(\pi_{(\overline{L_1})}(T_1) \times \pi_{(\overline{L_2})}(T_2)\right) - T_1\right) \quad (2.1)$$

To illustrate the operator as a database operation, let us consider the example from Chapter 1 that answers the query "What Users post about a specific set of contents?". In the division $UserImages [(Content) \div (Content)] Contents = UserIDs$, $L_1 = (Content)$ and $L_2 = (Content)$ and the quotient is relation UserIDs. UserImages is partitioned based on the active domain of elements in $\overline{L_1}$, in this case, attribute UserID. In Figure 4a, the active domain of UserID is {José, Sérgio, Luis}, thus UserImages is partitioned into three disjoint groups T_{G_1} , T_{G_2} and T_{G_3} – illustrated in different shades of gray – that have tuples with **equal** values in attribute UserID. This process is called **intra-relation comparison**. The projection $\pi_{(UserID)}(T_{G_k})$ indicates a possible tuple in the quotient relation UserIDs. For each group $T_{G_k} \subseteq UserImages$, the attribute Content is compared to the attribute Content in relation Contents. This process is called **inter-relation comparison**. A group T_{G_k} belongs to the quotient if it has tuples with Content **equal** to the value of Content in **all** tuples of T_2 .

	(a) UserImages	$\hat{\div}$	(b) Contents	=	(c) UserIDs														
T_{G_1}	<table><tr><th>UserID</th><th>Content</th></tr><tr><td>José</td><td>Flowers</td></tr><tr><td>José</td><td>Sunset</td></tr><tr><td>José</td><td>Dog</td></tr></table>	UserID	Content	José	Flowers	José	Sunset	José	Dog		<table><tr><th>Content</th></tr><tr><td>Dog</td></tr><tr><td>Flowers</td></tr></table>	Content	Dog	Flowers		<table><tr><th>UserID</th></tr><tr><td>José</td></tr><tr><td>Sérgio</td></tr></table>	UserID	José	Sérgio
	UserID	Content																	
	José	Flowers																	
José	Sunset																		
José	Dog																		
Content																			
Dog																			
Flowers																			
UserID																			
José																			
Sérgio																			
T_{G_2}	<table><tr><td>Sérgio</td><td>Crops</td></tr><tr><td>Sérgio</td><td>Dog</td></tr><tr><td>Sérgio</td><td>Flowers</td></tr></table>	Sérgio	Crops	Sérgio	Dog	Sérgio	Flowers												
	Sérgio	Crops																	
	Sérgio	Dog																	
Sérgio	Flowers																		
T_{G_3}	<table><tr><td>Luis</td><td>Sunset</td></tr><tr><td>Luis</td><td>Flowers</td></tr></table>	Luis	Sunset	Luis	Flowers														
	Luis	Sunset																	
Luis	Flowers																		

Figure 4 – Example of Relational Division on image contents as keywords.

Figure 5 illustrates the equation $T_R \times T_2 \subseteq T_1$. The Cartesian product between UserIDs and Contents is a subset of UserImages. Figure 6 illustrates the remainder $T_1 - (T_R \times T_2)$. In Figure 6a, tuples 2,4,7 and 8 (from top to bottom) are not present in the quotient, therefore they are the remainder of the operation.

Figure 5 – The Cartesian Product of UserIDs and Contents is in UserImages.

Figure 6 – The remainder is the difference of UserImages and the Cartesian product between Contents and UserIDs

The Division Operator is defined for traditional data types. However, complex data are better compared for similarity. Before we discuss the Similarity-Aware Division operator, we present the main similarity operators in the following.

2.2 Similarity Queries in Metric Spaces

Similarity queries have been widely employed to manipulate complex data such as audio, video, images, genetic sequences, long texts, etc (TRAINA *et al.*, 2003; IQBAL; ODETAYO; JAMES, 2012; TROJACANEC; DIMITROVSKI; LOSKOVSKA, 2009; OLIVEIRA *et al.*, 2016; VASCONCELOS *et al.*, 2018; CORDEIRO; FALOUTSOS; TRAINA-JR., 2013). Such data do not present total order relationship so operators like $<$, $\leq$, $>$, $\geq$ cannot be used and the identity comparison ($=$) is usually meaningless as these types of data will hardly ever be equal to one another (CHÁVEZ *et al.*, 2001; SAMET, 2005; ZEZULA *et al.*, 2010). In this context, complex data are represented by their intrinsic features, i.e., images can be represented by their color, texture

Figure 7 – Illustration of Similarity Selection. (a)+ The *range query* selects elements (in light gray) that are similar to the query center (dark gray) up to a threshold. (b)+ The *k-NN query* selects elements that are the k nearest elements to the query center, in this illustration $k = 5$.

or shape patterns extracted from their visual content and compared using a distance function that evaluates how much an element is similar to another. This approach to represent data is named metric space. Formally, a metric space is a pair defined as $M = \{\mathbb{S}, d\}$ where $\mathbb{S}$ is a data domain and d is a metric that expresses the similarity between pairs of elements in $\mathbb{S}$ by a value in $\mathbb{R}^+$ where values closer to zero means more similar. Here, metric d is a distance function that satisfies the following properties: symmetry: $d(s_1, s_2) = d(s_2, s_1)$; non-negativity: $0 < d(s_1, s_2) < \infty$ iff $s_1 \neq s_2$ and $d(s_1, s_2) = 0$ iff $s_1 = s_2$; triangular inequality: $d(s_1, s_2) \leq d(s_1, s_3) + d(s_3, s_2)$ for any $s_1, s_2, s_3 \in \mathbb{S}$ (CHÁVEZ *et al.*, 2001; SAMET, 2005; ZEZULA *et al.*, 2010).

Through the metric space approach, many researchers have been extending the Relational Algebra (ADALI *et al.*, 1998; FERREIRA *et al.*, 2009; BUDÍKOVÁ; BATKO; ZEZULA, 2012; TRAINA-JR. *et al.*, 2006) and its operators, such as the **Selection** operator (ZEZULA *et al.*, 2010; SILVA *et al.*, 2013; BÖHM; BERCHTOLD; KEIM, 2001; BARIONI *et al.*, 2009), to support similarity queries based on types of queries: a) the range query, which selects all elements that are similar enough to a query center element, up to a predefined threshold; and b) k -Nearest Neighbors (k -NN) query, which returns the k -most similar elements to the query center. These two queries are illustrated in Figure 7. Extensions to these queries to deal with multiple query centers were presented in (RAZENTE *et al.*, 2008; PAPADIAS *et al.*, 2005) as **aggregate similarity queries**, in which the similarity is evaluated between each element stored in the dataset and all query centers, employing a function that aggregates distances between the element to every query center.

Other relational operators have been extended to support similarity, such as the **Similarity Join**. There are a number of variants for this operator, usually following the same principles of range and k -NN queries. The three most common Similarity Join operations are: a) *range join*, given two relations R and S , the *range join* returns a relation that pairs elements in R

Figure 8 – Illustration of Similarity Join. a) The *range join* pairs elements in Relation R (light gray) with elements in Relation S (dark gray) that are similar up to a threshold ϵ . b) The *k-NN join* pairs every element in Relation R (light gray) with its k nearest elements from Relation S (darker gray), in this case, $k = 5$. c) The *k-closest join* pairs elements in R with elements in S and returns the k most similar pairs over all, in this illustration, $k = 5$.

with elements in S that are similar up to a threshold. b) *k-NN join*, given two relations R and S, the *k-NN join* returns a relation that pairs elements in R with its k most similar elements from relation S. c) *k-closest join*, given two relations R and S, the *k-closest join* returns a relation containing the k most similar pairs overall. Figure 8 illustrates these queries.

The Similarity Join has been widely studied and many algorithms for it have been proposed. Silva *et al.* (2015) proposed the Join-Around, which combines *k-NN range* properties by pairing every element in relation R with its k nearest elements from relation S that are similar enough up to a predefined threshold. Jacox and Samet (2008) proposed a *range join* algorithm that recursively partitions data around two random elements until partitions become small enough to have their elements joined using a nested loop algorithm. Silva and Pearson (2012) provided an iterative version of this algorithm and implemented it on PostgreSQL. Index-based join algorithms have also been proposed. Chen *et al.* (2017a), Dohnal, Gennaro and Zezula (2003), Pearson and Silva (2014) and Paredes and Reyes (2009) each one with a distinct form of partitioning the metric space to prune irrelevant elements, such as ball-partitioning or hashing.

There are proposals extending the set-based operators to be similarity-aware. In these proposals the equality condition ($=$) is replaced by a set of distance function and thresholds for each corresponding attribute (MARRI *et al.*, 2016; POLA *et al.*, 2015), so that for a given pair of relations R and S, a query $R \hat{=} S$ returns tuples from R that are not similar to any tuple in S considering all attributes. Similarly, $R \hat{\cap} S$ returns tuples from R that are similar to any tuple in S considering all attributes. Figure 9 illustrates these two operators.

Marri *et al.* (2016) and Marri *et al.* (2014) provided efficient similarity-aware algorithms for the **Intersect** and **Difference** operators and implemented them on PostgreSQL. The algorithms take two sorted relations and use the first attribute as a filter to indicate possible matches. By ordering elements, they assume that for two given relations R and S, if the difference from

Figure 9 – Illustration of Similarity Set Operators. a) The basis of both set operators is to find tuples in R (light gray) that are similar to tuples in S (dark gray) up to a threshold. b) The Similarity Intersect will return elements in R (in black) with similar elements in S . c) The Similarity Except will return elements in R (in black) with **no** similar element in S .

tuple element R_i in R to a tuple element S_j in S is greater than the similarity threshold, then the difference from R_{i+1} to S_j is also greater than the similarity threshold. Then, they use a Mark and Restore mechanism that iterates through both tables and marks tuples that may satisfy the similarity condition and completely discards tuples that do not satisfy it, avoiding the quadratic complexity of nested loops. [Pola et al. \(2015\)](#) and [Pola et al. \(2013\)](#) presented the concept of **SimSets**, which are sets with no pair of elements that are similar to each other up to a threshold. A $\text{SimSet-}\epsilon$ from a relation R is a subset of R that does not have any element such that R_i is similar to R_j up to ϵ . To identify a SimSet of threshold ϵ , the elements from R are represented in a graph in which elements are nodes. If a pair of nodes is similar up to ϵ , then they are connected. The next step removes iteratively elements according to the number of edges, resulting in a $\text{SimSet-}\epsilon$. Then, set operations can be applied between distinct $\text{SimSet-}\epsilon$ using their proposed algorithm. The algorithm unites two $\text{SimSet-}\epsilon$ of S and R resulting in a graph in which nodes of R are linked to nodes of S and nodes of R are not linked together. Then, for each node in a $\text{SimSet-}\epsilon$ of R that has a link to a node in $\text{SimSet-}\epsilon$ of S , the algorithm removes the node and its linked elements and returns $S \hat{-} R$ (if it is a Difference Operation) or removes nodes with no links and returns $S \hat{\cap} R$ (if it is an Intersect Operation). These proposals present two limitations: the first approach is limited to unidimensional data only and the second approach does not allow a table to have near duplicate elements considering a threshold.

The **Group By** and **Aggregation** operators have been extended to assign tuples to groups based on a similarity condition and perform aggregation functions on them. [Silva, Aref and Ali \(2009\)](#) defined three classes of Group By: 1) Unsupervised Similarity Group By, in which data are grouped according to the `MAXIMUM_ELEMENT_SEPARATION` – that is, elements similar up to a threshold belong to the same group; and/or the `MAXIMUM_GROUP_DIAMETER` – that is, the diameter of the group must not exceed an assigned value; 2) Supervised Similarity Group Around groups elements around a set of points. Each tuple is assigned to the group of its closest point. This operator also takes the `MAXIMUM_ELEMENT_SEPARATION` and the

Figure 10 – Illustration of the Similarity Group By. a.) The Similarity Group By Any assigns new elements to groups if the new element is similar to any element in the group. b.) The Similarity Group By All assigns new elements to groups if the new element is similar to all elements in the group. c.) The Similarity Group By Vote assigns new elements to groups based on an election criteria, i.e, the distance from elements already assigned in groups. In the picture, element d would be assigned to group e-h because it is closer to element e than to element a. d.) The Unsupervised Similarity Group By assigns elements to groups if they are similar up to a threshold or does not exceed a predefined diameter. e.) The Supervised Similarity Group Around assigns elements to groups according to the distance of center elements. f.) The Supervised Similarity Group By Using Delimiters groups elements based on a set of delimiting points.

MAXIMUM_GROUP_DIAMETER as parameters; 3) Supervised Similarity Group By Using Delimiters groups elements based on a set of points to delimit the space. These points can be directly assigned or obtained from table through a SQL Statement. For instance, consider an uni-dimensional set of elements ranging from 1 to 10 and a set of delimiters 3,6,10. In this case, three groups will be formed: elements from 1 to 3 will belong to the first group; elements from 4 to 6 will belong to a second group; and elements from 7 to 10 will belong to a third group. Tang et al. (2016) proposed the Similarity Group-By All that returns groups of elements whose pairwise distance between every element in the group is less than or equal to a threshold; and the Similarity Group By-Any, that creates groups in which one element must be similar to at least another element in the group. To deal with overlapping elements, the clause ON OVERLAP takes the following parameters: JOIN-ANY to assign element randomly to any overlapping groups; ELIMINATE to discard overlapping elements; and FORM-NEW-GROUP to create new groups for

overlapping objects. Laverde *et al.* (2017) propose the Similarity Group By Vote, a grouping algorithm for metric space, and use an election criteria based on weighted votes to choose the most appropriate group for an object counting the votes from objects that satisfy the grouping constraint. The work also extended the algorithms to group elements using k -NN predicate instead of only the *range* predicate. These six classes of Similarity Group By are illustrated in figure 10.

Finally, the **Similarity-Aware Division** operator was proposed by Gonzaga and Cordeiro (2017b), who demonstrated its suitability to process complex data by answering queries with the concept of “for all”. This operator is the focus of this Master’s thesis, therefore we thoroughly describe it in the next session.

2.2.1 Similarity-Aware Division

Remember that the Relational Division is defined as $T_1 [L_1 \div L_2] T_2 = T_R$ where T_1 , T_2 and T_R are relations corresponding to the **dividend**, the **divisor**, and the **quotient**, and L_1 and L_2 are lists of attributes of T_1 and T_2 respectively to be compared with each other. The Relational Division only performs comparison by **equality** ($=$). As stated in subsection 2.2, complex data are better compared for similarity. Thus, the work of Gonzaga and Cordeiro (2019) extends the intrinsic comparisons performed by the Division operator, namely the **inter-relation** and **intra-relation** comparisons, to accept a threshold of dissimilarity. Thus, the operator relaxes the comparison allowing similarity-based for both **inter-relation** and **intra-relation** comparisons.

To illustrate, let us consider again the query "What Users post about a specific set of contents?" from the running example, in which relation *UserImage* associates an UserID to an image Content, represented as a feature vector, as shown in Figure 2. The similarity “**intra-relation comparison**” will partition the dividend relation into groups based on the active domain of L_1 , so that $T_{G_k} \subseteq T_1$. In Figure 2a, the operator will divide the relation based on the active domain of attribute UserID, however, if the elements are of complex type, they will be grouped by an appropriate Similarity Group By algorithm. Considering that Similarity Group By is a stand out research field on its own and that grouping elements depends heavily on the problem’s semantic, domain rules, data type and many other factors, this process is abstracted from the Similarity-Aware Division as a parameter table T_G , alias **grouping relation**, that relates each tuple in the dividend with its respective groups. A basic schema for T_G is $Sch(T_G) = (objectID, groupID)$. As for the similarity “**inter-relation comparison**”, it compares attributes in L_1 and L_2 by means of a set of distance functions and thresholds. Finally, the division operator verifies which groups in T_G have tuples similar to **all** tuples in relation *Contents*, that is, every group in T_G such that for every tuple in the divisor, the group has at least one tuple that is similar to it. The groups that satisfy this condition are added to relation *UserIDs*.

To perform the Similarity-Aware Division there are two proposed algorithms

(GONZAGA; CORDEIRO, 2019). One of them is executed when the attributes in L_1 are indexed; the other one is based on a full table scan. For the remainder of this monograph, we refer to the non-indexed algorithm as **Full Table Scan** (Algorithm 1) and to the indexed one as **Indexed Algorithm** (Algorithm 2). Both algorithms take the grouping relation T_G as parameter and save the information pair $\langle \text{UserID}, \text{groupID} \rangle$ in a hashtable. The Full Table Scan creates an in-memory metric access method for the divisor relation (line 3). Initially, it considers all groups invalid (line 5). Then, each tuple in the dividend is compared with every tuple in the divisor and the result is stored in a temporary matrix $\text{GroupsXRequirements}$ (lines 8-13). After evaluating every pair of tuples, the algorithm runs over the matrix looking for lines with only positive values, that is, groups that satisfy all requirements. These groups are added to the resulting relation (lines 15-18). The algorithm returns the resulting relation (line 19). In the Indexed Algorithm, initially all groups are valid (line 3), then, every tuple in the divisor relation is used as a query center on the indexed dividend relation (line 4). After every search, the array of valid groups is updated accordingly to the result set of the search (lines 8-10). This process is repeated until all tuples in the divisor have been processed and every unqualified group is removed (line 12). In line 36, the algorithm returns the resulting relation containing groups that were not previously removed in line 12.

Algorithm 1 – Similarity-aware division based on full table scan.

FTS_SimilarityDivision(T_1, L_1, T_2, L_2, T_G)

Result: IDs of the valid groups, i.e., the selected candidates.

```

1 begin
2   // indexes  $T_2$  to speed-up execution
3   Builds in-memory metric indexes for  $T_2$ ;
4   // IDs of valid groups and matrix of groups vs. requirements
5    $G_{vid} = \emptyset$ ;
6   // Matrix of booleans with all values set to false;
7    $M = T_G \times T_2$ 
8   // verifies the requirements
9   foreach tuple  $t_i \in T_1$  do
10    // finds every tuple  $t_j \in T_2$  such that  $t_j[L_2]$  is similar to  $t_i[L_1]$ 
11     $T = \text{IndexRangeQuery}(T_2, L_2, t_i[L_1])$ ;
12    // sets true for the appropriate values in  $M$ 
13     $M_{(k,j)} = \text{true}, \forall j, k : t_j \in T \wedge t_i \in T_{G_k}$ ;
14  end
15  // spots the valid groups
16  foreach group  $T_{G_k} \in T_G$  do
17    if there is no false value in line  $k$  of matrix  $M$  then
18       $G_{vid} = G_{vid} \cup \{k\}$ ;
19    end
20  end
21  return  $G_{vid}$ 
22 end

```

Algorithm 2 – Similarity-aware division based on index.

$\text{Index_SimilarityDivision}(\mathbb{T}_1, L_1, \mathbb{T}_2, L_2, T_G)$

Result: A relation containing IDs of the valid groups, i.e., the selected candidates.

```

1 begin
2   // all groups are valid at the beginning
3    $G_{vid} = T_G$ ;
4   foreach tuple  $t_j \in \mathbb{T}_2$  do
5     // finds every tuple  $t_i \in \mathbb{T}_1$  such that  $t_i[L_1]$  is similar to  $t_j$ 
6      $\mathbb{T} = \text{IndexRangeQuery}(\mathbb{T}_1, L_1, t_j)$ ;
7     // spots IDs of the groups that satisfy requirement  $t_j$ 
8      $G_{qid} = \emptyset$ ;
9     foreach tuple  $t_i \in \mathbb{T}$  do
10      |  $G_{qid} = G_{qid} \cup t_i.\text{groupIDs}$ ;
11    end
12    // updates valid groups
13     $G_{vid} = G_{vid} \cap G_{qid}$ ;
14  end
15  return  $G_{vid}$ ;
16 end

```

2.3 Metric Access Methods

Processing similarity queries is costly. Therefore, a number of metric access methods (MAMs) exist (CIACCIA; PATELLA; ZEZULA, 1997; DOHNAL *et al.*, 2003; DOHNAL; GENNARO; ZEZULA, 2003; CHEN *et al.*, 2017a; FILHO *et al.*, 2001; CHÁVEZ; NAVARRO, 2005; MICO; ONCINA; CARRASCO, 1996) to partition the metric space and discard irrelevant elements, thus reducing processing time, distance calculations and disk access. This Master's thesis implements the Indexed Algorithm using the Slim-Tree index and the Full Table Scan using the Onion-Tree index. Therefore, in this section we describe these indexes.

Tree-based methods organize the structure in an hierarchical fashion where the elements are stored in leaf nodes while index nodes store routing elements (called pivots) that accurately represent the subspace they point to. During the execution of a query, the query element is compared to the pivots from the root to the leaves, pruning elements considered too distant (or too dissimilar) through triangular inequality (CHEN *et al.*, 2017b; SAMET, 2005; CHÁVEZ *et al.*, 2001; BÖHM; BERCHTOLD; KEIM, 2001). There are disk-based and memory-based MAMs. A classical example of a disk-based MAM is the M-Tree (CIACCIA; PATELLA; ZEZULA, 1997) and the Slim-Tree (TRAINA-JR. *et al.*, 2000).

The Slim-Tree (TRAINA-JR. *et al.*, 2000) is a dynamic balanced tree with bottom-up construction, as illustrated in Figure 11. The idea is to divide the space in predefined maximum capacity regions or nodes. When a node overflows, the algorithm splits the node into two and redistributes the elements according to their distances to two pivots (or representative elements), which are elected by a given node splitting policy. This process is recursively performed up to

Figure 11 – Visual representation of a Slim-Tree and its logic structure. In this picture, the maximum capacity of a region is three and S1 to S17 represents database elements. This image was adapted from (KASTER, 2012).

Figure 12 – Illustration of the Slim-Tree's optimization process. The FatFactor quantifies the overlapping degree between nodes. a) The quantification of overlapping between nodes. b) The optimization algorithm "Slim-Down" replaces the element in red to a neighbor node and reduces the radius of the leftmost node without increasing the radius of the rightmost one, which might potentially prevent queries in the rightmost radius from accessing the leftmost region.

the root node. When the root node splits, the tree grows by one level. The structure provides a post-construction algorithm, the *Slim-Down*, that reduces the overlapping degree of pairs of nodes by reassigning elements from one node to another. For example, in Figure 12a, the element in red was originally placed in the leftmost node. In Figure 12b, the *Slim-Down* moved the element in red to the rightmost one. As a result, the leftmost node's radius is reduced, which might potentially prevent queries in the rightmost radius from accessing the leftmost region.

Regarding in-memory MAMs, the Onion-Tree (CARÉLO *et al.*, 2009) is an efficient option. Its nodes are built around two pivots, initially creating four disjoint partitions. Then, each new element is inserted in a node partition according to its distance to each of the two pivots. Since this procedure may lead to unbalanced node partitions, this structure proposes

a non-quadratic partitioning method that increases the number of partitions on the node to a value greater than four, building a shallower and wider structure without additional distance calculations. During insertion, the structure also verifies if data can be better partitioned by replacing one of the pivots with the new element, which would improve query processing. If it is the case, the new element replaces one of the previous pivots and the elements are redistributed according to the new region boundaries. The method also has an algorithm to perform (k -NN) queries that efficiently visits subspace partitions in a specific order. Figure 13a illustrates the expansion procedure of a Onion-Tree node, showing how the partitioning procedure reduces the most external subspace into smaller partitions. Figure 13b is another alternative to avoid elements being placed in the most external subpartition: during insertion, if the new element reduces the number of elements in the most external subpartition, it replaces one of the pivots. Figure 13c finds the k nearest elements of query element S_q visiting the nodes in the following order: 7, 5, 6, 4, 2, 1 and 3.

Figure 13 – Illustration of the Onion-Tree features. a) The expansion procedure partitions the space in more regions and provides a more shallow structure. b) The pivot replacement during insertion of elements redistributes the node regions and improves query processing. c) The visiting order for k -NN queries. Adapted from (CARÉLO *et al.*, 2009).

2.4 Concluding Remarks

This chapter presented the background concepts used as the basis of this MSc work. In this chapter, we presented the algebraic properties of the Relational Division and showed that the operator is the appropriate way to represent queries with the concept of “for all”. We also showed that in order to execute this operation with complex data, it is necessary to extend the comparisons performed by the division operator. The approach used by the state-of-art work is to perform similarity comparisons, which has been widely employed to deal with data that do not present a total order relation. However, so far, this database operator is not actually implemented in any Relational Database Management System. Before we present our solution to implement this operator in a RDBMS, we must present state-of-art solutions to implement other similarity operators in an RDBMS and discuss on how these solutions relate to our work.

RELATED WORK

This chapter presents the related work on the Relational Division and on the inclusion of similarity operators and resources in RDBMS. In this chapter, the schemas for all Statements is the same from our running example "What Users post about a specific set of contents?", illustrated in Table 4. Remember that the Schema is defined as $Sch(UserImages) = (UserID, Content)$ and $Sch(Contents) = (Content)$, where UserImages is the **dividend** and Contents is the **divisor**.

3.1 Algorithms for Relational Division

The Structured Query Language (SQL) (ELMASRI; NAVATHE, 2010) is a very useful tool to handle structured data in Relational Database Management Systems. It allows the user to manipulate data and retrieve information using Relational Algebra's operators. The language implements all independent operators such as Selection (σ), Union ($\cup$), Difference ($-$), Cartesian Product ($\times$) and some dependent operators such as Join ($\bowtie$) and Intersection ($\cap$), but it does not present a simple way to write queries using Relational Division, and there is no reserved word like there are for Join and Intersection. However, one can represent the Relational Division using a combination of Relational Algebra and other database operators such as WHERE EXISTS. Several works (CELKO, 2009; CAMPS, 2014; MATOS; GRASSER, 2001; GONZAGA; CORDEIRO, 2016) focus on writing SQL Statements for Relational Division using Relational Operators combined with other RDBMS's resources. This section discusses the most relevant ones.

3.1.1 Plain SQL approaches

Division query using literal translation from Relational Algebra: The most basic representation of Relational Division in SQL is the literal translation from Relational Algebra, i.e., Equation 2.1 illustrated in Statement 1. It uses operators such as Cartesian Product ($\times$) and

Except (–) to obtain the quotient by subtracting groups (UserID) from UserImages that does not have a required Content in Contents.

Statement 1 – Division query using literal translation from Relational Algebra.

```
(SELECT DISTINCT userID FROM UserImages)
EXCEPT
(
  SELECT userID FROM
  (
    (SELECT UserImages.userID,Contents.content FROM UserImages,Contents)
    EXCEPT
    (SELECT userID,content FROM UserImages)
  )
);
```

Division query using sorting and counting: The works of [Matos and Grasser \(2001\)](#) and [Gonzaga and Cordeiro \(2016\)](#) perform experimental evaluations on several Statements to retrieve data using Relational Division and their results show that sorting and counting approach is the most efficient provided there are no duplicate tuples in the dividend relation and the divisor relation is not empty. Given two relations R and S, where R is the dividend and S is the divisor, this approach creates groups in R over attributes that are not compared to attributes in S. Then, it joins R and S and checks which groups in R matched all tuples in S. To illustrate this approach, Statement 2 creates groups based on the attribute UserID. Then it joins by equality UserImages.content with Content.content. Finally it checks, for each group (or UserID), if the number of joined tuples is equal to the cardinality of relation Contents; if so, then the UserID is qualified as valid.

Statement 2 – Division query using sorting and counting.

```
SELECT userID FROM UserImages U
JOIN Contents C ON U.content = C.content
GROUP BY userID
HAVING COUNT(*) = (SELECT COUNT(*) FROM Contents);
```

Division query based on the double negation: The ‘‘Double Negation’’ approach verifies which groups do not miss any requirement from the divisor relation. This verification is performed by the nested query, that selects tuples from the divisor that does not exist in a particular group. If this query returns an empty relation, it means that does not exist a missing requirement and the group is qualified as a valid group, i.e, it has all the requirements. In Statement 3, the innermost query checks which Content is missing from every distinct UserID in UserImages. If there are no missing Content, then the UserID is qualified as valid.

Statement 3 – Division query based on double negation.

```
SELECT DISTINCT userID FROM UserImages uGlobal
WHERE NOT EXISTS
(
    SELECT content FROM Contents C
    WHERE NOT EXISTS
    (
        SELECT userID FROM UserImages uLocal
        WHERE uLocal.userID = uGlobal.userID AND uLocal.content = C.content
    )
);
```

Division based on Set operations: The ‘ ‘Subset’ ’ approach is very intuitive: for every group in the dividend relation, it checks if their set of values is a superset of the values in the divisor relation. In Statement 4, the innermost query uses the Except algorithm to verify this condition, in which the difference must be empty, meaning that all values in Contents are present in the set of the userID ’s values.

Statement 4 – Division based on Set operations.

```
SELECT userIDs FROM (SELECT DISTINCT userID userIDs FROM UserImages)
WHERE NOT EXISTS
(
    (SELECT content FROM Contents C)
    EXCEPT
    (SELECT content FROM UserImages WHERE userID = userIDs)
);
```

3.1.2 Other approaches

The work of [Gonzaga and Cordeiro \(2017a\)](#) proposes an PL/SQL algorithm to execute the Relational Division. The algorithm performed as well as the best division query presented in the literature (Statement 2). According to the authors, this is because of the nature of the procedural language and that an implementation inside the RDBMS’s core would provide a better performance. In the work of [Draken, Gao and Alhajj \(2001\)](#), a Java parser uses a keyword DIVIDE that translates to Statement 2 makes it possible to write division Statements in a very simple manner, as illustrated in Statement 5.

Statement 5 – A new syntax for Relational Division by [Draken, Gao and Alhaji \(2001\)](#).

(`SELECT` user,content `FROM` UserImages)

`DIVIDE`

(`SELECT` content `FROM` Contents)

Statements 1 to 5 are employed in SQL to execute the Relational Division. In order to execute the Similarity-Aware Division, one must adapt them to compare attributes for similarity. The next section will discuss strategies to represent similarity operators in SQL and how they can be executed. Such strategies will serve as guidelines to our proposed SQL-like representation and execution of the Similarity-Aware Division operator.

3.2 Similarity Awareness in RDBMS

The current Relational Database Management Systems do not offer a complete support to complex data. A simple alternative provided is to represent complex data as BLOB - *Binary Large Object*, however, this approach is limited to basic CREATE, READ, UPDATE and DELETE operations.

Fortunately, researchers have been proposing solutions to perform similarity queries inside RDBMS by extending Relational Algebra to include similarity operators such as range and k -NN. In this regard, the works of [Atnafu, Brunie and Kosch \(2001\)](#) and [Atnafu et al. \(2004\)](#) present equivalence rules for Similarity Selection and Similarity Join. [Traina-Jr. et al. \(2006\)](#) explore *range* and k -NN operators using simple and composed predicates with one or multiple query centers, identifying equivalence rules for composed predicates and query optimization through query rewriting. The works of [Ferreira et al. \(2011\)](#) show that range predicates have equivalent properties to the Relational Selection and that k -NN has limited equivalence rules, thus having limited moving options in the query plan. They show, however, that k -NN based on set inclusion properties are useful to delay k -NN predicates in the query plan. [Silva, Aref and Ali \(2009\)](#) and [Silva et al. \(2015\)](#) proposed algebraic rules for Similarity Group By and Similarity Join with useful algorithms implemented in a prototype inside RDBMS core ([SILVA et al., 2010](#)). The works of [Budíková, Batko and Zezula \(2012\)](#) and [Barioni et al. \(2009\)](#) present formal and elaborated extensions to SQL grammar to incorporate similarity operators along with other relational operators.

These proposals incorporate similarity operators in Relational Algebra, providing important guidelines for alternative query plans and efficient execution. They follow one of these approaches: a) extension of the RDBMS's source code, b) user-defined functions and c) external execution of similarity algorithms.

This section reviews literature proposals on the incorporation of similarity operators and algorithms in RDBMS. The two most relevant proposals related to this work are: SIREN ([BARI-](#)

Figure 14 – The SIREN architecture: An external API that intercepts the query submitted to the RDBMS to perform similarity operators and rewrite the query using standard SQL.

ONI *et al.*, 2006) and FMI-SIR (KASTER *et al.*, 2010). The next two subsections describe them in detail. Subsection 3.2.3 discusses other approaches that could not be used in this work due to intrinsic limitations.

3.2.1 SIREN

SIREN (*Similarity Retrieval Engine*) (BARIONI *et al.*, 2009; BARIONI *et al.*, 2006) is a mechanism that works in between the application layer and the RDBMS. Through its similarity-oriented SQL, queries using similarity operators are written similarly to queries using standard SQL. In level of abstraction, the knowledge of implementation details and restrictions are removed from the application programmer’s responsibility. Figure 14 illustrates this solution: it is a middleware working in between the application layer and the database layer providing a query interpreter from its extended SQL to standard database SQL as well as feature extractors, distance functions and metric access methods. After processing similarity predicates, the query is rewritten in standard SQL and processed by the RDBMS. Here we illustrate how similarity queries operate in this middleware. Throughout this section, all words from the extended SQL are highlighted in red.

To perform similarity queries over complex data such as images, the user must create a METRIC (a combination of a distance function and feature extractors) and associate it to the complex attribute, defined by the new type MONOLITHIC. This is done through extended DDL commands illustrated in Statements 6 and 7. The first command creates a metric named EDIST that uses a feature extractor histogram and the euclidean distance function (LP2). The second command creates table UserImages and associates the new metric to the attribute Content; this attribute stores an image as a feature vector generated by feature extractor ‘Histogram’. DML commands were extended too. For instance, let us consider relation UserImages from our running example where a feature vector is stored in attribute Content. A *range selection* over UserImages is illustrated in Statement 8.

Statement 6 – Creating a metric using SIREN’s SQL.

```
CREATE METRIC EDIST
USING LP2
FOR MONOLITHIC (histogram (histogramc as hist 2);
```

Statement 7 – Creating a table and associating a metric to a complex attribute.

```
CREATE TABLE UserImages (
userID VARCHAR(255),
content MONOLITHIC METRIC USING EDIST
);
```

Statement 8 – User input to perform range query to select images from UserImages that are similar to image identified by ROWID 1.

```
SELECT userID,content
FROM UserImages
WHERE content
NEAR
(SELECT content FROM UserImages WHERE ROWID = 1) BY EDIST
RANGE 2;
```

The query in Statement 8 selects images (Content) from UserImages that are similar to a particular image in the same relation, identified by ROWID = 1. It employs the metric access method Slim-Tree to speed up execution. Note that the similarity predicate is represented in the form <tuple-attribute> similarity operator <constant> similarity parameter, a close representation to relational predicates: <tuple-attribute> operator <constant>. During query execution, SIREN’s query interpreter accesses its metadata dictionary to perform semantic analysis, such as, to check if there is a metric associated and/or an index associated to the attribute. Then, the framework performs similarity operators and returns ROWIDs of elements that satisfy the similarity predicates – in our example, elements that are similar to tuple of ROWID 1 up to a threshold of 2 distance units. Finally, the query is rewritten using Standard SQL and the returned ROWIDs are inserted in a WHERE IN clause, as illustrated in Statement 9.

Statement 9 – Translation from Statement 8 that is executed by the RDBMS after processing similarity operators externally.

```
SELECT user,images
FROM UserImages
WHERE ROWID IN (2, 3, 6);
```

As well as Similarity Selection, SIREN offers support to Similarity Join. Much

like Similarity Selection, Similarity Join operations are performed externally and the result set is inserted in a WHERE IN clause. Another support offered by SIREN is the generalized specification of cluster operations through *table functions* – an approach that benefits from SQL’s ability to specify functions in the FROM clause that returns data in table format which can be manipulated using Relational and Similarity operators and predicates in the WHERE clause. In this scenario, a clustering algorithm over a complex attribute is invoked through and the result is stored in a pair of relations that can be manipulated by SQL commands. This process is illustrated in Statements 10 to 12. As shown in Statement 10, a set of parameters defines the clustering algorithm’s method, metrics and number of groups for partition algorithms. The result are the pair of relations Cluster – that stores meta-information regarding the groups – and Clustering – that relates tuples to their groups. Statements 10 to 12 show how to use Cluster and Clustering tables. The query in Statement 11 selects the most representative images and their respective groups. The query in Statement 12 selects all the images and their groups. Tables Cluster(UserImage.content) and Clustering(UserImage.content) in Statements 11 and 12 represent an intuitive form to reference groups and attributes in this extended SQL. During query processing, these references are translated to SIREN’s internal tables.

Statement 10 – Example of how to associate a cluster algorithm to the attribute Image from table UserImage.

```
SET CLUSTERING METHOD = 'CLARANS', METRIC='EDIST', K=5 ON UserImages.Content
```

Statement 11 – Query using the result Cluster table to find the relevant tuples used as cluster centers.

```
SELECT userID, content FROM
UserImages, Cluster(UserImage.content) ImageCluster
WHERE User.content = ImageCluster.CenterID;
```

Statement 12 – Query using the result Clustering table to find groups that each tuple from UserImages belongs to.

```
SELECT userID, ImageCluster.ClusterLabel
FROM UserImage, Clustering(UserImage.content) ImageCluster
WHERE user.content = ImageCluster.ObjectID;
```

SIREN does not support the dynamic inclusion of new feature extractors and distance functions. Therefore, [Bedo, Traina and Traina Jr. \(2014\)](#) proposed an extension to this middleware named SimbA (Similarity By Authorities). It extended SIREN’s meta-data dictionary to offer dynamic inclusion of data types, distance functions, metric access methods and feature extractors being more flexible to the domain rules.

Figure 15 – The FMI-SiR architecture. During the execution of a query that uses similarity predicates, the query processor changes context from SQL to the module, performs the algorithms and returns information to the RDBMS.

3.2.2 FMISIR

FMI-SiR (KASTER *et al.*, 2010) is a similarity retrieval module implemented as a shared library that uses Oracle’s extensible interfaces to perform similarity operations. The module offers feature extractors, distance functions and metric access methods. As illustrated in Figure 15, during the execution of a query that uses similarity predicates, the query processor changes the context from SQL to the module, calls the external functions that execute the algorithms and return information to the RDBMS. The module allows the integration of similarity operators with relational operators and can invoke similarity operators anywhere in the query plan, thus allowing to write alternative query plans for queries involving similarity operators.

FMI-SiR performs Similarity Selection and Similarity Join using full table scan or indexed accesses, using the Slim-Tree metric access method. Similarity data manipulation begins with feature extraction that generates a feature vector and stores it in a BLOB. Then, similarity comparisons are performed using User Defined Functions (UDFs) that implement distance functions. For instance, let us consider relation UserImages from our running example where an image’s feature vector is stored in attribute Content. A *range selection* over UserImages is illustrated in Statement 13. The query selects images from UserImages that are similar to the image identified by ROWID = 1 up to a threshold of 2 distance units. Note that this is the same query performed by SIREN in Statement 8 but representing similarity operators in UDF with a more complex syntax.

Statement 13 – User input to perform range query to select images from UserImages that are similar to image identified by ROWID 1.

```
SELECT user,content
FROM UserImages A
WHERE euclidean_dist(A.content,
(SELECT content FROM UserImages WHERE ROWID = 1)) <= 2;
```

If there is an index over the image attribute Content, this query performs index lookup to speed-up execution; otherwise, it performs a full table scan. The same approach happens when performing *range join* presented in Statement 14.

Statement 14 – User input to perform range join query to join images from UserImages that are similar to each other up to a threshold of 2.

```
SELECT * FROM
UserImages A JOIN UserImages B
ON
euclidean_dist(A.content, B.content) <= 2;
```

FMI-SiR does not support the dynamic inclusion of feature extractors and distance functions. It is necessary extend the source code to implement new resources and recompile it. It is also necessary to run DDL commands to associate the new resource to its implementation inside the module.

3.2.3 Other Relevant Approaches

simDB (SILVA *et al.*, 2010) is an extension to PostgreSQL to perform Similarity Group By and Similarity Join as physical operators with query optimization using equivalence rules. The solution also presents an extended SQL to write queries using similarity operators. Because it is a source-code extension, it is able to perform query optimizations providing fast execution, however, it is sensible to version control: code must be reincorporated into PostgreSQL whenever there is a new update for it.

PostgreSQL-IE *PostgreSQL with Image-handling Extension* (GULIATO *et al.*, 2009) implements Similarity Selection via user-defined functions in C and PL/pgSQL. There are functions for data definition, data manipulation and similarity retrieval and also to the dynamic inclusion of feature extractors. It does not support any other operator besides Similarity Selection and it does not include metric access methods in its current implementation, although it offers support for integrating the later.

Kiara (OLIVEIRA *et al.*, 2016) uses an extended SQL that serves as a proxy to the user-defined functions that performs feature extraction, distance comparisons, Similarity Selection and Similarity Join. Through the extended SQL it is also possible to associate new distance functions and feature extractors without recompilation. When Kiara receives a query that uses similarity operators, it executes the appropriated algorithms and then rewrites the query using standard SQL operators that are submitted to PostgreSQL query processor. However, while the later executes the similarity algorithms in an external framework, Kiara handles all data manipulation inside the RDBMS, allowing multiple query plans involving traditional and similarity predicates altogether. RAFIKI (*Retrieval-based Application for Imaging and Knowledge Investigation*) (NESSO-JR *et al.*, 2018) extends Kiara to perform Similarity Selection using a metric acces method, namely the Slim-Tree.

MSQL (LU *et al.*, 2017) answers similarity queries using standard SQL only. Distance functions are implemented using user-defined functions. The solution stores pre-computed

Figure 16 – Our proposed approach integrates SIREN’s extended SQL and interpreter for simple query specification with FMI-SiR’s similarity resources processed inside the RDBMS.

distances from complex attribute to an associated pivot element in a new column using a bi-dimensional data type, in the format $\langle \text{pivotid}, \text{distance} \rangle$. Similarity Selection and Similarity Join use this column to prune objects that are not result candidates by evaluating if the column value falls into the upper bound and lower bound limits of the distance between the query element and the pivot. Only tuples whose column values fall between the limits have their associated feature vectors computed using a distance function. It is possible to evaluate the boundaries via index by indexing the bi-dimensional data type using B⁺-Tree. The solution also presents new heuristics for pivot selection and pruning rules.

3.3 Discussion

Our proposal for the implementation of the Similarity-Aware Division as a database operator follows the workflow of Figure 16:

1. The user writes the query in a simple and intuitive manner;
2. The query interpreter parses the query operators;
3. The query optimizer performs optimizations (if any), and;
4. The executor runs similarity comparisons.

To write queries using the Similarity-Aware Division (step 1), one approach is to adapt the queries presented in Section 3.1 replacing Relational Operators and comparisons to their similarity counterpart – e.g in Statement 3, the comparison between attributes Content, originally by equality ($=$) must be replaced by a distance function and a threshold; in Statement 4, the Relational Except operator must be replaced by a Similarity-Aware Except. A second approach is to implement the algorithms presented in Section 2.2.1 and invoke them in SQL, which can be using *table functions*, however, this approach demands the support to metric access methods. Furthermore, the RDBMS must support group elements in L_1 for similarity and referencing these groups in the Statements.

In order to replace the Relational Operators and comparisons in SQL, similarity operators must be included in the language (step 2). Section 3.2 presented two options: to extend the

SQL language by means of an intermediate parser; or to represent these operators through user-defined-functions in the WHERE clause. In terms of execution (steps 3 and 4), similarity operations can be: executed externally or inside the RDBMS through user-defined-functions and appropriate extensible interfaces.

Thus, to support the Similarity-Aware Division operator, the solutions must have: 1) Distance Functions to compare data for similarity; 2) A Similarity-Aware Except Algorithm to execute a similarity version of the queries present in Statements 1 and 4 - which will be further explained in the next Chapter; 3) Metric access methods to implement the state-of-art algorithms and useful to speed execution of queries; 4) Group By/Clustering Algorithms for Similarity to partition the dividend table in groups during the intra-relation comparison when attributes in $\overline{L_1}$ are of complex type; and 5) Multiple Query Plans are needed when combining Similarity Operators with Relational Operators to write a Similarity-Aware Division query. As the executor must be able to create different query plans and choose the best one to provide more efficient execution. And also, to be able to explore and study the Similarity-Aware Division in relation with other Similarity and Relational Operator in the query plan of more complex queries. 6) Extended SQL, because it is also of our interest to represent similarity operators in SQL queries in a simple manner.

Table 1 summarizes the related works discussed in this Chapter in regards to the needs of our proposal. simDB is a solution for similarity comparisons in multidimensional data and in our work, similarity comparisons are represented in metric space. PostgreSQL-IE and Kiara allow similarity comparison in metric spaces on Postgres but the former is limited to images only and the later does not provide a metric access method. While this Master's thesis illustrates examples of similarity queries using images, our solution is to all types of data that can be represented on metric space. A metric access method is important to make feasible the implementation of the state-of-art algorithms. RAFIKI is an improvement of Kiara that provides a metric access method but only for Similarity Selection. MSQ performs similarity comparison using a B⁺-Tree and not a metric access method. In terms of relevant features for this work, SIREN, and its improvement, SimbA, are equal. Their only difference is that the later allows the inclusion of distance functions and feature extractors without the need of recompiling the source code. They only lack the ability to create alternate query plans and a Similarity-Aware Except algorithm, but no solution provides the later either. So SIREN and SimbA have enough features to support the inclusion of the Similarity-Aware Division operator, but since they do not provide multiple query plans, they hinder the exploration of properties of the Similarity-Aware Division in the query plan and its relation with others similarity and algebraic operators, which is an interesting future work. While FMI-SIR provides this feature, it lacks an Extended SQL and clustering algorithms.

No approach supports our needs fully and none of them provides a Similarity-Aware Except operator. However, SIREN and FMI-SIR complement each other. Thus, we change

	Distance Function	Extended SQL	Alternate Query Plan	Metric Access Method	Similarity-Aware Except	Similarity Group By / Clustering
PostgreSQL-IE	✓					
MSQL	✓		✓			
SIREN	✓	✓		✓		✓
SimbA	✓	✓		✓		✓
Kiara	✓	✓	✓			
FMI-SiR	✓		✓	✓		
simDB			✓			✓
RAFIKI	✓	✓	✓	✓		

Table 1 – State-of-the-art solutions for similarity-awareness in RDBMS and our requirements to implement the similarity division.

SIREN’s similarity query processor - which is done externally - to be performed on FMI-SiR, which is tied the Oracle’s query processor, providing metric access methods and alternate query plans while still being able to represent queries in an Extended SQL and providing algorithms to group elements by similarity. So in our workflow, illustrated by Figure 16, steps 1-2 are performed by SIREN and steps 3-4 are performed by FMI-SiR on Oracle. We will supply the need of a Similarity-Aware Except by implementing an algorithm using the idea of *table functions*. Further details are presented in the following Chapters.

3.4 Concluding Remarks

This chapter presented the main related works regarding our proposal. First, we discussed how to write queries using Relational Division in SQL. There are many ways to do it, each one with its own query plan, time performance and memory usage. Then, we presented literature implementations of similarity-related resources in RDBMS. According to Table 1, none of them fits our need completely, but two of them can be combined to suit our work: SIREN’s extended SQL and and FMI-SiR’s similarity retrieval resources inside Oracle query processor. The next two chapters detail how we combined these two solutions to implement the Similarity-Aware Division on Oracle.

PROPOSED SQL STATEMENTS FOR THE SIMILARITY-AWARE DIVISION

In Subsection 2.2.1 we presented the Similarity-Aware Division, which is a new operator to handle complex data using the concept of “for all”. It is an extension to Relational Division that relaxes the equality comparison performed by the operator, namely **inter-relation** and **intra-relation** comparisons, to accept as valid candidate groups with **similar values to all** values required. This chapter proposes SQL-like Statements to represent these comparisons for similarity.

In Chapter 3, we presented multiple Statements that specify the Relational Division in RDBMS. All comparisons performed by these Statements are done by equality (=). However, in order to perform Similarity-Aware Division, the RDBMS must support similarity operators on SQL commands. Typically, similarity operators are represented as user-defined functions to be evaluated in the WHERE clause or by extending the Database Management System’s SQL language to include these operators. We chose the later to write queries using Similarity-Aware Division because it allows similarity predicates to be represented in a very similar manner to traditional predicates. It fits a principle defined earlier in this Master’s thesis that SQL’s syntax should not get more complex just because the attribute’s data types changed. This is clear observing Statement 8 and 13: while the former only requeries the actual parameters of a range query, the later requires knowledge of user-defined functions name, header, parameters orders and expected return value.

To represent Similarity-Aware Division in SQL, we studied the division’s SQL statements from the works of [Matos and Grasser \(2001\)](#), [Gonzaga and Cordeiro \(2016\)](#) and [Draken, Gao and Alhajj \(2001\)](#), and carefully designed strategies to create similarity-aware versions of them. This Chapter proposes Statements for the user to write to express the Similarity-Aware Division. Our basis is the similarity-oriented SQL from [Barioni et al. \(2009\)](#), which offers support to Similarity Selection, Similarity Join and Similarity Group By.

We extended the language to handle all operators necessary to perform our Statements, such as the Similarity Except Operator.

4.1 SQL Statements for the Similarity-Aware Division

To provide the similarity **intra-relation comparison**, we consider that the primary key of the dividend (T_1) is always a single-column integer attribute. This premise works for all cases since all tables can be generalized to this specification by creating an artificial primary key. For instance, let us consider our running example from Chapter 1 in Table 1; the dividend table's schema $Sch(UserImages) = (UserID, Content)$ is modified to $Sch(UserImages) = (ID, UserID, Content)$. We also consider that the grouping relation (T_G) has at least two attributes: the group identifier and the dividend's primary key, such as in the following schema: $Sch(T_G) = (ClusterID, ObjectID)$ where $ClusterID$ identifies the group and $ObjectID$ is a foreign key from $UserImages$. The similarity **intra-relation comparison** is provided by joining the dividend relation with its grouping relation using the equality of $T_G.ObjectID$ and $UserImages.ID$ as the predicate and then projecting the feature vectors along with the group identifier.

Therefore, in our proposed syntax, we represent the Similarity-Aware Division as the equation $\boxed{T_U [L_1 \hat{\div} L_2] T_2 = T_R}$ where $T_U = \pi_{(ClusterID, L_1)} \left(T_1 \begin{matrix} (T_1.UserID = T_G.ObjectID) \\ \bowtie \end{matrix} T_G \right)$. SIREN's support of Similarity Group By by means of Cluster and Clustering tables, as in Statement 12, is employed here to represent the grouping relation. This section presents SQL-like commands to perform Similarity-Aware Division. We consider our running example query "What Users post about a specific set of contents?". The extended schema $\boxed{T_U [L_1 \hat{\div} L_2] T_2 = T_R}$ is illustrated in Figure 17.

To represent similarity **inter-relation comparison**, Statement 16 performs range queries by replacing the equality operator ($=$) with the reserved words NEAR and RANGE. Statements 15 and 17 employ the Similarity Except operator. To write queries using the Similarity Except operator in SIREN's SQL, we propose a new keyword "RANGE EXCEPT" and incorporate this new keyword in SIREN's grammar. The grammar rule presented as follows obeys the already established Relational Set Operators's syntax. The word $\langle query \rangle$ is always a relation. The words $\langle numeric_values \rangle$ and $\langle metric_names \rangle$ provide the similarity threshold and the metric associated to each attribute, respectively. Examples of usage are illustrated in Statements 15 and 17.

```

<similarity_except> ::= (<query>)
                        RANGE EXCEPT
                        (<query>)
                        NEAR <numeric_values> USING <metric_names>

```

Figure 17 – The Extended Similarity-Aware Division schema on images.

4.1.1 Similarity Division using literal translation from Extended Relational Algebra

Algebraically, the Relational Division is represented in Equation 2.1. A SQL Statement representation is shown in Statement 1. Using the Extended Relational Algebra with Similarity Operators, the Similarity-Aware Division is represented in Equation 4.1. A Similarity SQL Statement for Equation 4.1 is shown in Statement 15. We use the RANGE EXCEPT to perform “inter-relation comparison” and specify the metric EDIST, from Statement 6, meaning that the associated distance function is the euclidean distance and the associated feature extractor is Histogram.

$$\boxed{T_U [L_1 \hat{\div} L_2] T_2 = \pi_{(\overline{L_1})} (T_1) \hat{\div} \pi_{(\overline{L_1})} \left((\pi_{(\overline{L_1})} (T_1) \times \pi_{(L_2)} (T_2)) \hat{\div} T_1 \right)} \quad (4.1)$$

Statement 15 – Similarity Division query using literal translation from Extended Relational Algebra.

```
(SELECT DISTINCT ClusterID FROM Clustering(UserImages.userID))
EXCEPT
(
  SELECT ClusterID FROM
  (
    (SELECT ClusterID, content FROM Clustering(UserImages.userID),Contents)
    RANGE EXCEPT
    (SELECT ClusterID, content FROM UserImages
    JOIN Clustering(UserImages.userID) ON
    UserImages.userID = Clustering(UserImages.userID).objectID)
    NEAR 2 USING EDIST
  )
);
```

4.1.2 Similarity Division based on the double negation

The “Double Negation” approach (Statement 3) was adapted to perform Similarity-Aware Division resulting in Statement 16. It uses range queries to perform “**inter-relation comparisons**”.

Statement 16 – Similarity Division query based on the double negation.

```
SELECT DISTINCT ClusterID FROM Clustering(UserImages.userID) uGlobal
WHERE NOT EXISTS
(
  SELECT content FROM Contents co
  WHERE NOT EXISTS
  (
    SELECT ClusterID FROM Clustering(UserImages.userID) uLocal
    JOIN UserImages uimg ON uimg.userID = uLocal.objectID
    WHERE uLocal.clusterID = uGlobal.clusterID AND
    uimg.content NEAR co.content USING EDIST RANGE 2
  )
);
```

4.1.3 Similarity Division based on Set operations

The “Subset” approach (Statement 4) was adapted to perform Similarity-Aware Division by replacing the Relational Except by the new RANGE EXCEPT to perform similarity

“inter-relation comparison”.

Statement 17 – Similarity Division based on Set operations.

```
SELECT clusters FROM
(SELECT DISTINCT ClusterID clusters FROM Clustering(UserImage.userID))
WHERE NOT EXISTS
(
  (SELECT content FROM Contents)
  RANGE EXCEPT
  (SELECT content FROM UserImages uimg
  JOIN Clustering(UserImage.userID) cl ON uimg.userID = cl.objectID
  AND clusters = cl.clusterID)
  NEAR 2 USING EDIST
);
```

4.1.4 Similarity Division using sorting and counting

After careful consideration, we noted that the ‘‘Counting’’ approach (Statement 2) cannot be adapted to perform the Similarity Aware Division. Some issues were found when trying to adapt this query to perform similarity comparisons. This approach is based on the following question: “*for each group in the dividend, how many distinct tuples can be joined with the divisor?*”. If the number of tuples is **equal** to the cardinality of the divisor table, then it means that the group has tuples similar to all tuples in the divisor, therefore it should be in the result set. This checking is done in line 4 of Statement 2. The query works only if the dividend groups have a one-to-one relationship with the divisor relation, that is, each tuple of each group maps to at most one tuple of the divisor relation, and vice-versa.

When relations are compared for similarity, mapping can be one-to-many or many-to-many. This fails to guarantee that the checking in line 4 of Statement 2 can qualify actual valid groups. To illustrate, let us consider our running example of the query “What Users post about a specific set of contents?” shown in Figure 17. Figure 18 is the result of the JOIN operation in lines 1-3 of Statement 2. According to line 4, Group 1 is qualified because it has one tuple matching exactly one tuple in the divisor table and it matches all tuples. Group 0 also matches all tuples in the divisor table, however, in a many-to-one fashion and thus being disqualified by the checking in line 4. Finally, Group 2 is qualified by line 4 because it has two tuples joining the divisor relation, however, it matches two tuples containing pictures of flowers instead of one tuple containing picture of flowers and a second tuple containing picture of a dog. It misses one requirement (a picture of a dog), and is incorrectly outputted. Thus, this approach cannot be directly adapted to perform the Similarity-Aware Division.

(a) Clustering(UserImage.Image) $\bowtie$ UserImages
 $\bowtie$ Image

ClusterID	Image	Images_Image
0
0
0
1
1
2
2

Figure 18 – Example of similarity division on image pictures.

4.1.5 A new keyword to represent the Similarity Division in SQL

So far, all of the Statements presented in this chapter combine several operators to represent the Similarity-Aware Division. In order to make it easier the use of this operator, we present in the following an Extended SQL keyword RANGE DIVIDE with a grammar rule for it to perform a direct link to the algorithms presented in Section 2.2.1.

```

<similarity_division> ::= (<query>)
                        RANGE DIVIDE /*+ no-index */\
                        (<query>)
                        NEAR <numeric_values> USING <metric_names>

```

Similar to our proposed RANGE EXCEPT extension, the word <query> is always a relation. The words <numeric_values> and <metric_names> provide the similarity threshold and the metric associated to each attribute, respectively. The first <query> is T_U and it must present first the attributes in $\overline{L_1}$ than the attributes in L_1 . If there is an index, by default, the query will

always invoke the **Indexed Algorithm 2**, however, there is the possibility pass hints to prevent from using the index. In this case, the query will perform the **Full Table Scan Algorithm 1**. We illustrate the usage in Statement 18.

Statement 18 – A new syntax for Similarity Division

```
(SELECT ClusterID, content FROM UserImages uimg
JOIN Clustering(UserImages.userID) cl ON uimg.userID = cl.ObjectID)
RANGE DIVIDE
(SELECT content FROM Contents)
NEAR 2 USING EDIST
```

4.2 Conclusion

This chapter presented SQL Statements for the Similarity-Aware Division. We employed SIREN's SQL to represent similarity operators and predicates and added two new grammar rules to represent the Similarity Except and the Similarity-Aware Division. The next chapter discusses how these Extended SQL Statements translate to Standard SQL and how the algorithms are implemented.

ALGORITHMS TO EXECUTE THE SIMILARITY-AWARE DIVISION

Chapter 4 presented Statements that the user can write in order to execute the Similarity-Aware Division using an Extended SQL with similarity operators to provide a clean, simple and intuitive syntax. This Chapter discusses how these queries can be translated to RDBMS-executable SQL and how the Full Table Scan and the Indexed Algorithms from Subsection 2.2.1 and a new Similarity-Except can be implemented inside an RDBMS.

5.1 Algorithm Implementations

To implement the three algorithms (Full Table Scan (Algorithm 1), Indexed Algorithm (Algorithm 2) and SIMEXCEPT (Algorithm 3)) on Oracle we used the ODCITable, an interface that supports *table functions*. This interface is very useful because it allows any input relation, either an materialized or a temporary table resulting from an prior operation anywhere in the query plan, to be used.

The implementation is detailed in Statements 19 to 22 using the Full Table Scan algorithm as an example. The implementation of the remaining two algorithm is very similar and therefore omitted. First, the returning relation definition must be known by the RDBMS, so Statements 19 and 20 define an output object type and a table of this output - in our example, a single-column table containing the IDs of valid groups. Then, we define an object associated to the ODCITable interface. This object encapsulates the algorithm and defines interface's functions's implementation. The object has its header illustrated in Statement 21. The function ODCITableStart is responsible for initializing context. ODCITableFetch produces the subset of rows in the result set and it is invoked until the entire result set has been outputted. Finally, the ODCITableClose cleans memory and context after the last call to ODCITableFetch. This pipeline is illustrated in Figure 19. Each of these functions are associated to an implemented

function on FMI-SiR, i.e, function ODCITableStart is implemented as function FTS_Start inside the module. Finally, a function that embodies the implementation is defined in Statement 22. This function is invoked by the user in a SQL Statement, as illustrated in Statement 23.

Statement 19 – Creating the Output type for the interface.

```
CREATE TYPE Output AS OBJECT
(
    groupIdIdentifier INTEGER
);
```

Statement 20 – Creating a collection of the output type.

```
CREATE TYPE OutputSet AS TABLE OF Output;
```

Statement 21 – The interface header definition.

```
CREATE TYPE FTSEImpl as OBJECT
(
    STATIC FUNCTION ODCITableStart(...)
    RETURN PLS_INTEGER
    AS LANGUAGE C LIBRARY libfmsir
    NAME "FTS_Start" WITH CONTEXT
    PARAMETERS (... RETURN INT),

    MEMBER FUNCTION ODCITableFetch(...)
    RETURN PLS_INTEGER
    AS LANGUAGE C LIBRARY libfmsir
    NAME "FTS_Fetch" WITH CONTEXT
    PARAMETERS (... RETURN INT),

    MEMBER FUNCTION ODCITableClose(...)
    RETURN PLS_INTEGER
    AS LANGUAGE C LIBRARY libfmsir
    NAME "FTS_Close" WITH CONTEXT
    PARAMETERS (... RETURN INT)
);
```

Statement 22 – The table function associated to the header

```
CREATE FUNCTION
FTSDivide(t111 SYS_REFCURSOR, t212 SYS_REFCURSOR, tg SYS_REFCURSOR,
distanceFunction VARCHAR2, threshold VARCHAR2)
RETURN OutputSet PIPELINED USING FTSEImpl;
```

Figure 19 – Pipeline of the ODCITable Interface. Function ODCITableStart initializes context, then function ODCITableFetch is recursively called until all tuples have been processed and function ODCITableClose cleans context, memory and closes the connection.

Statement 23 performs the Similarity-Aware Division using the Full Table Scan Algorithm 1 from Section 2.2.1 to answer our running example query "What Users post about a specific set of contents?". The first parameter is the dividend relation T_1 combined with L_1 , the second parameter is the divisor relation T_2 combined with L_2 , the third parameter is the grouping relation T_G and last two parameters are the distance function and the threshold respectively. They are represented as VARCHAR2 because in case of multiple attributes in L_1 and L_2 , their respective distance functions and thresholds are all concatenated. The same approach is used to implement the Indexed Algorithm, however, the first parameter are indexes name for L_1 attributes instead of the actual relation T_1 combined with L_1 . Statement 23 is a direct translation of the Extended SQL's Statement 18. And in regards to the SIMEXCEPT Algorithm, its implementation is also very similar to the implementation of the Full Table Scan Algorithm, except that it does not have the the clustering table, (parameter 3 on Statement 23).

Statement 23 – SQL command that uses the table function.

```

SELECT * FROM TABLE(FTSDivide(
    CURSOR(SELECT userID, content FROM UserImages),
    CURSOR(SELECT content FROM Contents),
    CURSOR(SELECT objectID,clusterID FROM C$UserImagesUserID),
    'euclidean', '2')
);
  
```

This approach is also employed to implement our novel Similariy-Except algorithm. As it as discussed in Chapter 2, the state-of-art of Similariy-Except algorithms are limited to either only uni-dimensional data or do not allow tuples from the same relation to be similar to each other. Although this operator can be provided by combining a set of already supported operators, it is important to provide more efficient options. Thus, we propose the SIMEXCEPT, described in Algorithm 3. To perform a Similarity Except operation $T_1 \hat{-} T_2 = T$ with threshold ε , the SIMEXCEPT algorithm builds in-memory indexes over T_2 and each tuple T_{1i} in

T_1 is used as a query center. After querying through the indexes in T_2 , if the intersection between result sets is empty, T_{1i} is included in the output relation T . SQL usage example of this algorithm is illustrated in Statement 24.

Algorithm 3 – Similarity-Aware Except

SIMEXCEPT(T_1 , T_2 , *distanceFunctions*, *thresholds*)

Result: IDs of tuples from T_1 with no similar to T_2 up to a threshold.

```

1 begin
2   // indexes  $T_1$  to speed-up execution
3   Builds in-memory indexes for  $T_2$  using the appropriated distances listed in distanceFunctions;
4   //IDs of tuples  $T_1$  with no similar to  $T_2$  up to a threshold
5    $T = \emptyset$ ;
6   foreach tuple  $t_i \in T_1$  do
7     // stores tuples similar to  $t_i$ , initially, no tuple is similar
8      $G_{qid} = \emptyset$ ;
9     foreach attribute  $a_j$  from  $t_i$  do
10      //finds tuples that similar to  $t_i$  on attribute  $a_j$ 
11       $G = \text{IndexRangeQuery}(T_2, a_j, t_i, \text{thresholds}[j])$ ;
12      //update valid groups
13      if  $j == i$  then //if this is the first attribute
14        |  $G_{qid} = G$ ;
15      else //for all other attributes
16        |  $G_{qid} = G_{qid} \cap G$ ;
17      end
18    end
19    //if no tuple is similar to  $t_i$ , include it in the output relation  $T$ 
20    if  $G_{qid} = \emptyset$  then
21      |  $T = T \cup G_{qid}$ ;
22    end
23  end
24  return  $T$ ;
25 end

```

5.2 Statement's Translation

Now that we presented how the algorithms are implemented, we show how the queries in Section 4.1 can be translated to Oracle/FMI-SiR functional SQL. In FMI-SiR, the similarity comparisons are encapsulated in UDFs, so the similarity predicate $t1.attr \text{ NEAR } t2.attr \text{ RANGE } 2 \text{ USING EDIST}$ on the Extended SQL is translated to `euclidean_dist(t1.attribute, t2.attribute) <= 2` in FMI-SiR. The Clustering table is translated to a system table in the format `C$tableNameAttributeName`, in our example it is translated from `Clustering (UserImages.userID)` to `C$UserImagesUserID`. The `RANGE EXCEPT` is translated to a query invoking a table function, similar to Statement 23.

5.2.1 Similarity Division using literal translation from Extended Relational Algebra

Statement 24 is the translated query from Statement 15. The query uses two EXCEPT operators. The most internal one invokes our novel Similarity Except Algorithm to remove tuples from $\text{UserImages} \times \text{Contents}$ that are similar to UserImages $\left(T_1.UserID = T_G.ObjectID \right)$ $\bowtie$ $C\$UserImagesUserID$. Because $ClusterID$ is a scalar attribute, there is no need to specify a distance function nor a threshold in the fourth and fifth argument of the internal query. The most external EXCEPT is the Relational Operator provided by the RDBMS.

Statement 24 – FMI-SiR Statement for Similarity Division query using literal translation from Extended Relational Algebra.

```
(SELECT DISTINCT ClusterID FROM C$UserImagesUser)
EXCEPT
(
  SELECT * FROM TABLE(SIMEXCEPT(
    CURSOR(SELECT ClusterID,content FROM C$UserImagesUserID,Contents),
    CURSOR(SELECT ClusterID, tag content UserImages
      JOIN C$UserImagesUserID ON userID = objectID),
    'euclidean', '2' ))
);
```

5.2.2 Similarity Division based on the Double Negation

Statement 25 is the translated query from Statement 16, the ‘‘Double Negation’’ approach. Remember that this approach verifies which group in UserImages does not miss any Content required in relation Contents . It replaces the similarity predicate to an UDF the $\text{Clustering}(\text{UserImages.UserID})$ to $C\$UserImagesUserID$; the UDF function can employ an index, to speed up execution. To distinguish these two different implementations, we refer to the indexed one as ‘‘Indexed Double Negation’’ and to the non-indexed implementation simply as ‘‘Double Negation’’.

Statement 25 – FMI-SiR Statement for Similarity Division query based on the double negation.

```
SELECT DISTINCT ClusterID FROM C$UserImagesUserID uGlobal
WHERE NOT EXISTS
(
  SELECT content FROM Contents co
  WHERE NOT EXISTS
  (
    SELECT ClusterID FROM UserImages uimg
    JOIN C$UserImagesTag uLocal ON uimg.userID = uLocal.objectID
```

```

WHERE uLocal.ClusterID = uGlobal.ClusterID
AND euclidean_dist(uimg.content,co.content) <= 2
)
);

```

5.2.3 Similarity Division based on Set Operations

Statement 26 is the translated query from Statement 17, the ‘ ‘Subset’ ’ approach. This approach checks for every group (UserID) in UserImages if its values in attribute Content form a superset of relation Contents. It uses the Similarity Except algorithm for similarity comparison between relations.

Statement 26 – FMI-SiR Statement for Similarity Division based on Set operations

```

SELECT cs FROM (SELECT DISTINCT ClusterID cs FROM C$UserImagesUserID)
WHERE NOT EXISTS
(
  SELECT * FROM TABLE(SIMEXCEPT(
    CURSOR(SELECT content FROM Contents),
    CURSOR(SELECT content FROM UserImages JOIN C$UserImagesUser
      ON userID = objectID AND clusterID = cs),
    'euclidean', '2' ))
);

```

5.2.4 Similarity Division using the new keyword

Statements 27 and 28 are the two possible translations for Statement 18. If there is an index for attribute Content in UserImages, Statement 18 invokes the Indexed Algorithm; otherwise, it invokes the Full Table Scan. The user can force the translation to one of the options using the appropriate hint as specified in the grammar rule in Subsection 4.1.5. Note that Statement 27 is equal to Statement 23 from Section 5.1; it is repeated here for the reader’s convenience.

Statement 27 – SQL command to invoke Similarity Division using the Full Table Scan Algorithm.

```

SELECT * FROM TABLE(FTSDivide(
  CURSOR(SELECT userID, content FROM UserImages),
  CURSOR(SELECT content FROM Contents),
  CURSOR(SELECT objectID,clusterID FROM C$UserImagesUserID),
  'euclidean', '2')
);

```

Statement 28 – SQL command to invoke the Similarity Division using the Indexed Algorithm.

```
SELECT * FROM TABLE(IndexedAlgorithm(
    'SlimUserImagesContent',
    CURSOR(SELECT content FROM Contents),
    CURSOR(SELECT objectID,clusterID FROM C$UserImagesUserID),
    'euclidean', '2')
);
```

5.3 Experiments and discussion

Each query presented in the previous section performs Similarity-Aware Division using a different query plan. Statements 24 to 26 combine different operators and the RDBMS decides the operator's order and access methods. Statements 27 and 28 invoke algorithms with established execution order. To evaluate which one of these approaches is the most efficient, we created synthetic datasets, using the data generator from [Gonzaga and Cordeiro \(2016\)](#), which represents possible input relations to perform the Similarity-Aware Division. Remember that the division is represented as $T_U [L_1 \hat{\div} L_2] T_2$. Therefore, in our experiments, the dividend T_1 is defined as $Sch(T_1) = (id, signature)$, the divisor T_2 is defined as $Sch(T_2) = (signature)$ and the grouping relation T_G is defined as $Sch(T_G) = (ClusterID, ObjectID)$. In their respective relation, L_1 and L_2 contain the attribute SIGNATURE, a five-dimensional array with values varying from 0 to 1 stored in the RDBMS as a BLOB. The experiments intent to evaluate the scalability of the algorithms in an RDBMS environment. The experiments were performed in an Intel 2 machine with CPU of 2,67GHz, 32GB of RAM running Ubuntu. The Oracle version used was the 12c with the FMI-SiR library.

Unless stated otherwise, the cardinality of the dividend is 100 thousand tuples and the cardinality of the divisor is 5 tuples. We did not create very large divisor relations because division with too many requirements is likely to be useless, returning a very limited or empty result set. A divisor with a thousand tuples makes the division operation too restrictive as it requires each valid group in the dividend to have at least a thousand tuples, and these tuples must be similar to the all of the one thousand tuples in the divisor. Except in Figure 20e, the similarity threshold is always 20% of the estimated diameter of the dataset. All similarity comparisons were performed using the euclidean distance function.

The total number of groups is fixed to 10 but only one group is considered valid, that is, meets all the requirements. The tuples were distributed among groups in the dividend relation in an exponential fashion, that is, the higher the group's ID, the more tuples it has. In the graphics and the following paragraphs Subset refers to Statement 26, Double Negation refers to Statement 25, the Indexed Double Negation refers to the same statement, however, access is done via index. Indexed Algorithm and Full Table Scan refer to Statements 27 and 28,

respectively. The y-axes are always presented in log scale.

Figure 20a shows the run time of the algorithms when increasing the dividend size. The Indexed Double Negation was the most efficient option in every case tested. This is because of the inter-relation comparison task. While Indexed Algorithm and Full Table Scan algorithms load the entire grouping relation in an in-memory *hashtable* and perform several repeated searches in it, the Indexed Double Negation simply joins the query result elements with a filtered grouping relation. The Full Table Scan and the Subset are the reverse versions of each other. While the former builds an index over divisor table, the later builds the index over the dividend table, but both algorithms need to fetch the dividend relation entirely.

Figure 20b shows the runtime of the algorithms when increasing the divisor size. The divisor size varies from 5 to 100 tuples. The Indexed Double Negation algorithm presented an increasing curve, implying that the number of tuples in the divisor relation is directly related to this algorithm's performance. High cardinalities for the divisor relation also affected the Full Table Scan algorithm as it requires more iterations on an internal loop. The other algorithms did not show much impact regarding this parameter.

Figure 20c shows the runtime of the algorithms when increasing the total number of groups, varying from 10 groups to 100. The Indexed Double Negation showed to be more sensitive to this parameter as increasing the number of groups also increases the number of calls to the intern query where the index is accessed repeatedly, resulting in more disk access.

Figure 20d shows the runtime of the algorithms when increasing the number of valid groups, i.e., groups that satisfy all the requirements. The total number of groups was fixed to 40. The x-axis represents the percentage of valid groups where 0.01 is 5 groups and 1 is 40 groups. This characteristic did not seem to affect the algorithms.

Figure 20e shows the runtime of the algorithms when increasing the similarity threshold. The value represents the percentage of the dataset's estimated diameter, in which 0 means identity comparison and 1 means that all elements are considered to be similar to each other. This parameter did not seem to affect the Indexed Algorithm and Subset, however, the Full Table Scan and Indexed Double Negation showed to be more sensitive to large threshold values. This is due to a well-known problem regarding indexes in general; they lose performance in queries with low selectivity. The Full Table Scan algorithm has an index over the divisor table, and it is degrading fast to a linked-list. As for Indexed Double Negation, while still using the very same index of Indexed Algorithm, it differs in the access form and the memory space used to retrieve the elements.

5.3.1 Discussion

As shown in Figure 20, there is no indexed method that works appropriately for all cases. The ‘‘Indexed Double Negation’’ works for most cases, but it is considerably slower than

Figure 20 – Runtime of Similarity-Aware Division Algorithms. We consider the operation represented as $T_1 [L_1 \div L_2] T_2 = T_R$. **Figure 3a** is the runtime when increasing the number of tuples of the **dividend** T_1 . **Figure 3b** is the runtime when increasing the number of tuples of the **divisor** T_2 . **Figure 3c** is the runtime when increasing the **number of groups** in the dividend relation. **Figure 3d** is the runtime when increasing the number of tuples of the **quotient/valid groups** T_R . **Figure 3e** is the runtime when increasing the **similarity threshold** between the **attributes in L_1 and L_2** , that is attributes in the dividend that are compared with the attributes in the divisor.

the ‘‘Indexed Algorithm’’ on high number of groups and tuples on the divisor. The former is also more sensitive to high similarity threshold. As for non-indexed approaches, the ‘‘Subset’’ showed to be more efficient than the ‘‘Full Table Scan’’ algorithm in all cases.

Figure 16, from Section 3.3, presents our proposed architecture: We use SIREN’s query interpreter to translate from an Extended SQL to Oracle’s SQL where similarity processing is done inside the RDBMS. A ‘‘zoomed in’’ abstraction of our proposed SIREN’s translation step, in Figure 21a, shows our initial intention in which the translation step would be done from one Extended SQL Statement presented in Section 4.1 to its counterpart in 5.1. However, our experiments shows that to efficiently perform Similarity-Aware Division, the translator must analyze query parameters such as the similarity threshold, the number of groups present in T_G and the cardinality of the divisor. In this regard, we present a future work: an optimized version of the translation step, shown in Figure 21b. To illustrate, let us consider that the user wrote a query using the ‘‘Indexed Double Negation’’ and the divisor relation cardinality is high. The query translator should translate the query to the ‘‘Indexed Algorithm’’ before submitting to the RDBMS, because empirically, it is more efficient, thus providing faster execution. Moreover, the new keyword `RANGE DIVIDE` could be an even deeper generalization providing easy query writing for the user but for the execution, the query translator and processor analyze query parameters and decide which execution method and Statement is the most efficient.

Figure 21 – Optimizing the Extended SQL Interpreter. a) Currently, the Query Interpreter translates a query written in the Extended SQL to a direct counterpart in Oracle’s SQL. b) Our experiments show that deeper evaluation can be performed and an optimized solution is to take any Statement from the Extended SQL and translate it to the most efficient Oracle’s SQL Statement accordingly to parameters such as the number of groups or divisor’s cardinality.

5.4 Conclusion

In this chapter, we presented implementations to the Extended SQL queries discussed in the previous chapter. We presented one-to-one mapping for each query, that is, each Extended

SQL Statement is translated to a counterpart in Oracle/FMI-SiR. After comparing each implementation, we discovered that there is **no** approach that works appropriately for every case. However, we identified hints to when one algorithm is better than the other. This is a very interesting finding that can be used in future works to execute the Similarity-Aware Division efficiently. Regardless of how the statement is written, the executor should observe parameters such as the divisor's size, number of groups and the presence or absence of indexes and perform the query using the most efficient approach. As future work, we suggest more variables to be explored, such as: 1) The distribution of tuples on each group of the dividend relation and 2) The number of attributes participating in the inter-relation comparison.

TENDER-SIMS - SIMILARITY RETRIEVAL SYSTEM FOR PUBLIC TENDERS

This Chapter presents a case study that validates our proposed implementation of the Similarity-Aware Division. The previous Chapter presented experiments regarding time performance using synthetic data, in this Chapter we employ our proposed method in a realistic approach using real data and evaluate the quality of the results – that is, if the result of the Similarity-Aware Division operation is what it is expected to be in terms of problem’s semantic and domain. This work is a collaboration with colleagues PhD student Daniel Mário de Lima, Msc student Guilherme Felipe Zobot and supervisors Robson Leonardo Ferreira Cordeiro, José Fernando Rodrigues Júnior and Caetano Traina Júnior from the Databases and Imagens Group – GBDI at the Institute of Mathematics and Computer Sciences - ICMC-USP.

The initial step of a public tendering process, known in Portuguese as *licitação pública*, 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, they are usually grouped in lots, which requires that bidding companies provide **all** the goods listed. RFTs are semi-structured documents, varying for each specific request and agency; the product/service descriptions are usually written in natural language. These characteristics make challenging to automatically match companies to tenders.

It is interesting to note that the Division operator from the Relational Algebra is well-suited to find out whether or not a given business can provide all products of a lot, since this operator was designed to answer queries with the concept of “for all”. For example, let us consider a third party company that intermediates between companies and public Request For Tenders processes by matching RFTs that a company can participate in. In this company’s database they would have a relation containing the product catalogs of all the companies they work with - that can be represented by the schema $Sch(CompanyProducts) = (CompanyID, textDescription)$. The database would also contain several relations representing RFTs lots with their list of required

products - represented by the schema $Sch(RFTLotProducts) = (textDescription)$. The Division $\boxed{\text{CompanyProducts} [textDescription \hat{=} textDescription] RFTLotProduct = Companies}$ would return companies that can bid in that particular lot.

However, we emphasize that one cannot expect that the product descriptions presented in a lot will be exactly identical to its counterparts in a business catalog. Requests For Tender usually present product descriptions in a generic way while in a company catalog the product can be heavily detailed presenting information such as trademarks, quantity, packing format, etc. Thus, comparing descriptions by identity ($=$) is useless in this scenario. In this regard, the Similarity-Aware Division is the appropriate operator to find prospective clients that can bid in a lot of a Request For Tender.

With that in mind, this chapter presents Tender-Sims: Tender Retrieval By Similarity System, a system capable of comparing product descriptions in Requests For Tenders with those in businesses catalogs in order to filter out irrelevant lots — those not satisfiable by the company – and provide lots that the company can bid in. This system is aimed at reducing manual work by filtering out irrelevant data. For that, we provide a similarity comparison method to compare products from RFT with products from businesses catalogs using an ontology that organizes terms according to their semantics in a given tender domain and the Jaccard distance. In this comparison, elements from the RFT lots and businesses catalogs are considered to be similar to each other if one of their k-nearest element in the ontology belongs to the same category.

To validate our proposals, we performed a case study using data from real RFTs and businesses catalogs in the context of the Brazilian food market. In this study, our Tender Sims system was highly effective with up to 66% cost reduction at a 90% recall when searching request for tenders where a given business is most likely to be able to attend.

6.1 Code the System

This section details our proposal, the *TendeR-Sims* system – Tender Retrieval by Similarity. Provided that textual product descriptions are better compared for similarity and such form of manipulation is still not properly supported by RDBMS, it becomes necessary to implement new functionalities in the RDBMS environment in order to execute queries that use the Similarity-Aware Division operator with textual data.

We acquired request for tender documents and businesses products catalogs from online websites and created a food ontology to semantically group different representations of similar products. Then, we carefully designed a novel distance function well-suited to compare product descriptions for similarity, based on the developed ontology and the Jaccard distance. It is worth to notice that the similarity measure developed, in which we use the bag-of-words representation and Jaccard distance, was chosen for simplicity. Any other text representation model and similarity metric can be used in TendeR-Sims. Finally, we took advantage of our DBMS implementation

for the Similarity-Aware Division operator to execute the queries.

6.1.1 Data Acquisition

The tender documents were collected from Brazilian governmental agencies and non-governmental organizations websites: the Social Commerce Service of the state of Paraná (SESC-PR)¹, the Regional Council of Engineering and Agronomy of the State of Paraná (CREA-PR)², and the Camboriu/SC City Administration website³. They are all written in Brazilian Portuguese. These websites provide tender documents in Adobe's Portable Document Format (PDF), in such a way that any one can download the documents freely. Only food-related tenders that have been already completed were collected, that is, those that have already been taken by some company. A total of nine RFT documents containing up to 100 lots each were obtained in this process.

We created our dataset by selecting companies that participated in the bidding process and won at least one lot of any the RFTs we obtained previously. In total there were 18 companies. We did not have access to the internal database of the companies participating in the tenders, so we extracted lists of products from their websites using *web-crawlers*. Data was then processed to remove information related to brands, trademarks, units of measure, quantity as well as symbols, stopwords and any other information that was irrelevant about the product itself. Part of the normalized data was used to build the ontology and the other part was used in our experiments.

6.1.2 Building the Ontology

In our ontology, the leaf nodes represent unique product categories and the internal nodes are generalizations of these categories, creating a hierarchical structure in which products share common nodes. We defined categories and subcategories to represent products of a food domain in Brazilian Portuguese in a hierarchical structure, because we did not find an ontology available in this language. In Figure 22, the leaves are illustrated as rectangles and circles represent internal nodes⁴. Thus, “whole bread” and “white bread” are placed in different leaf nodes, however, they are both types of breads, so their ontology relation branches on the bread's node, meaning they are different subcategories of bread products. In the leaf nodes there are many synonyms of the product. These synonyms are used to compare similarity between products. As shown in Figure 22, the leaf product “whole grain” contains the following representations: “whole grain bread vitãozinho 12 grains”, “supreme whole grain bread apple and nut” and “whole bread”. Similarly, the leaf product “white bread” contains the following representations: “white bread vitãozinho”, “supreme bread loaf” and “white bread”.

¹ <<http://www.sescpr.com.br/sesc-parana/licitacoes>>

² <<http://www.crea-pr.org.br/ws/licitacoes-do-crea-pr>>

³ <http://www.camboriu.sc.gov.br/licitacoes_compra.php>

⁴ Note that our ontology is in Portuguese, but we translated it to English in the figure for easier comprehension.

Figure 22 – Overview of our ontology with its classes and subclasses. Adapted from (VASCONCELOS *et al.*, 2018).

The creation of the categories represented in Figure 22 was based on the food pyramid, in which the food domain is divided into four large groups: a) the energetics – the bases of human nutrition such as bread, pasta and other products rich in carbohydrates; b) the builders – protein-rich foods; c) the regulators – known for their high content of vitamins and minerals; and d) the extra energetics – sugars, oils and fats. However, we also inserted new nodes to represent products that do not fit into the four large groups, such as industrialized products, as is the case of canned food, colorants and flavoring agents. Each product description was manually added to the ontology category according to their position in the food pyramid. An example is the bread subcategory, which derives from the energetics.

Based on the ontology created, methods that use comparison for similarity can be implemented using the hierarchy present in the ontology to improve the result returned by the comparison.

6.1.3 The Similarity Comparison

We evaluated two types of similarity: lexical and semantical. To evaluate the lexical similarity, the leaves elements are represented as a bag-of-words. Then, the bag-of-words is evaluated using the Jaccard distance, which is the relation between the amount of exclusive words (set difference) and the overall amount of distinct words (set union). This procedure is represented in Figure 23. According to the first matrix, the two most similar elements to “white bread” are “white bread vitão grão” (placed in leaf node “WHITES”) and “white bread loaf” (placed in leaf node “WHITES”). According to the second matrix, the two most similar elements

"WHITE BREAD"	text repr.	7	9	8	6	4	5
	leaf node	3	3	3	2	2	2
	jaccard	33	33	50	66	86	88
"BREAD"	text repr.	6	9	7	8	4	5
	leaf node	2	3	3	3	2	2
	jaccard	50	66	66	75	84	86

Figure 23 – Illustration of the similarity comparison between “white bread” and “bread”. The ID corresponds to the numbers in Figure 22.

to “bread” are “whole bread” (placed in leaf node “WHOLE GRAIN”) and “white bread loaf” (placed in leaf node “WHITES”).

The semantical similarity is achieved by the ontological categorization of the elements. If one of the k-nearest elements of a product in the RFT’s lot belongs to the same category of one of the k-nearest elements of the businesses catalog, we consider the pair to be similar. In our example in Figure 23, “WHITE BREAD” and “BREAD” are similar because their second element belongs to the same category, that is, the same leaf node.

6.1.4 Implementation

To implement our proposed TendeR-Sims system, we further extended the FMI-SiR similarity retrieval module. As it was discussed before, this module offers resources such as distance functions, feature extractors and database indexes for complex data management, being implemented in C++ as an Oracle library that is controlled by the query processor. This allows applications to manipulate complex data and use the Oracle database resources together.

We inserted into the module our ontology, the Jaccard distance function and the similarity function *OntologySimilarity*, which evaluates pairs of text segments. In order to associate our external similarity function to the schema, one must run the following command in Oracle:

Statement 29 – Statement to create a call to our ontology function.

```
CREATE FUNCTION
OntologySimilarity
(text1 VARCHAR2, text2 VARCHAR2,
ontologyPathFile VARCHAR2, k NUMBER)
RETURN FLOAT
AS
LANGUAGE C LIBRARY libfmsir
NAME "OntologySimilarity"
WITH CONTEXT;
```

Figure 24 – Sequence of operations to execute Similarity-Aware Division. During the query execution, the DBMS query processor executes similarity operation externally.

This command creates a function that takes, as parameters, two texts and the number of neighbouring elements and a structured file containing the ontology. When this function is invoked in a SQL query, the RDBMS changes context from SQL to C++, as shown in Figure 24. The function loads the ontology from a file into memory and evaluates whether or not the pair of product descriptions received is classified to the same node. Then, it returns one if a pair of elements is similar and zero otherwise. The pseudo-code of this function is illustrated in Algorithms 4 and 5. Algorithm 4 is invoked by SQL; it creates the Ontology object based on a structured file and calls the function in Algorithm 5 to compare two products. This result is sent back to the RDBMS, which continues to process the query.

Algorithm 4 – Pseudo-code of the OntologySimilarity algorithm.

`OntologySimilarity(productDescription1, productDescription2, ontologyFile, k)`

Result: a value representing if elements are similar or not

```

1 begin
2   creates ontology structure based on relationships in the ontologyFilePath;
3   if ontologyStructure.similar(productDescription1,productDescription2,k) then
4     return 1;
5   end
6   return 0;
7 end
  
```

We used the ‘‘Double Negation’’ approach to present the Similarity-Aware Division, that is, we selected companies that do not have any product that are not similar to a given product from a given lot of a request for tender, as presented in Statement 30. In our schema, the dividend relation is $Sch(CompanyProducts) = (CompanyID, textDescription)$ where *CompanyID* is the identifier of a company and *textdescription* is the product’s description in the company’s catalog; the divisor relation contains every lot of every RFT in the database and is represented by the schema $Sch(RFTLotProducts) = (RFTID, LOT, textDescription)$. Because of that, to be used as the divisor, the *RFTLotProducts* must be filtered by the lot’s primary key (a composition of *RFTID* and *LOT*).

Statement 30 – Statement to perform Similarity Division on Tender-Sims.

```

SELECT DISTINCT company FROM CompanyProducts
WHERE NOT EXISTS
  
```

```

(
    SELECT textDescription FROM RFTLotProducts request
    WHERE lot = 1 AND RFTID = 1 AND
    NOT EXISTS
    (
        SELECT * FROM CompanyProducts product
        WHERE product.company = this.company
        AND OntologySimilarity(
            request.textDescription,
            product.textDescription,
            "ontology_file", 2
        ) = 1
    )
);

```

Algorithm 5 – Pseudo-code of the similarity evaluation algorithm.

OntologySimilarity(*productDescription1*, *productDescription2*, *k*)

Result: a value representing if elements are similar or not

```

1  begin
2      initialize an empty list of leaf's names and distances for productDescription1;
3      initialize an empty list of leaf's names and distances for productDescription2;
4      foreach leaf in ontologyStructure do
5          foreach textRepresentation in leaf do
6              calculate the jaccard distance between textRepresentation and productDescription1;
7              save value and leaf's name in the list for productDescription1;
8              calculate jaccard distance between textRepresentation and productDescription2;
9              save value and leaf's name in the list for productDescription2;
10         end
11     end
12     foreach item in the list for text1 do
13         foreach item in the list for text2 do
14             if leaf's name in item for productDescription1 is equal to leaf's name in item for
15                 productDescription2 then
16                 return 1;
17             end
18         end
19     end
20 end

```

Figure 25 – Evaluation results. Adapted from (VASCONCELOS *et al.*, 2018)

6.2 Experiments

To evaluate TendeR-Sims, we performed a semi-supervised case study. We randomly selected 20% of the RFT lots and manually labelled all the companies to *satisfy* (real positives) or *not satisfy* (real negatives) each RFT lot. This resulted in a set of 35 RFT lots that were used as our ground truth data, with a positive prevalence of 0.163, i.e., on average each company satisfies 16.3% of all lots.

From this ground truth data, we selected two companies catalogs (11.1%) to “train” our model — only the products in these two catalogs were preprocessed and added to the ontology. A total of 3,068 products categories are present in this training list; they were further filtered by a more strict list of stopwords to better reflect the semantics of the product categories present in the RFT lots. The words removed include numbers, besides manufacturer and product proper names. The resulting list has 2,229 product categories, which were then put into the ontology. The other companies were used as the test dataset.

For each one of the RFT lots in the test dataset, TendeR-Sims queries the companies satisfying all products in the lot. This behaviour makes this system to act as a *filter* that removes irrelevant items from the analytical workflow. Those items are the lots that cannot be satisfied by the company. The efficiency of this filter is measured by *how much* manual work can be avoided by the user when he/she uses TendeR-Sims.

The power of such a filter can be expressed by the results reported in Figure 25. Here, **Specificity** tells how many irrelevant lots are left out of the answer, which should be as high as possible in order to reduce the number of RFTs to be analyzed manually by the user; **Recall** informs how many of relevant lots are returned in the query, which should be high in order to not ignore possibly satisfiable contracts; the value of **Precision** is how many of the returned lots are indeed satisfiable, which should be high, so the user does not spend time analyzing irrelevant

lots; Finally, **F1-Score** is the harmonic average of Precision and Recall, which summarizes the overall predictive and selective power for the positive class.

According to Figure 25 at $k = 1$, if we use TendeR-Sims to filter these RFTs, the system will answer with ≈ 5 RFT lots on average, where 3 or 4 (≈ 3.7) of those can really be satisfied by the company. The Precision of this answer is $3.7/5 \approx 74\%$, and the Recall is $3.7/6 \approx 61\%$. This means that we can find (recall) $\approx 61\%$ of relevant RFT lots by using TendeR-Sims then manually verifying only the five lots that were returned in the answer. For comparison, if we want to find the same amount of relevant lots by hand, we could randomly shuffle and select 21 lots (60% of the dataset), then verify all the selected lots manually. The number of objects to be manually verified in this case was reduced from 21 to 5, a 76.2% cost reduction.

When we adjust the similarity threshold k , it is possible to recall more RFT lots by trading off precision. In Figure 25 we observe that setting $k = 2$ raises Recall to 90.2% and lowers Precision to 47.4%. In this case, TendeR-Sims will return 11 results, where 5 or 6 (≈ 5.4) of them will be relevant lots for the company. Thus, to find 90% of the relevant RFT lots, the manual effort is reduced from 32 (randomly picking 90% of the dataset) to 11 objects by using TendeR-Sims, a 65.6% cost reduction for achieving 90% recall. It is still a significant cost reduction to find nearly all (90%) of the relevant items in the database.

This **Estimated Cost Reduction** (ECR in Figure 25) is then an expected estimate of how many RFT lots will not be needed to be manually processed if the user wants find the same proportion (recall) of relevant RFT lots from the database. Larger databases will benefit even more from our system, for example, a company looking for tendering any RFT lot in the country may need to analyze tens of thousands of lots. As the ECR is a ratio, proportionally many hours may be saved by TendeR-Sims. We then conclude that TendeR-Sims can reduce the amount of manual work and the associated cost by a significant amount.

6.3 Conclusion

TendeR-Sims is a system carefully designed to filter Requests for Tender (RFTs) according to whether or not a company of interest is suited to bid for them. Considering that product descriptions in a request document may differ from the companies product catalogs by little variations, so any comparison by identity (=) will retrieve only exact matches and leave out relevant items, making this approach unsuited for this type of data. On the other hand, comparing complex data is better addressed by how similar (or inversely, how distant) their are. Thus, our system relies on the Similarity-aware Relational Division Operator.

We created a dataset, proposed an ontology and extended a similarity retrieval module so that all data manipulation runs inside a well-known commercial RDBMS (Oracle 12c). Our results indicate that TendeR-Sims can significantly reduce the number of manually analyzed RFT lots and their overall associated cost by up to 76% of the original amount.

CONCLUSION

The Relational Division operator is not suited to process complex data, once it always perform comparisons by equality ($=$) and data such as image, video, music and genetic sequences are better compared for similarity. A brand new study found in the literature extended the Relational Division operator to perform similarity comparison and proposed two efficient algorithms. The proposed Similarity-Aware Division proved to be efficient to deal with complex data answering queries with the concept of “for all”. This Master’s thesis provides a solution to implement this new operator inside an RDBMS. The importance of this work is that it extends the computational power of the traditional RDBMS to better deal with the data management demanded by nowadays applications.

In our work, we proposed a clean and simple syntax for query representation by using a similarity-oriented SQL and efficient execution inside the RDBMS through a similarity retrieval content module. We adapted several Relational Division’s SQL Statements to perform comparisons for similarity and implemented the two state-of-art algorithms using *table functions*. During the evaluation step, we uncovered interesting hints that can be used by the query interpreter to translate queries to the appropriate statement for the Similarity-Aware Division and improve performance. The hints tell which algorithm to perform when there is an the lack of an index, which algorithm to perform when there is high number of groups and which algorithm to perform when there is high cardinality on the divisor relation.

We emphasize that we tackled the support of the Similarity-Aware Division operator in an RDBMS in a similar way the RDBMS supports relational operators such as Relational Join: ($\bowtie$): The algorithm can be represented as a combination of Cartesian Product ($\times$) and Selection(σ), however, the RDBMS provides more efficient execution for this operator. A keyword JOIN exists to represent the operation, whose execution options include at least three well-known physical algorithms: indexed nested loops, hash join and merge join, besides several variations such as INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN and CROSS JOIN. Similarly, we provided a SQL keyword for easy represen-

tation of the Similarity-Aware Division and several execution options. Then, we provided hints to the query optimizer, so it knows when to use each execution option and perform the Similarity-Aware Division in the most efficient manner.

In order to show how the Similarity-Aware Division can be employed on real world situations, we evaluated the quality of the results in a semi-supervised case study: A System that finds prospective clients for Requests For Tender, known in Portuguese as *licitações públicas*. Our results show that the system is able to identify 90% of the relevant RFTs documents, and significantly reduce the manual work by 76%.

This Master's thesis generated the following publications:

1. Guilherme Queiroz Vasconcelos, Daniel dos Santos Kaster, Robson Leonardo Ferreira Cordeiro: On the Support of the Similarity-Aware Division Operator in a Commercial RDBMS. In: 22nd European Conference on Advances in Databases and Information Systems, 2018, Budapest, Hungary. Lecture Notes in Computer Science. 1ed.: Springer International Publishing, 2018, v. 1, p. 142-155. (International Conference **Qualis B1**);
2. Guilherme Queiroz Vasconcelos, Guilherme Felipe Zabet, Daniel Mario de Lima, José Fernando Rodrigues Júnior, Caetano Traina Júnior, Daniel dos Santos Kaster, Robson Leonardo Ferreira Cordeiro: TendeR-Sims - Similarity Retrieval System for Public Tenders. In: 20th International Conference on Enterprise Information Systems, 2018, Funchal, Portugal. Proceedings of the 20th International Conference on Enterprise Information Systems, 2018. p. 143-150. (International Conference **Qualis B2**);

Here we present ideas to be explored as future work stemming from this Master's thesis:

1. **Exploring and identifying new optimization hints:** Experiments in Section 5.3 showed that there are no algorithm that is efficient on all cases, but provided hints that show when one algorithm is more efficient than the other. We present, as future work, to improve these hints as well as to explore and to identify new ones by adding new variables to the operation. We also suggest the following variables to be explored:
 - a) **The distribution of tuples on each group of the dividend relation:** In our experiments, the tuples on the dividend table were assigned to groups in an exponential fashion where the higher the group's ID, the more tuples it would be assigned. We propose to assign tuples to groups following the normal and uniform distribution and evaluate the algorithm's behaviour.
 - b) **The number of attributes participating in the inter-relation comparison:** Consider the cases where a dividend table needs to be divided according to two or more attributes. Would this affect one of the algorithms experimented? To illustrate, let us use our running example, the query: "What Users post about a specific

set of contents?". However, in a new scenario where there are two feature extractors representing the image: one feature extractor describes texture and the other describes the colors. So our schema is represented by a dividend T_1 defined as $Sch(T_1) = (id, color, texture)$ and a divisor T_2 defined as $Sch(T_2) = (color, texture)$.

2. **An optimized parser:** Stemming from item 1, we present as future work to create an optimized parser that receives as input a query using the Similarity-Aware Division operator and the parser translates it to the most efficient form according to the parameters and the knowledge obtained by the hints. To illustrate, let us consider that the user wrote a query using the "Indexed Double Negation" and the divisor relation cardinality is high. The parser should translate the query to the "Indexed Algorithm" before submitting to the RDBMS. Similarly, if the user writes the query using the new keyword RANGE DIVIDE, the parser should decide execution method and Statement is the most efficient. The challenges in this idea is to evaluate if a query contains the Similarity-Aware Division operator because it can be represented by a combination of many operators and even nested queries. A possible idea would be to check if the submitted query's parse tree contains a parse tree of one of Statements from Section 4.1 the as subtree.
3. **Algorithms for Similarity-Aware Division with more attributes:** Stemming from item 1b, we present as future work to propose new algorithms for the Similarity-Aware Division when dealing with more attributes participating in the inter-relation comparison. While all approaches presented in this work are able to process queries with more attributes, none of them use the selectivity of the attributes efficiently, to improve performance, specially the indexed approaches such as the "Indexed Algorithm" or the "Indexed Double Negation". In this regard, we present the idea and the need of a new algorithm, for this specific scenario, that takes into account the selectivity of the attributes. A suggestion would be an algorithm that compares attributes from the most selective to the least selective and uses the index of the most selective attribute as a filter.
4. **Identification of algebraic properties for the Similarity-Aware Division Operator:** Similar to the works of [Atnafu, Brunie and Kosch \(2001\)](#), [Atnafu et al. \(2004\)](#), [Traina-Jr. et al. \(2006\)](#), [Ferreira et al. \(2011\)](#), [Silva, Aref and Ali \(2009\)](#) and [Silva et al. \(2015\)](#), we present as future work to explore properties of the Similarity-Aware Division operator in the query plan and its relation with others similarity and algebraic operators. Is the operator associative? Is it commutative? Distributive? How can it be moved in the query plan?

BIBLIOGRAPHY

ADALI, S.; BONATTI, P.; SAPINO, M. L.; SUBRAHMANYAN, V. S. A multi-similarity algebra. **SIGMOD Rec.**, ACM, New York, NY, USA, v. 27, n. 2, p. 402–413, Jun. 1998. ISSN 0163-5808. Available: <http://doi.acm.org/10.1145/276305.276340>. Citation on page 34.

ATNAFU, S.; BRUNIE, L.; KOSCH, H. Similarity-based operators and query optimization for multimedia database systems. In: **International Database Engineering & Applications Symposium, IDEAS '01, July 16-18, 2001, Grenoble, France, Proceedings**. [s.n.], 2001. p. 346–355. Available: <https://doi.org/10.1109/IDEAS.2001.938103>. Citations on pages 46 and 87.

ATNAFU, S.; CHBEIR, R.; COQUIL, D.; BRUNIE, L. Integrating similarity-based queries in image dbms. In: **Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, March 14-17, 2004**. [s.n.], 2004. p. 735–739. Available: <https://doi.org/10.1145/967900.968052>. Citations on pages 46 and 87.

BARIONI, M. C. N.; RAZENTE, H. L.; TRAINA, A. J. M.; TRAINA-JR., C. SIREN: A similarity retrieval engine for complex data. In: **Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006**. [s.n.], 2006. p. 1155–1158. Available: <http://dl.acm.org/citation.cfm?id=1164232>. Citation on page 47.

_____. Seamlessly integrating similarity queries in SQL. **Softw., Pract. Exper.**, v. 39, n. 4, p. 355–384, 2009. Available: <https://doi.org/10.1002/spe.898>. Citations on pages 34, 46, 47, and 55.

BEDO, M. V. N.; TRAINA, A. J. M.; Traina Jr., C. Seamless integration of distance functions and feature vectors for similarity-queries processing. **JIDM**, v. 5, n. 3, p. 308–320, 2014. Available: <https://seer.lcc.ufmg.br/index.php/jidm/article/view/689>. Citation on page 49.

BöHM, C.; BERCHTOLD, S.; KEIM, D. A. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 33, n. 3, p. 322–373, Sep. 2001. ISSN 0360-0300. Available: <http://doi.acm.org/10.1145/502807.502809>. Citations on pages 34 and 40.

BUDÍKOVÁ, P.; BATKO, M.; ZEZULA, P. Query language for complex similarity queries. In: **Advances in Databases and Information Systems - 16th East European Conference, ADBIS 2012, Poznań, Poland, September 18-21, 2012. Proceedings**. [s.n.], 2012. p. 85–98. Available: https://doi.org/10.1007/978-3-642-33074-2_7. Citations on pages 34 and 46.

CAMPS, D. High performance relational division in sql server. **Simple-Talk**, Red Gate, 2014. [Online; accessed April 26, 2016]. Available: <https://www.simple-talk.com/sql/t-sql-programming/divided-we-stand-the-sql-of-relational-division>. Citation on page 43.

CARÉLO, C. C. M.; POLA, I. R. V.; CIFERRI, R. R.; TRAINA, A. J. M.; TRAINA-JR., C.; CIFERRI, C. D. de A. The onion-tree: Quick indexing of complex data in the main memory. In: _____. **Advances in Databases and Information Systems: 13th East European Conference**,

ADBIS 2009, Riga, Latvia, September 7-10, 2009. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 235–252. ISBN 978-3-642-03973-7. Available: https://doi.org/10.1007/978-3-642-03973-7_18. Citations on pages 41 and 42.

CELKO, J. Divided we stand: The sql of relational division. **Simple-Talk**, Red Gate, 2009. [On-line; accessed April 26, 2016]. Available: <https://www.simple-talk.com/sql/t-sql-programming/divided-we-stand-the-sql-of-relational-division>. Citation on page 43.

CHÁVEZ, E.; NAVARRO, G.; BAEZA-YATES, R.; MARROQUÍN, J. L. Searching in metric spaces. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 33, n. 3, p. 273–321, Sep. 2001. ISSN 0360-0300. Available: <http://doi.acm.org/10.1145/502807.502808>. Citations on pages 25, 26, 33, 34, and 40.

CHEN, L.; GAO, Y.; LI, X.; JENSEN, C. S.; CHEN, G. Efficient metric indexing for similarity search and similarity joins. **IEEE Transactions on Knowledge and Data Engineering**, v. 29, n. 3, p. 556–571, March 2017. ISSN 1041-4347. Citations on pages 35 and 40.

CHEN, L.; GAO, Y.; ZHENG, B.; JENSEN, C.; YANG, H.; YANG, K. Pivot-based metric indexing. **Proceedings of the VLDB Endowment**, v. 10, p. 1058–1069, 06 2017. Citation on page 40.

CHÁVEZ, E.; NAVARRO, G. A compact space decomposition for effective metric indexing. **Pattern Recognition Letters**, v. 26, p. 1363–1376, 07 2005. Citation on page 40.

CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: **Proceedings of the 23rd International Conference on Very Large Data Bases**. [S.l.: s.n.], 1997. p. 426–435. Citation on page 40.

CODD, E. F. Relational completeness of data base sublanguages. **IBM Research Report**, RJ987, 1972. Citations on pages 25, 31, and 32.

_____. **The Relational Model for Database Management, Version 2**. [S.l.]: Addison-Wesley, 1990. ISBN 0-201-14192-2. Citations on pages 31 and 32.

CORDEIRO, R. L. F.; FALOUTSOS, C.; TRAINA-JR., C. **Data Mining in Large Sets of Complex Data**. Springer, 2013. (Springer Briefs in Computer Science). ISBN 978-1-4471-4889-0. Available: <https://doi.org/10.1007/978-1-4471-4890-6>. Citations on pages 25 and 33.

DATABASE, O. **Data Cartridge's Developer's Guide**. 2010. 11g Release 2 (11.2). Citation on page 25.

DOHNAL, V.; GENNARO, C.; SAVINO, P.; ZEZULA, P. D-index: Distance searching index for metric data sets. **Multimedia Tools Appl.**, v. 21, n. 1, p. 9–33, 2003. Available: <https://doi.org/10.1023/A:1025026030880>. Citation on page 40.

DOHNAL, V.; GENNARO, C.; ZEZULA, P. Similarity join in metric spaces using ed-index. In: **Database and Expert Systems Applications, 14th International Conference, DEXA 2003, Prague, Czech Republic, September 1-5, 2003, Proceedings**. [s.n.], 2003. p. 484–493. Available: https://doi.org/10.1007/978-3-540-45227-0_48. Citations on pages 35 and 40.

DRAKEN, E.; GAO, S.; ALHAJJ, R. Making query coding in sql easier by implementing the sql divide keyword: An experimental query rewriter in java. In: _____. first. [S.l.]: IGI Global, 2001. chap. 12. Citations on pages 19, 45, 46, and 55.

ELMASRI, R.; NAVATHE, S. B. **Sistemas de Banco de Dados**. 6th. ed. Boston, MA, USA: Pearson Education Inc, 2010. Citations on pages 25 and 43.

FERREIRA, M. R. P.; SANTOS, L. F. D.; TRAINA, A. J. M.; DIAS, I.; CHBEIR, R.; TRAINA-JR., C. Algebraic properties to optimize knn queries. **JIDM**, v. 2, n. 3, p. 385–400, 2011. Available: <<http://seer.lcc.ufmg.br/index.php/jidm/article/view/143>>. Citations on pages 46 and 87.

FERREIRA, M. R. P.; TRAINA, A. J. M.; DIAS, I.; CHBEIR, R.; TRAINA-JR., C. Identifying algebraic properties to support optimization of unary similarity queries. In: **Proceedings of the 3rd Alberto Mendelzon International Workshop on Foundations of Data Management, Arequipa, Peru, May 12-15, 2009**. [s.n.], 2009. Available: <<http://ceur-ws.org/Vol-450/paper6.pdf>>. Citation on page 34.

FILHO, R. F. S.; TRAINA, A. J. M.; TRAINA-JR., C.; FALOUTSOS, C. Similarity search without tears: The OMNI family of all-purpose access methods. In: **Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany**. [s.n.], 2001. p. 623–630. Available: <<https://doi.org/10.1109/ICDE.2001.914877>>. Citation on page 40.

GANTER, B.; WILLE, R. **Formal concept analysis: mathematical foundations**. [S.l.]: Springer Science & Business Media, 2012. Citation on page 25.

GONZAGA, A. dos S.; CORDEIRO, R. L. The similarity-aware relational division database operator with case studies in agriculture and genetics. **Information Systems**, v. 82, p. 71 – 87, 2019. ISSN 0306-4379. Available: <<http://www.sciencedirect.com/science/article/pii/S0306437917303484>>. Citations on pages 38 and 39.

GONZAGA, A. S.; CORDEIRO, R. L. F. Fast and scalable relational division on database systems. In: **31º Simpósio Brasileiro de Banco de Dados, SBBD 2016, Salvador, Bahia, Brasil, October 4-7, 2016**. [s.n.], 2016. p. 169–174. Available: <http://sbbd2016.fpc.ufba.br/sbbd2016/sbbd/sbbd_s012.pdf>. Citations on pages 43, 44, 55, and 69.

_____. A new division operator to handle complex objects in very large relational datasets. In: **Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017**. [s.n.], 2017. p. 474–477. Available: <<https://doi.org/10.5441/002/edbt.2017.48>>. Citations on pages 26 and 45.

_____. The similarity-aware relational division database operator. In: **Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017**. [s.n.], 2017. p. 913–914. Available: <<http://doi.acm.org/10.1145/3019612.3019869>>. Citations on pages 26 and 38.

GULIATO, D.; MELO, E. V.; RANGAYYAN, R. M.; SOARES, R. C. POSTGRESQL-IE: an image-handling extension for postgresql. **J. Digital Imaging**, v. 22, n. 2, p. 149–165, 2009. Available: <<https://doi.org/10.1007/s10278-007-9097-5>>. Citation on page 51.

IQBAL, K.; ODETAYO, M. O.; JAMES, A. Content-based image retrieval approach for biometric security using colour, texture and shape features controlled by fuzzy heuristics. **Journal of Computer and System Sciences**, v. 78, n. 4, p. 1258–1277, 2012. Citations on pages 25 and 33.

JACOX, E. H.; SAMET, H. Metric space similarity joins. **ACM Trans. Database Syst.**, v. 33, n. 2, 2008. Citations on pages 26 and 35.

KANELLAKIS, P. C. **Elements of Relational Database Theory**. Providence, RI, USA, 1989. Citation on page 25.

KASTER, D. d. S. **Tratamento de Condições Especiais para Busca por Similaridade em Bancos de Dados Complexos**. Phd Thesis (PhD Thesis) — Instituto de Ciências Matemáticas e de Computação, 2012. Citation on page 41.

KASTER, D. dos S.; BUGATTI, P. H.; TRAINA, A. J. M.; TRAINA-JR., C. Fmi-sir: A flexible and efficient module for similarity searching on oracle database. **Journal of Information and Data Management**, v. 1, n. 2, p. 229–244, 2010. Citations on pages 47 and 50.

Kim, S. J.; Al-Kateb, M.; Sinclair, P.; Crolotte, A.; Zhang, C.; Rose, L. Dynamic statistics collection in the teradata unified data architecture. In: **2017 IEEE 33rd International Conference on Data Engineering (ICDE)**. [S.l.: s.n.], 2017. p. 255–258. ISSN 2375-026X. Citation on page 25.

LAVERDE, N. A.; CAZZOLATO, M. T.; TRAINA, A. J. M.; TRAINA-JR., C. Semantic similarity group by operators for metric data. In: BEECKES, C.; BORUTTA, F.; KRÖGER, P.; SEIDL, T. (Ed.). **Similarity Search and Applications**. Cham: Springer International Publishing, 2017. p. 247–261. Citation on page 38.

LU, W.; HOU, J.; YAN, Y.; ZHANG, M.; DU, X.; MOSCIBRODA, T. Msql: efficient similarity search in metric spaces using sql. **The VLDB Journal**, v. 26, p. 1–26, 10 2017. Citation on page 51.

MARRI, W. J. A.; MALLUHI, Q. M.; OUZZANI, M.; TANG, M.; AREF, W. G. The similarity-aware relational intersect database operator. In: **Similarity Search and Applications - 7th International Conference, SISAP 2014, Los Cabos, Mexico, October 29-31, 2014. Proceedings**. [s.n.], 2014. p. 164–175. Available: <https://doi.org/10.1007/978-3-319-11988-5_15>. Citations on pages 26 and 35.

_____. The similarity-aware relational database set operators. **Inf. Syst.**, v. 59, p. 79–93, 2016. Available: <<https://doi.org/10.1016/j.is.2015.10.008>>. Citation on page 35.

MATOS, V. M.; GRASSER, R. Assessing performance of the relational division operator. **Data Base Management**, p. 22–20, 2001. Citations on pages 43, 44, and 55.

MICO, L.; ONCINA, J.; CARRASCO, R. A fast branch bound nearest neighbour classifier in metric spaces. **Pattern Recognition Letters**, v. 17, p. 731–739, 06 1996. Citation on page 40.

NESSO-JR, M. R.; CAZZOLATO, M. T.; SCABORA, L. C.; OLIVEIRA, P. H.; SPADON, G.; SOUZA, J. A. de; OLIVEIRA, W. D.; CHINO, D. Y. T.; JR., J. F. R.; TRAINA, A. J. M.; TRAINA-JR., C. RAFIKI: retrieval-based application for imaging and knowledge investigation. In: **31st IEEE International Symposium on Computer-Based Medical Systems, CBMS 2018, Karlstad, Sweden, June 18-21, 2018**. [s.n.], 2018. p. 71–76. Available: <<https://doi.org/10.1109/CBMS.2018.00020>>. Citation on page 51.

OLIVEIRA, P. H.; FRAIDEINBERZE, A. C.; LAVERDE, N. A.; GUALDRON, H.; GONZAGA, A. S.; FERREIRA, L. D.; OLIVEIRA, W. D.; Rodrigues Jr., J. F.; CORDEIRO, R. L. F.; Traina Jr., C.; TRAINA, A. J. M.; SOUSA, E. P. M. On the support of a similarity-enabled

relational database management system in civilian crisis situations. In: **Proceedings of the 18th International Conference on Enterprise Information Systems, ICEIS 2016**. [s.n.], 2016. p. 119–126. Available: <https://doi.org/10.5220/0005816701190126>. Citations on pages 25, 33, and 51.

PAPADIAS, D.; TAO, Y.; MOURATIDIS, K.; HUI, C. K. Aggregate nearest neighbor queries in spatial databases. **ACM Trans. Database Syst.**, ACM, New York, NY, USA, v. 30, n. 2, p. 529–576, Jun. 2005. ISSN 0362-5915. Available: <http://doi.acm.org/10.1145/1071610.1071616>. Citation on page 34.

PAREDES, R.; REYES, N. Solving similarity joins and range queries in metric spaces with the list of twin clusters. **Journal of Discrete Algorithms**, v. 7, n. 1, p. 18 – 35, 2009. ISSN 1570-8667. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP). Available: <http://www.sciencedirect.com/science/article/pii/S1570866708000774>. Citation on page 35.

PEARSON, S. S.; SILVA, Y. N. Index-based r-s similarity joins. In: _____. **Similarity Search and Applications: 7th International Conference, SISAP 2014, Los Cabos, Mexico, October 29-31, 2014. Proceedings**. Cham: Springer International Publishing, 2014. p. 106–112. Citation on page 35.

POLA, I. R. V.; CORDEIRO, R. L. F.; TRAINA-JR., C.; TRAINA, A. J. M. A new concept of sets to handle similarity in databases: The simsets. In: **Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings**. [s.n.], 2013. p. 30–42. Available: https://doi.org/10.1007/978-3-642-41062-8_4. Citation on page 36.

_____. Similarity sets: A new concept of sets to seamlessly handle similarity in database management systems. **Inf. Syst.**, v. 52, p. 130–148, 2015. Available: <https://doi.org/10.1016/j.is.2015.01.011>. Citations on pages 35 and 36.

RAZENTE, H. L.; BARIONI, M. C. N.; TRAINA, A. J. M.; TRAINA-JR., C. Aggregate similarity queries in relevance feedback methods for content-based image retrieval. In: **Proceedings of the 2008 ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2008. (SAC '08), p. 869–874. ISBN 978-1-59593-753-7. Available: <http://doi.acm.org/10.1145/1363686.1363887>. Citation on page 34.

SAMET, H. **Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 0123694469. Citations on pages 25, 26, 33, 34, and 40.

SILBERSCHATZ, A.; KORTH, H.; SUDARSHAN, S. **Database Systems Concepts**. 5. ed. New York, NY, USA: McGraw-Hill, Inc., 2006. ISBN 0072958863, 9780072958867. Citation on page 25.

SILVA, Y. N.; ALY, A. M.; AREF, W. G.; LARSON, P. Simdb: a similarity-aware database system. In: **Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010**. [s.n.], 2010. p. 1243–1246. Available: <http://doi.acm.org/10.1145/1807167.1807330>. Citations on pages 46 and 51.

SILVA, Y. N.; AREF, W. G.; ALI, M. H. Similarity group-by. In: **Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China**. [s.n.], 2009. p. 904–915. Available: <https://doi.org/10.1109/ICDE.2009.113>. Citations on pages 26, 36, 46, and 87.

SILVA, Y. N.; AREF, W. G.; LARSON, P.-A.; PEARSON, S. S.; ALI, M. H. Similarity queries: Their conceptual evaluation, transformations and processing. **The VLDB Journal**, v. 22, n. 3, 2013. Citation on page 34.

SILVA, Y. N.; PEARSON, S. Exploiting database similarity joins for metric spaces. **Proceedings of the VLDB Endowment**, v. 5, p. 1922–1925, 08 2012. Citation on page 35.

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. Available: <https://doi.org/10.1016/j.is.2015.01.008>. Citations on pages 35, 46, and 87.

STONEBRAKER, M.; ROWE, L. A. The design of postgres. **SIGMOD Rec.**, ACM, New York, NY, USA, v. 15, n. 2, p. 340–355, Jun. 1986. ISSN 0163-5808. Available: <http://doi.acm.org/10.1145/16856.16888>. Citation on page 25.

TANG, M.; TAHBOUB, R. Y.; AREF, W. G.; ATALLAH, M. J.; MALLUHI, Q. M.; OUZZANI, M.; SILVA, Y. N. Similarity group-by operators for multi-dimensional relational data. **IEEE Transactions on Knowledge and Data Engineering**, v. 28, n. 2, p. 510–523, Feb 2016. ISSN 1041-4347. Citation on page 37.

TRAINA, A. J. M.; TRAINA, C.; BUENO, J. M.; CHINO, F. J. T.; AZEVEDO-MARQUES, P. Efficient content-based image retrieval through metric histograms. **World Wide Web**, v. 6, n. 2, p. 157–185, 2003. Citations on pages 25 and 33.

TRAINA-JR., C.; TRAINA, A.; SEEGER, B.; FALOUTSOS, C. Slim-trees: High performance metric trees minimizing overlap between nodes. In: _____. **Advances in Database Technology — EDBT 2000: 7th International Conference on Extending Database Technology Konstanz, Germany, March 27–31, 2000 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 51–65. Available: https://doi.org/10.1007/3-540-46439-5_4. Citation on page 40.

TRAINA-JR., C.; TRAINA, A. J. M. Fast indexing and visualization of metric data sets using slim-trees. **IEEE Transaction On Knowledge And Data Engineering**, v. 14, n. 2, p. 244–260, 2002. Citation on page 26.

TRAINA-JR., C.; TRAINA, A. J. M.; VIEIRA, M. R.; ARANTES, A. S.; FALOUTSOS, C. Efficient processing of complex similarity queries in rdbms through query rewriting. In: **Proceedings of the 15th ACM International Conference on Information and Knowledge Management**. New York, NY, USA: ACM, 2006. (CIKM '06), p. 4–13. ISBN 1-59593-433-2. Available: <http://doi.acm.org/10.1145/1183614.1183620>. Citations on pages 34, 46, and 87.

TROJACANEC, K.; DIMITROVSKI, I.; LOSKOVSKA, S. Content based image retrieval in medical applications: An improvement of the two-level architecture. In: **IEEE EUROCON 2009**. [S.l.: s.n.], 2009. p. 118–121. Citations on pages 25 and 33.

VASCONCELOS, G. Q.; ZABOT, G. F.; LIMA, D. M. de; JR., J. F. R.; TRAINA-JR., C.; KASTER, D. dos S.; CORDEIRO, R. L. F. Tender-sims - similarity retrieval system for public tenders. In: **ICEIS 2018**. [S.l.: s.n.], 2018. p. 143–150. Citations on pages 25, 26, 33, 78, and 82.

WORBOYS, M. F.; DUCKHAM, M. **GIS: a computing perspective**. [S.l.]: CRC press, 2004. Citation on page [25](#).

ZEZULA, P.; AMATO, G.; DOHNAL, V.; BATKO, M. **Similarity Search: The Metric Space Approach**. 1st. ed. United States: Springer Publishing Company, Incorporated, 2010. Citations on pages [25](#), [26](#), [33](#), and [34](#).