UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Storage and Navigation Operations on Graphs in Relational DBMS

Lucas de Carvalho Scabora

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

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Lucas de Carvalho Scabora

Storage and Navigation Operations on Graphs in Relational DBMS

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

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Caetano Traina Junior

USP – São Carlos
May 2021

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

S277s Scabora, Lucas de Carvalho
 Storage and Navigation Operations on Graphs in
 Relational DBMS / Lucas de Carvalho Scabora;
 orientador Caetano Traina Junior. -- São Carlos,
 2021.
 116 p.

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

 1. Recursive Queries. 2. Graph Queries. 3.
 RDBMS. 4. Adjacency-List Storage Approaches. I.
 Traina Junior, Caetano, orient. II. Título.

Lucas de Carvalho Scabora

Armazenamento e Operações de Navegação em Grafos em SGBDs Relacionais

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

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Caetano Traina Junior

USP – São Carlos
Maio de 2021

*I dedicate this Thesis to my family,
that always believed and supported me
in the best and worst days of my journey.*

ACKNOWLEDGEMENTS

O presente trabalho foi realizado com apoios da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001, da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) - Processos 2016/17330-1, 2016/17078-0, e 2020/07200-9, e do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

Primeiramente agradeço a Deus pelas bênçãos que recebi em minha vida.

Aos meus pais, José Roberto e Maria Cristina, à minha irmã Thais e ao meu cunhado Gabriel, por sempre me apoiarem em minhas escolhas, incentivarem meus estudos, e estarem sempre presentes quando precisei. Sou imensamente grato a eles.

Ao meu orientador, Prof. Dr. Caetano Traina Junior, por ter me orientado, auxiliado, e ensinado durante todo o projeto. À Prof.^a Dr.^a Agma Juci Machado Traina, por me apoiar nos mais diversos momentos quando precisei.

Ao Prof. Dr. Daniel dos Santos Kaster, por sempre me auxiliar, ensinar e, quando necessário, fazer críticas construtivas em prol de meu crescimento tanto pessoal quanto profissional. Ele foi um verdadeiro companheiro e amigo que espero ter por toda a vida.

Ao Grupo de Bases de Dados e Imagens (GBDI), por todas as oportunidades de colaboração e de trocas de experiência. E a todos os amigos que lá fiz: Jaqueline Brito, Gabriel Spadon, Mirela Cazzolato, Paulo Oliveira, Daniel Chino, Willian Oliveira, e muitos outros.

Aos amigos que me auxiliaram nos momentos mais difíceis, em especial ao Armando Toda e ao Bruno Bottazzini, por não desistirem de mim nos piores momentos que passei.

Aos demais professores e funcionários do ICMC, pelo suporte ao longo desses anos, inclusive durante minha graduação e mestrado.

E a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho.

*“Grafozinho quando nasce,
espalha arestas pelo chão,
Relacional quando dorme,
bota índice na relação. ”*

(Adaptado de Silvio Romero, em Cantos populares do Brasil)

ABSTRACT

SCABORA, L. C. **Storage and Navigation Operations on Graphs in Relational DBMS**. 2021. 116 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

A complex network models relationships among data elements in several applications, such as social networks, urban street organization, disease monitoring and treatment co-occurrences, and communication and transport infrastructures. To store and manage large volumes of data, the Relational Database Management Systems (RDBMSs) provide efficient disk-based data management, being present in the most diverse application domains, including enterprise information, finance, medical or clinical support, and in airline reservations and scheduling. However, RDBMSs still face challenges to efficiently process queries that “navigate” over complex networks represented as graphs. Those queries, named in this work as graph queries, are often expressed by recursive queries, which execute multiple join operations between an internal, temporary table that stores intermediary results and the table that stores the graph’s edges. This Ph.D. dissertation focuses on developing methods and specialized data structures based on adjacency-lists aiming at improving graph query performance in an RDBMS through recursive queries. This project’s main goal targets the question: “*How can we improve the performance of queries over graph data executed in a RDBMS?*”. To answer it, we proposed specializations over each table involved in a recursive query: the table of edges E , and the temporary result table T . The modifications in table T consists of adding extra measurement columns to enable employing it in multiple graph queries executed in a single database connection. In its turn, table E is specialized to allow storing multiple edges into a single row optimized for the execution of each join operation of the recursive query. Experiments using data from RDBMSs modeled as a graph allow concluding that: (i) enriching table T provides increased efficiency to execute multiple graph queries, such as the query types Single-Source Shortest Path (SSSP), Connected Components (CC), and PageRank (PR); (ii) grouping multiple edges from table E into a single row significantly improves the performance of individual recursive queries, reducing not only their elapsed time but also the required size to store the graph data; and (iii) employing Clustered Tables to group the edges into a single page also enables improved performance for recursive queries, with the advantage of requiring just a few changes in the SQL statements. Overall, specializing both tables T and E helps diminishing the gap of efficiently executing graph queries in a RDBMS.

Keywords: Recursive Queries, Graph Queries, RDBMS, Adjacency-List Storage Approaches.

RESUMO

SCABORA, L. C. **Armazenamento e Operações de Navegação em Grafos em SGBDs Relacionais**. 2021. 116 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

Uma rede complexa modela a inter-relação entre elementos nas mais diversas aplicações, tais como redes sociais, organização de ruas em regiões urbanas, monitoramento de doenças e co-ocorrência de tratamentos e infraestruturas de comunicação e de transporte. Para armazenar e gerenciar grandes volumes de dados, os Sistemas Gerenciadores de Bases de Dados Relacionais (SGBDRs) proporcionam um eficiente gerenciamento dos dados em disco, estando presente nos mais diversos domínios de aplicação, incluindo o gerenciamento de informações corporativas, de finanças, suporte médico ou clínico, e em reservas e agendamento de passagens aéreas. Entretanto, os SGBDRs ainda enfrentam desafios para processar eficientemente consultas para “navegar” em redes complexas representadas por grafos. Essas consultas, chamadas neste trabalho de consultas em grafos, são frequentemente expressas por meio de consultas recursivas, as quais realizam várias operações de junção entre uma tabela temporária interna que armazena resultados intermediários e a tabela que armazena as arestas do grafo. Esta tese de doutorado foca no desenvolvimento de métodos e de estruturas de dados especializadas baseadas em listas de adjacência com o objetivo de melhorar o desempenho de consultas em grafos em um SGBDR através de consultas recursivas. O objetivo principal deste projeto visa a questão: “*Como podemos melhorar o desempenho de consultas sobre dados de grafos executados em um SGBDR?*”. Para respondê-la, foram propostas especializações em cada uma das tabelas envolvidas na consulta recursiva: a tabela de arestas E e a tabela de resultados T . As modificações na tabela T consistem em adicionar colunas de medida extras para permitir sua utilização em múltiplas consultas em grafos executadas em uma única conexão de banco de dados. Por sua vez, a tabela E é especializada para permitir o armazenamento de múltiplas arestas em uma única linha otimizada para a execução de cada operação de junção da consulta recursiva. Experimentos usando dados de SGBDRs modelados como grafos permitem concluir que: (i) enriquecer a tabela T melhora a eficiência para executar várias consultas em grafo, como os tipos de consulta Single-Source Shortest Path (SSSP), Componentes Conexas (CC) e PageRank (PR); (ii) agrupar as arestas da tabela E em uma única linha melhora significativamente o desempenho de consultas recursivas individualmente, reduzindo não apenas o tempo decorrido, mas também o tamanho necessário em disco para armazenar os dados do grafo; e (iii) utilizar tabelas clusterizadas para agrupar as arestas em uma única página de disco também melhora desempenho para consultas recursivas, com a vantagem de exigir poucas alterações nas instruções SQL. No geral, especializar as tabelas T e E ajuda a reduzir a lacuna de execução eficiente de consultas em grafo em um SGBDR.

Palavras-chave: Consultas Recursivas, Consultas em Grafos, SGBDR, Abordagens de Armazenamento em Listas de Adjacências.

LIST OF FIGURES

Figure 1 – Two random models used to generate synthetic graphs.	40
Figure 2 – Property graph example representing the three storage schemas: (a) the RDF table; (b) binary tables; and (c) an n-ary table.	45
Figure 3 – Approaches for executing graph queries in an RDBMS, with their results (a), divided into: (b) recursive queries; (c) extraction approaches; and (d) graph layers or views.	48
Figure 4 – Schema proposed for G-FranC, highlighting the more relevant attributes in each table.	59
Figure 5 – The Top 4 largest communities identified in G-FranC dataset considering their quantity of criminal activities as weight.	60
Figure 6 – Neighborhood analysis over the vertex with the highest number of crime occurrences (black star), showing the regions around it up to 0.5 km (in orange), 1 km (in purple), and 2 km (in green).	61
Figure 7 – Overview of SHARq framework to reuse recursive join results among multiple graph queries.	62
Figure 8 – The execution time reductions for 5, 10, 15, and 20 combined SSSP queries observed with SHARq framework in 18 datasets, organized by their category in different columns.	71
Figure 9 – The execution time reductions for the combinations of SSSP, CC, and PR queries observed with SHARq framework in 18 datasets, organized by their category in different columns.	72
Figure 10 – Comparison between the overhead introduced by additional measurement columns in each <i>Recursive Join</i> operation and the total number of operations in SHARq framework.	74
Figure 11 – Graph example and its representation: (a) a conventional edge table, with one edge per row; and (b) the table Edge- k , with each edge e_k storing up to k edges in a single row.	76
Figure 12 – The DBLP dataset, representing the direct and indirect relationships between the researchers through their shared publications.	82
Figure 13 – The average elapsed times for the insert, update, and delete operations when using the conventional edge table (<i>i.e.</i> , with $k = 1$) and the table Edge- k using $k = 5, 10$, and 15	83

Figure 14 – The average processing time of the bulk loading processes varying k . The bulk loading process is performed following the steps: Sort, Cast, and Load.	84
Figure 15 – The dataset size reduction, in megabytes, achieved by the Edge- k table in both real and synthetic datasets while varying the k value from 1 to 50.	85
Figure 16 – The average SSSP query processing time in three scenarios for varying k in the table Edge- k : the DBLP dataset with the 2 nd , 3 rd , and 4 th iterations, and in both synthetic datasets varying their respective parameters.	85
Figure 17 – The highest SSSP query processing time reductions using Edge- k varying: (a) the number of iterations in the DBLP dataset; (b) $ V $ for Newman-Watts-Strogatz dataset; and (c) p^{Erdos} for Erdős–Rényi dataset.	86
Figure 18 – The average elapsed time for executing distribution degree queries comparing both Edge- k and conventional edge tables with the Neo4j database.	87
Figure 19 – The elapsed time comparison for the SSSP query execution regarding the Neo4j and the conventional edge and Edge- k tables, while varying the starting vertex and the number of iterations from 1 to 15.	89
Figure 20 – An example schema for storing graph data and its three physical representations regarding different Clustered Table’s physical designs explored.	91
Figure 21 – The statistics obtained from the <i>DIMACS</i> dataset, (a) vertex distribution degree, (b) histogram of the edge tuple sizes, and (c) structure sizes for the Clustered Tables.	94
Figure 22 – The average SSSP query processing time (a) and the number of buffer gets (b) over the <i>DIMACS</i> dataset employing Clustered Tables starting from a single point.	95
Figure 23 – The average query processing time of Egonet queries for the four evaluated Clustered Tables’ physical designs.	96
Figure 24 – The average query processing times of the SSSP query for $ T^0 = 5,000$, 10,000, 20,000, 50,000, and 100,000.	98

LIST OF ALGORITHMS

Algorithm 1 – Algorithm employed in SHARq framework for multiple graph queries. .	67
---	----

LIST OF SOURCE CODES

Source code 1 – Example of an SSSP query, expressed in Cypher language, executed in Neo4j.	44
Source code 2 – Recursive Query in PostgreSQL RDBMS using CTEs.	54
Source code 3 – Recursive Query in Oracle RDBMS using CTEs.	55
Source code 4 – PL/pgSQL procedure to compute the SSSP query in PostgreSQL RDBMS.	56
Source code 5 – Query for retrieving the set of patients in a city that are “far away” from the n hospitals $h_1, h_2, \dots, h_n$	64
Source code 6 – Query speeded up by simultaneously computing the shortest paths to the set of hospitals $\mathcal{H}$	64
Source code 7 – Adaptation of the recursive-step query when employing table Edge- k	80
Source code 8 – Creation of tables V and E using the Clustered Tables physical designs.	92

LIST OF TABLES

Table 1 – Examples of datasets modeled as graphs, showing their respective sets of vertices and edges.	32
Table 2 – Rank with the top 5 graph query types executed by both researchers and practitioners, with respective aliases and variations.	41
Table 3 – The Top 7 most frequent crime categories in the G-FranC dataset, with the number of occurrences and criminal activity examples for each category. . . .	59
Table 4 – The 18 datasets employed to evaluate the SHARq framework, divided into three categories: Healthcare, Social Network, and Urban Streets.	68
Table 5 – Characteristics of both real and synthetic datasets explored with table Edge- k	82

LIST OF ABBREVIATIONS AND ACRONYMS

APSP	All-Pairs Shortest Paths
CC	Connected Component
CTE	Common Table Expression
NoSQL	Not Only SQL
NULL	A value which is unknown or not applicable
PR	PageRank
RDBMS	Relational Database Management System
SSSP	Single-Source Shortest Path
ACID	Atomicity, Consistency, Isolation and Durability
CDM	Common Data Model from the OMOP
DDL	Data Definition Language, in SQL
DML	Data Manipulation Language, in SQL
EAV	Entity–Attribute–Value model
FK	Foreign Key to a Relation or Table
OMOP	Observational Medical Outcomes Partnership
PK	Primary Key of a Relation or Table
PL/pgSQL	Procedural Language for SQL in PostgreSQL RDBMS
RDF	Resource Description Framework
ROWID	Data type used for uniquely identifying where a row resides on the disk

LIST OF SYMBOLS

T — Temporary result table

G — Graph, represented as a pair of sets $G = \{V, E\}$

V — Vertex set, stored in the vertex table V

E — Edge set, stored in the edge table E

id — Unique identifier column of a table

v — Vertex from the vertex table V

e — Edge from the edge table E

id_{src} — Foreign key (FK) to the source vertex identifier id

id_{tgt} — FK to the target vertex identifier id

p^{Erdos} — Probability of creating a edge between a pair of vertices in the Erdős–Rényi model

X^{Newman} — Number of shortcuts defined in the Newman-Watts-Strogatz model

G' — Induced subgraph of G

V' — Subset of vertices $\subset V$

$degree_v$ — Degree of vertex v , corresponding to the number of vertices in its neighborhood

Q^Q — Common Table Expression acquired after a Recursive Query execution

T^i — Temporary result table T at i -th recursive query step

x, y — Random values employed in SQL query predicates

Q_f — Final query executed over the materialized table Q^Q

v^{start} — Starting vertex for an SSSP query

T' — Temporary result table of the next step in a Recursive Query

w — Weight value of an edge (also may refers to its distance or length)

c — Criminal activity stored in the crime table

$\mathcal{D}$ — Great-circle distance, corresponding to the distance between georeferenced elements

$\mathcal{R}$ — Earth's radius (*i.e.*, 6,371km)

max^{dist} — Maximum path length employed to limit the execution of a SSSP query

T^{SSSP} — Result table T from a SSSP query

$T_{v_i}^{SSSP}$ — Result table T from the SSSP query started from vertex v_i

t_{v_i, v_j}^{SSSP} — Tuple from the table $T_{v_i}^{SSSP}$, denoting the shortest path length from v_i to v_j

w — Measurement column employed in result table T

$T_{V'}^{SSSP}$ — Enriched table T, having the results from an SSSP query for each vertex in V'

d — Damping factor of a PageRank query, where $d \in \mathbb{R}$, $d \in [0, 1]$

$\mathcal{Q}$ — Set of Recursive Queries executed, with $\mathcal{Q} = \{ Q_1, Q_2, \dots, Q_n \}$

Q_i — Recursive Query $\in \mathcal{Q}$, where Q_i is either a SSSP, CC, or PR query

e_k — Row in the table Edge- k

k — Quantity of edges stored per row in table Edge- k

η_{rows}^k — Number of rows in table Edge- k

η_{NULL}^k — Quantity of NULL values in table Edge- k

$\eta_{exceeding}^k$ — Exceeding space occupied by table Edge- k

T^{UNNEST} — Result table of a Recursive Query after applying the UNNEST function

$n^{publications}$ — Number of shared publications between a pair of researchers

$cluster_size^{ith}$ — Cluster size defined over the i -th percentile of the graph degree distribution

CONTENTS

1	INTRODUCTION	29
1.1	Motivation	30
1.2	Objectives and Contributions	32
1.3	Manuscript Organization	34
2	GRAPH QUERY PROCESSING IN AN RDBMS	35
2.1	Relational Database Management Systems	35
2.1.1	<i>Processing Join Operations</i>	36
2.1.2	<i>Physical Data Organization</i>	38
2.2	Complex Networks and Graphs	39
2.2.1	<i>Graph Characteristics and Categories</i>	39
2.2.2	<i>Graph Queries</i>	41
2.2.3	<i>Graph Data Management and Storage</i>	43
2.3	Graph Management in RDBMSs	44
2.3.1	<i>Hybrid Storage Approaches based on Adjacency-Lists</i>	46
2.3.2	<i>Graph Query Processing in a RDBMS</i>	47
2.3.2.1	<i>Recursive Query Approaches</i>	47
2.3.2.2	<i>Extraction Frameworks</i>	50
2.3.2.3	<i>Approaches employing Graph Layers or Views</i>	51
2.4	Final Considerations	52
3	RECURSIVE QUERIES ANALYSIS	53
3.1	Recursive Query Definition	53
3.1.1	<i>Implementation using CTEs</i>	54
3.1.2	<i>Implementation using PL/pgSQL</i>	55
3.1.3	<i>Application Scenario example</i>	57
3.2	Sharing Join Operations in Recursive Queries	62
3.2.1	<i>Motivation for Sharing</i>	63
3.2.2	<i>The Common Join Operation</i>	65
3.2.2.1	<i>Adapted Algorithm for SHARq</i>	66
3.2.3	<i>Experimental Evaluation using SHARq</i>	67
3.2.3.1	<i>Application Scenarios for SHARq</i>	69
3.2.3.2	<i>Performance Evaluation</i>	70

3.2.3.3	<i>Trade-off when Including Extra Columns</i>	73
3.3	Final Considerations	74
4	ADJACENCY-LIST REPRESENTATIONS IN AN RDBMS	75
4.1	Tuple-Level Grouping: Edge-k	75
4.1.1	<i>The Edge-k Method and Table</i>	75
4.1.1.1	<i>Main Parameters for estimating the value of k in the table Edge-k</i>	77
4.1.2	<i>Graph Management Adaptations with Edge-k</i>	79
4.1.2.1	<i>Table Management and Bulk Loading</i>	79
4.1.2.2	<i>Graph Navigation Query Adaptation</i>	80
4.1.3	<i>Experimental Evaluation using the table Edge-k</i>	81
4.1.3.1	<i>Overhead in Insert, Update, Delete and Bulk Load Operations</i>	82
4.1.3.2	<i>Reductions in Storage Size and Query Processing Times</i>	84
4.1.3.3	<i>Comparison with Neo4j NoSQL Database</i>	87
4.1.4	<i>Final Remarks over Edge-k</i>	89
4.2	Block-Level Grouping: Clustered Tables	90
4.2.1	<i>The Clustered Tables Proposal and Methodology</i>	90
4.2.1.1	<i>Cluster Size estimation for employing Clustered Tables' physical designs</i>	92
4.2.1.2	<i>Application Scenario for Clustered Tables</i>	93
4.2.2	<i>Experimental Evaluation using Clustered Tables</i>	93
4.2.2.1	<i>Recursive Query starting from a Single Point</i>	95
4.2.2.2	<i>Egonet queries starting from Multiple Points</i>	96
4.2.2.3	<i>Recursive Query starting from Multiple Points</i>	97
4.2.3	<i>Final Remarks over Clustered Tables approaches</i>	97
4.3	Final Considerations	99
5	CONCLUSION	101
5.1	Contributions	101
5.2	Future Work	102
5.3	List of Publications	103
	BIBLIOGRAPHY	107

INTRODUCTION

Complex networks are present everywhere, from social networks to communication infrastructures, urban street organization, and protein interactions (BARABÁSI; PÓSFAL, 2016). These networks are represented as graphs, where the nodes (*i.e.*, entities) are mapped as vertices and their relationships as edges. The rapid growth of Web-based applications has led to an increasing volume of graph data (CORBELLINI *et al.*, 2017), and the amount of applications using graph structures to represent data has increased significantly in the last years. Such volume is large enough to not fit in main memory, compelling application developers to rely on disk-based approaches. There are many issues to solve beyond their storage, such as the need to guarantee data independence, sharing, integrity, security, and redundancy control, to name a few. Relational Database Management Systems (RDBMSs) provide infrastructure to support disk-based data management, with some useful features such as storage, data buffering, indexing, and query optimizations (ELMASRI; NAVATHE, 2016).

Since 1970, the Relational Model has been employed to meet the needs of data storage in the most diverse application domains (CODD, 1970). Examples of domains include enterprise information (*e.g.*, sales, accounting, human resources, and manufacturing), finance (*e.g.*, banking and credit card transactions), telecommunication, and airline reservations and scheduling (SILBERSCHATZ; KORTH; SUDARSHAN, 2005). The fundamental query language for the Relational Model is the SQL (Structured Query Language), covering all the data definition, manipulation, and retrieval in any RDBMS (GARCIA-MOLINA; ULLMAN; WIDOM, 2008). SQL is a generic language that can manage data from various sources to support decision-making based on data analysis and mining. However, they demand additional support to handle graphs and execute queries efficiently. This need led to specific implementations and variations of Database Management Systems (DBMSs) that do not follow the Relational Model, such as the NoSQL (Not only SQL) Systems (CATTELL, 2011).

Graph-Based NoSQL Systems are DBMSs that focus on performing queries and analyses over graphs (VICKNAIR *et al.*, 2010; KLEPPMANN, 2017). They represent graphs using

specialized structures, usually based on adjacency lists, to speed up operations that transverses the graph's vertices through its edges. They overcome the limitation imposed by RDBMSs because they usually do not perform any join operation to “navigate” the graph data by transversing its vertices. Examples of Graph-Based NoSQL Systems include Neo4j (VUKOTIC *et al.*, 2014) and GraphDB (GÜTING, 1994). Both systems employ graph structures and analysis techniques to insert, update, and execute queries over the topological structure of networks (MCCOLL *et al.*, 2014). In general, these Graph-Based NoSQL Systems have better performance when processing queries over the graph's topology, computing paths between vertices, or ranking their importance (e.g., ranking them through a PageRank algorithm or computing centrality metrics) (VICKNAIR *et al.*, 2010). However, RDBMSs still stands out in several other mining tasks over the data regarding association rules, classification, clustering, and so on (ELMASRI; NAVATHE, 2016).

Although RDBMSs were not specifically designed to compute queries that “navigate” a graph, they still play a role in managing and processing graphs (SAHU *et al.*, 2017; SAHU *et al.*, 2019). These queries are often represented as recursive queries (ORDONEZ, 2005; PRZYMUS *et al.*, 2010; CORTESI; HALDER, 2013; AL-AMIN; ORDONEZ; BELLATRECHE, 2018), characterized by computing repetitive join operations to perform a graph transversal (GAO *et al.*, 2014; FAN; RAJ; PATEL, 2015; JINDAL *et al.*, 2015). Several graph analyses were mapped as recursive queries to execute in RDBMSs (ZHAO; YU, 2017), and most RDBMSs, including DB2, Microsoft SQL Server, Oracle, and PostgreSQL support recursive queries. Join operations plays a central role in almost every SQL query plan (SCHUH; CHEN; DITTRICH, 2016), being studied and improved since its existence. Hence, this Ph.D. research aims at improving the performance of recursive queries by either reducing the number of join operations performed or using specialized storage approaches based on adjacency lists.

1.1 Motivation

The literature from the last years covers a range of optimizations for processing queries that “navigate” the graph data stored in an RDBMS. EmptyHeaded (ABERGER *et al.*, 2016; ABERGER *et al.*, 2017) and GraphGen (XIROGIANNOPOULOS; KHURANA; DESHPANDE, 2015; XIROGIANNOPOULOS; SRINIVAS; DESHPANDE, 2017; XIROGIANNOPOULOS; DESHPANDE, 2017) materialize the graph stored in an RDBMS to process their queries through third-party tools. GRFusion (HASSAN *et al.*, 2017; HASSAN *et al.*, 2018b; HASSAN *et al.*, 2018a) and Db2 Graph (TIAN *et al.*, 2019; TIAN *et al.*, 2020) create graph views to materialize paths using customized commands and executes queries over these paths. Regarding recursive query optimizations, there are approaches for reducing its intermediary size's results (ORDONEZ, 2005); partitioning methods for the edge table (GAO *et al.*, 2014; HONG *et al.*, 2017; SEO; AHN; IM, 2019); specialized data representations based on adjacency lists (SUN *et al.*, 2015; CHEN, 2013; AHMED; THOMO, 2017); in-memory graph process engines (MA *et al.*, 2016; ZHAO *et al.*, 2019); relational matrix algebra adaptations (DOLMATOVA; AUGSTEN; BÖHLEN,

2020); and the development of new relational operators (ZHAO; YU, 2017). The relevance of recursive queries as a field of study can be seen through several research articles (ORDONEZ; CABRERA; GURRAM, 2017; JINDAL *et al.*, 2015; FAN; RAJ; PATEL, 2015; SIMMEN *et al.*, 2014; PASSING *et al.*, 2017; FLORATOS *et al.*, 2018; CABRERA; ORDONEZ, 2017). Hence, the large quantity of research and studies over recursive queries motivated us to further improve recursive queries' performance.

Another motivation for this research refers to providing interoperability between queries that “navigate” through entities in the database and other queries regarding mining tasks and analyses (*e.g.*, grouping, classification, and clustering) over the same data. The alternatives for materializing the graph in a third-party tool or employing Graph-Based NoSQL Systems emerged to provide efficiency mainly for the category of queries that transverse the graph. However, in the era of data that we live in nowadays, providing efficiency for only one restricted category of queries is no longer enough. Researchers and practitioners usually cross data from several domains to acquire broader and more comprehensive information regarding an object of study. For example, given a study over disease spread in a city, some transversal queries may be:

- (i) *What is the closest hospital or healthcare facility to my current location?;*
- (ii) *Retrieve the most propitious place to build a temporary medical service station considering the critical regions in the city with high transmission rates.; and*
- (iii) *Search for all the citizens not immune to a given disease within up to 10 blocks from a point of interest, such as parks, museums, and others.*

Queries (i), (ii), and (iii) require materializing paths over the topology of a city, but they only provide an initial insight into the disease spread. It will only be possible to understand and better contain the spread when we complement the answer to these queries with detailed data from the affected citizens, such as grouping them by their age range and categorizing their vulnerabilities to more severe disease's cases, or by considering the available resources for treating the disease in hospitals or healthcare facilities. Moreover, recursive tasks are a fundamental part of Computer Sciences, being employed in a range of Machine Learning algorithms and are not limited to graph data (JANKOV *et al.*, 2020). Therefore, improving the execution of recursive queries without relying on materializing the graph in third-party external tools nor employing Graph-Based NoSQL Systems would help to execute broader and more complex analyses.

There is also an extensive range of data that can be analyzed as a graph (PAPALEXAKIS *et al.*, 2016). Table 1 exemplifies ten datasets that can be usefully modeled as graphs, showing possible mappings to sets of vertices and edges. For example, in a power grid dataset, the vertices may be the power stations, while the edges are the energy wires connecting them. For the medicine field, the vertices may be the diseases and symptoms observed in a group of patients, and the edges are their co-occurrence among these patients (ROTMENSCH *et al.*, 2017). For a

research network, the vertices are the researchers, while the edges can be either co-authorship in the same publication or the works cited between them. The other examples in Table 1 follow the same reasoning. Notice that there may be several related data to improve the studies performed, as previously discussed. For instance, regarding urban streets, finding only the shortest paths may not suffice because some paths can pass through dangerous or highly criminal areas in a city. Hence, searching simultaneously for the shortest and the safer paths provides more relevant analysis over the data (GALBRUN; PELECHRINIS; TERZI, 2016).

Table 1 – Examples of datasets modeled as graphs, showing their respective sets of vertices and edges.

Dataset	Vertices	Edges
Power grid	Power stations	Energy wires
Internet	Routers and Hubs	Data wires
Web	Web pages	Hyperlinks
Medicine	Diseases and Symptoms	Co-occurrence among patients
Urban Streets	Street intersections	Street segments
Social Networks	People	Friendship
Research Networks	Researchers or Authors	Co-authorship or citations
Aerial Transportation	Airports	Flights
Email Networks	Receivers and Senders	Emails
Protein Interactions	Protein	Interactions between them

Source: Elaborated by the author.

1.2 Objectives and Contributions

This Ph.D. research aims at answering the following question: “*How can we improve the performance of queries over graph data executed in a RDBMS?*”. Our main challenge regarding this research question refers to the complexity of efficiently executing graph queries in a RDBMS thought recursive queries. Accordingly, we propose methods and specialized data structures to optimize such queries’ execution. Due to the extensive range of studies, methods, and techniques over recursive queries, in this Ph.D. research, we propose the following thesis:

Thesis. *By specializing data structures (i.e., tables, indexes, and internal data representations) in a RDBMS using adjacency-based storage approaches, it is possible to improve the performance of graph queries that depends on the several join operations involved in recursive query executions.*

A recursive query includes an initial statement declaring a temporary table to store each iteration’s results. The temporary table T is first initialized, then iteratively joined with the table E representing the graph’s edges. At each iteration, the results in the table T converges to the

expected outcome for the graph query processed. Each join operation can be computationally costly because the table E denotes a many-to-many relationship in the RDBMS, and most of the recursive queries require several of these join operations. In order to achieve our goal of efficiently executing graph queries, we divide our Ph.D. research into two questions, each one focusing on specializing one of the tables employed in the recursive queries: the tables T and E .

Question 1. *Is it possible to improve the performance of executing multiple graph queries, such as Single-Source Shortest Path (SSSP), Connected Components (CC), and PageRank (PR), by combining them into a single recursive query which employs an enhanced result table T containing the result of each graph query in a distinct column?*

Enhancing table T consists of including extra measurement columns, one for each graph query. Therefore, we enable multiple computations simultaneously without re-evaluating the same join operations in the recursive query. Adding extra columns results in costlier join operations because the result table T has a larger size and requires multiple aggregation functions, one for each graph query. However, despite being individually costlier, we have fewer join operations executed when considering analyses that require multiple graph query results. Chapter 3 discusses the results from this trade-off of enhancing the table T focused on answering **Question 1**, showing improvements in the performance of executing either multiple SSSP queries and combining SSSP, CC, and PR queries. In this research, we enhanced table T through the SHARq (*SHARED Recursive queries*) framework, and the work [Scabora et al. \(2021\)](#) reports the contributions related to this question.

Question 2. *Is it possible to increase the recursive join operation performance when processing graph queries by enhancing the edge table E representation using adjacency-based storage approaches?*

Notice that enhancing table T focuses on processing multiple graph queries, not improving the performance of executing each individual join operation. **Question 2** from this Ph.D. research focuses on specializing the table E with adjacency-list storage approaches. The most straightforward way to perform this enhancement is by grouping the edges for each vertex's neighborhood. Consequently, we enable adjacency lists, or at least part of them, represented in the RDBMS. Chapter 4 presents the approaches proposed in this Ph.D. research to group the edges in the table E , focused on answering **Question 2** by improving the performance of the join operations executed in recursive queries. Two different grouping methods were proposed in this research. The former is the table Edge- k , which groups multiple edges from each vertex' neighborhood into a single tuple in the table. The works [Scabora et al. \(2017\)](#), [Scabora et al. \(2018\)](#) report all results related to this contribution. The latter method groups these edges into

the same disk page on the computer hard disk through clustered table physical designs. The work [Scabora *et al.* \(2020\)](#) reports the results related to this contribution.

1.3 Manuscript Organization

This chapter summarized the motivation, objective, addressed questions, and main contributions of this Ph.D. research. The remaining parts of this manuscript are organized as follows:

- Chapter 2 reviews the required background. We highlight some of the existing limitations found in the literature that we tackle in the next chapters;
- Chapter 3 extends the concept of recursive queries, presents an application scenario driven by the interoperability of graph data and other data in the RDBMS, and answers **Question 1** by proposing the SHARq framework to share the execution of multiple graph queries;
- Chapter 4 presents our contributions regarding enhancing the table *E* using adjacency-list storage approaches, answering the **Question 2** of this Ph.D. research; and
- Chapter 5 concludes this manuscript, reviewing the main contributions and resulting publications, besides a range of potential future works.

GRAPH QUERY PROCESSING IN AN RDBMS

This chapter presents the relevant background related to RDBMSs, complex networks represented by graphs, and the execution of queries and other analyses over graph data.

2.1 Relational Database Management Systems

A Database Management System (DBMS) is a general-purpose computer system whose objective is to enable users to define, build, manipulate, and share databases among different applications. Moreover, a DBMS allows the management and protection of any database for extended time periods (SILBERSCHATZ; KORTH; SUDARSHAN, 2005). The Relational Model employs a collection of relations to represent both the data itself and the relationships among them. A relation defines a set of attributes to describe its tuples, where each attribute has a unique name to identify it. Each tuple has a value for every attribute defined in the relation, and some specific attributes may also be assigned as a Primary Key (PK), a Foreign Key (FK), or both. The Relational Model is widely adopted in most existing database systems (SILBERSCHATZ; KORTH; SUDARSHAN, 2005; ELMASRI; NAVATHE, 2016), covering a wide range of applications, including the ones that managing graph data (SAHU *et al.*, 2017; SAHU *et al.*, 2019). Each relation is implemented as a table, where a column represents an attribute and a row represents a tuple (GARCIA-MOLINA; ULLMAN; WIDOM, 2008). In this manuscript, we adopt the following nomenclature to distinguish a concept from the Relational Model (the first term) and its implementation in a DBMS (the second term): relation and table, tuples and rows, and columns and attributes.

An RDBMS is a DBMS based on the Relational Model. Most data manipulation in an RDBMS is performed using a Structured Query Language (SQL) (GARCIA-MOLINA; ULLMAN; WIDOM, 2008; ELMASRI; NAVATHE, 2016). SQL has mainly two sublanguages, the

Data Definition Language (DDL) for commands over the schema definition, such as creating and deleting tables, and the Data Manipulation Language (DML), for manipulating the data stored in the table, such as inserting and removing rows. The primary operations defined by the DML sub-language include projection, selection, cartesian product, joining, renaming, grouping, ordering, and union (SILBERSCHATZ; KORTH; SUDARSHAN, 2005; GARCIA-MOLINA; ULLMAN; WIDOM, 2008). The standardization provided by the SQL language can be considered as one of the main reasons for the success of existing commercial RDBMSs (ELMASRI; NAVATHE, 2016). Moreover, by using SQL commands in the DDL sublanguage, it is also possible to define index structures over the tables in the schema to speed up DML operations' execution, especially for data retrieval tasks.

An index is any data structure that, given the values of one or more attributes from a relation, efficiently retrieves the records filtered by those values (GARCIA-MOLINA; ULLMAN; WIDOM, 2008). In particular, an index associates values from the attributes with a data type named ROWID, which uniquely identifies where a row resides on the disk. Employing an index to locate the desired tuples enables avoiding to access all the records in a table, which requires a large amount of disk reading operations. For example, the summary of this manuscript can be interpreted as an index structure. It allows the reader to quickly retrieve the location (*i.e.*, page) of any desired chapter or section. The indexes can be divided between primary (*i.e.*, grouped) and secondary (*i.e.*, not grouped) (SILBERSCHATZ; KORTH; SUDARSHAN, 2005). In the primary index, the tuples' order in the table is determined by the indexed values, limiting only one primary index per table. In a secondary index, the order of tuples is independent of the index structure, allowing the definition of multiple secondary indexes on the same table. Among the main indexing techniques implemented in RDBMSs, the following three types stands out (ELMASRI; NAVATHE, 2016): B^+ -tree, *Hash* and *Bitmap*.

2.1.1 Processing Join Operations

Processing join operations is essential to almost every query plan (SCHUH; CHEN; DITTRICH, 2016). These operations were extensively studied, improved, and reexamined by the database community. This section describes the three most common join operations (ELMASRI; NAVATHE, 2016; SCHUH; CHEN; DITTRICH, 2016), named Nested-Loop, Sort-Merge, and Hash-Join. The description considers the join operation between two tables, R and S , based on comparing the attribute r from table R with the attribute s from table S .

The Nested-Loop join operation, also called brute force join operation, does not require any previous knowledge of the tables R and S . It works with any comparison operator in the join condition, such as the equality (*i.e.*, $=$), the difference (*i.e.*, $\neq$), and the inequality operators (*i.e.*, $>$, $<$, $\geq$, and $\leq$). Such a comparison operator also refers to the predicate in the query execution. For each tuple in table R , the implementation scans the tuples present in table S employing the comparison operator and retrieving the combinations that satisfy the predicate. Regarding the

equality operator, for example, this join operation only retrieves tuples where $R.r = S.s$. A variation of this implementation is the index-based Nested-Loop, which uses only the index structure instead of accessing the table if that index has the columns required in the query.

In the Sort-Merge join operation, its implementation loads the disk pages from both tables R and S into the primary memory. The tuples on each disk page are then sorted by the attribute used in the comparison operator. By executing the comparison co-sequentially between the pair of disk pages, one from each table, the operation retrieves combinations of tuples that satisfy the comparison operator employed. Like the Nested-Loop join operation, index structures can be used instead of accessing the table, mainly because they are already sorted. The Sort-Merge join operation is efficient when the join attribute already orders the tuples in the table. That is, it is especially advantageous if combined with primary indexes. The problem refers to using the Sort-Merge join operation over secondary indexes, where the tuples can be physically distributed across multiple disk pages, thus requiring several random accesses.

The Hash-Join operation's implementation divides the tuples from tables R and S into chunks based on a hash function employed over the compared attributes, r and s , respectively. This operation is divided into build and probe steps. The build step scans the table R , for example, organizing its tuples into chunks using the hash function. The probing step scans the table S , combining its tuples with the tuples from table R whenever they are hashed into the same chunk and respects the comparison operator. It is important to notice that the Hash-Join operation only works with the equality operation (*i.e.*, $=$) since it relies on organizing the compared tuples into the same chunks. Despite being efficient, the main problem with this implementation lies in the fact that the primary memory may not be enough to store or the build step or the probe step's result, requiring the chunks' management in the disk, negatively impacting the performance of queries. Moreover, there are non-partitioned and partitioned Hash-Join operation categories. The former uses a global hash table, and the latter divides the data into partitions, building hash tables for every one of them in the build step.

Alternatively to these three join operation implementations, it is possible to store the result from the join operation, keeping them in a new table or materialized view (VALDURIEZ, 1987). If this new table or view is indexed, queries over it would be efficient. However, their occupied disk space can be enormous. Moreover, for the recursive queries explored in this research, keeping such a table or view updated may lead to its total reconstruction for each insert, update, or delete operation performed.

Notice that executing join operations is computationally costly independently of the join operation employed. Hence, our focus in this project is to employ specialized storage approaches to execute recursive queries. Following the hypothesis from this Ph.D. research (see Section 1.2), in **Question 1** we aim at reducing the number of join operations executed across multiple recursive queries. Regarding **Question 2**, our goal is to improve the performance of executing each join operation. Hence, our thesis contributions focus on reducing both quantities

and individual computational costs of these three join operations' execution.

2.1.2 Physical Data Organization

The three-schema architecture of an RDBMS describes its organization in three different conceptual levels (ELMASRI; NAVATHE, 2016). Each level is defined as follows, ordered from the highest to lowest conceptual levels:

- The External or View Level is the highest conceptual level and refers to the information about the schema and user's permission access. It defines the parts of the database's schema that a user, or a group of users, have access to manage and retrieve the data stored. All the other parts in the schema are hidden from these users;
- The Conceptual Level refers to the conceptual schema, describing the database's structure for a group of users. This level includes the description of entities (*i.e.*, tables), data types of each attribute, relationships between entities, operations performed by users, and all the constraints defined over the data (*e.g.*, FKs). This level is an intermediary level because it hides any detail about the physical storage structures; and
- The Internal or Physical Level refers to every physical storage structure employed by the RDBMS. It describes the location of tables and indexes in the disk with their access paths.

Based on the three-schema architecture, the RDBMS starts by converting a query performed at the external level to the conceptual level, validating schemes, and structures. Then, the RDBMS converts from the conceptual level to the physical level, where the data is stored, to retrieve it. Retrieving the data is made by random accesses on a disk (ELMASRI; NAVATHE, 2016). This process transfers the tuples, or rows, from the tables between the disk and main memory and takes place in units of disk blocks or pages. Monitoring and reducing this transference is essential in RDBMSs or any other system that accesses data from disk.

In this Ph.D. research, we propose specialized storage approaches that mainly affect the physical level, organizing the stored data based on adjacency-lists. However, there are also modifications at the conceptual level, mainly because our proposal introduces additional columns to the tables. Take, for example, the use of Clustered Tables¹. A Clustered Table groups the data from one or more tables that share the same attribute. The attributes used for grouping are named grouping key, and they are used to group tuples having the same values of these attributes into a specific disk page or block. As a result, any join operation over clustered tables based on the grouping key can be efficiently performed because the data blocks that will be combined tend to be the same, or at least to be close on the disk. We performed experiments using Clustered Tables as specialized storage approaches for optimizing recursive queries, as detailed in Chapter 4.

¹ <<https://docs.oracle.com/database/121/ADMIN/clustrs.htm>>

2.2 Complex Networks and Graphs

Complex networks help in modeling data from several domains, such as networks of social interaction, product recommendation, communication infrastructure, and urban street organization (BARABÁSI; PÓSFAL, 2016). In Table 1 we reviewed some examples of these modellings. Complex networks are usually structured in terms of two sets, the vertices (*i.e.*, entities, or nodes) and edges (*i.e.*, their relationships, or links). Both sets can be stored and managed using tables in an RDBMS. Moreover, RDBMSs can also be used to manipulate these tables among various users and applications (ELMASRI; NAVATHE, 2016).

A graph is formally represented as a pair of sets $G = \{V, E\}$. On the one hand, Table V stores the entities in set V , in which every $v \in V$ have an identifier id and other properties that define them, such as its name. On the other hand, Table E stores the relationship set E , defining the links between the entities in V . Each e in E has a pair of foreign keys, id_{src} and id_{tgt} , denoting the identifiers from its source and target vertices in V , respectively. Additional properties to the edges include their weight or distance, and these properties are important to a range of analyses, such as the ones described in Section 2.2.2. This manuscript employs the terms complex networks and graphs interchangeably since a graph represents a complex network.

2.2.1 Graph Characteristics and Categories

There are multiple statistical models to study graphs (HAN; KAMBER; PEI, 2011). The most popular are the scale-free and random graph models. The scale-free model states that the graph follows a power-law distribution. Power law distribution, also known as the heavy-tailed distribution, follows the 80/20 rule of the Pareto principle (BARABÁSI; PÓSFAL, 2016). In this rule, approximately 80% of the vertices have a low quantity of neighbors, usually one or two, while only 20% of them concentrate most of the graph's edges. For the random graph model, there exists the Erdős-Rényi e Newman-Watts-Strogatz models. Figure 1 illustrates both of these models. The Erdős-Rényi model, in Figure 1(a), considers a probability p^{Erdos} to create an edge between every pair of vertices in the graph (ALBERT; BARABÁSI, 2002). The Newman-Watts-Strogatz model, in Figure 1(b), initially connects the vertices in a ring, then creates a number X^{Newman} of shortcuts in between pairs of vertices in the ring (NEWMAN; WATTS, 1999).

Despite these graph models, graphs are also divided into a range of categories or types, and, in this section, we summarize them. The first division refers to directed and undirected categories of graphs (BARABÁSI; PÓSFAL, 2016). In a directed graph, if there is an edge from a vertex v_i to a vertex v_j , with v_i and $v_j \in V$, not necessarily there is an edge in the opposite way. A typical example of this category is sending emails, where person i sends an email to person j . Here, the email is the edge between the two persons i and j , but not necessarily there is an edge from person j to person i . In an undirected graph, every edge is symmetric. That is, if there is an edge from v_i to v_j , there will be an edge from v_j to v_i having the same properties (*e.g.*, weight or

distance). A typical example of this category is the “friendship” relationship in social networks, in which if a person i is a friend of person j , the opposite is also true. The standard approach to represent an undirected graph in an RDBMS is by appending edges to the table E with their source and target identifiers inverted.

Figure 1 – The two random models used to generate synthetic graphs: (a) Erdős–Rényi model creates edges between each pair of vertices based on a probability p^{Erdos} ; and (b) Newman-Watts-Strogatz model organizes the vertices in a ring and creates X^{Newman} shortcuts between each pair of vertices in the ring.

Source: Elaborated by the author.

The second category or division refers to a graph being bipartite or non-bipartite (BARABÁSI; PÓSFAL, 2016). The vertex set V can be split into two disjoint subsets in a bipartite graph, V^A and V^B . In this category of graph, edges can only exist between a vertex $v_a \in V^A$ and a vertex $v_b \in V^B$. That is, in a bipartite graph, no edge can be created between two vertices from the same subset. They only can originate from one subset to another, and vice versa. An example of this category is a teaching graph modeling professor and students. Both professors and students are persons, being the entities (*i.e.*, vertices) of the graph. They are not only disjoint groups, but also the “teaching” relationships are only possible between teachers and students. A bipartite graph can be projected into non-bipartite ones by defining edges between pairs of vertices from V^A if they both connect to the same vertex in V^B . In the example, the projection refers to creating a graph by connecting students that learned from the same professors.

Based on subsets of vertices, we can also create induced subgraphs from them (FAN; WU; XU, 2016). The induced subgraph G' is composed of a subset of vertices V' and filtering only the edges whose both endpoints are in V' . The subset of vertices V' may be retrieved based on a range of analysis. Considering the examples from this section, we can filter professors from a specific area or famous people in a friendship social network to focus on their relationships. In particular, induced subgraphs may be employed for scalability evaluation since they can generate varying-size graphs from the same domain. Moreover, Table 4 presents some examples of induced subgraphs from social networks, medical domains, and georeferenced graphs used in the scalability experiments realized in this project.

Finally, a particular type of graph worth mentioning is the Georeferenced Graph. For example, we refer to a georeferenced graph when set E represents a set of street segments (*i.e.*, relationships or edges), and set V is the set of street intersections (*i.e.*, entities or vertices). Each vertex $v \in V$ has, in addition to its identifier id , the pair of georeferenced coordinates latitude and longitude. Each edge $e \in E$ has at least a distance property that corresponds to the street segment's length, usually measured in meters. The subcategory of georeferenced graphs describes street meshes of cities and urban regions, being directed when referring to vehicles and undirected for pedestrians.

2.2.2 Graph Queries

Graph queries are computations executed over the graph to retrieve its statistics, navigate through the vertices using their relationships (*i.e.*, edges), or analyzing some of its characteristics. Table 2 shows a rank containing the top 5 query types most performed over graphs, as described in [Sahu et al. \(2017\)](#) and [Sahu et al. \(2019\)](#). The authors acquired this rank in an extensive study that consisted of an online survey, exploring emails, source repository, papers, and interviews with researchers and practitioners. This section describes these graph query types, which are explored during this Ph.D. research, as reported in both Chapters 3 and 4.

Table 2 – Rank with the top 5 graph query types executed by both researchers and practitioners, with respective aliases and variations.

rank	graph query	aliases
1	Finding Connected Components	SCC and WCC, or CC in general
2	Neighborhood queries	Egonet, Distribution Degree
3	Finding short/shortest paths	Reachability, SSSP
4	Subgraph matching	n -cliques
5	Ranking and retrieving centrality scores	PR, Betweenness, and Completeness

Source: Adapted from [Sahu et al. \(2019\)](#).

Before discussing the graph queries, we first need to define paths within a graph. A path $p = v_1 \xrightarrow{p} v_n$ is a sequence of non-repeated vertices, where each vertex is connected to the next vertex along the path by a relationship or edge ([BARABÁSI; PÓSFAL, 2016](#)). We then define a component as a subset of vertices V' with all its pairs of vertices connected by a path. In an undirected graph, each component is called Weakly Connected Component (WCC), and in a directed graph, these components are named Strongly Connected Components (SCC). Finding Connected Components (CC) is placed first on the graph query types rank because it helps in many analyses over graph data, such as identifying clusters or communities in the graph.

For the Neighborhood query type, we cover the Egonet and Distribution Degree queries. An Egonet query retrieves the neighborhood of a vertex $v \in V$, corresponding to the vertices directly connected to the given vertex v by only one relationship or edge ([MCCOLL et al., 2014](#)).

The degree of a vertex $degree_v$ is the size of its neighborhood, measured by the number of vertices in it. In an undirected graph, we retrieve all vertices which have an edge originated from or targeting the vertex v (BOCCALETTI *et al.*, 2006). In a directed graph, the vertex's degree is divided into in and out-degrees. The out-degree refers to retrieving the vertices based on the edges originated from the vertex v . Conversely, the in-degree considers the edges whose destination is the vertex v . A usual approach to combine both in and out-degree is by summing them. Moreover, evaluating the proportion of every vertex's degree in a graph leads to the Distribution Degree query, which is mostly used to validating if the graph follows a power-law distribution.

The third query type in the rank refers to finding short and shortest paths. We obtain the path size by counting the number of vertices in it. It is important to notice that, in a graph, there will probably be multiple paths connecting each pair of vertices. Thus, finding the shortest path refers to finding the path with minimal size. Searching for these paths leads to a Reachability query, which checks if there is at least one path between a pair of vertices. If there is a distance, weight, or length property in the edge, we define the path length as the sum of the corresponding property value in the path. Two common queries here are the Single-Source Shortest Path (SSSP), which calculates the shortest path's length from a given vertex $v \in V$ to every other vertex in the graph, and All-Pairs Shortest Paths (APSP), computing the shortest paths between every pair of vertices in the graph. The Bellman-Ford and Dijkstra's algorithms are the usual approaches for computing the SSSP queries, while the Floyd-Warshall and Johnson's algorithms are the preferred approaches for the APSP queries (CORMEN *et al.*, 2009).

The fourth place in the rank performs subgraph matching, corresponding to finding induced subgraphs with similar topology. A straightforward example is the triangle counting (TSOURAKAKIS *et al.*, 2011), which retrieves the number of triangle-induced subgraphs in the graph. More generally, we consider a triangle-induced subgraph as a subgraph composed of three vertices, and each one of them being connected to every other vertex in the induced subgraph. Considering that a "clique" is an induced subgraph where every vertex connects to each other, a triangle is a particular example of a 3-clique query (KARDES; GUNES, 2010). This kind of query can be expanded to 4-clique, 5-clique, and so on, which we refer to as n -clique queries.

Lastly, there are queries that retrieve a measurement value for each vertex or edge in the graph. These measurements are relevant to recommendation systems since they leverage the vertex's significance in the graph (KIM; CANDAN; SAPINO, 2016). The PageRank (PR) calculates the vertex importance based on the importance of the vertices in its neighborhood. This importance is determined by the probability of visiting this vertex while performing several random navigations over the graph's vertices through its edges. Other measures for quantifying the relevance of a vertex in the graph include the Betweenness and Closeness centrality scores (BRANDES, 2008).

2.2.3 Graph Data Management and Storage

To execute graph queries efficiently over a range of graph data, many researchers and companies invested in non-relational forms of storing data, resulting in NoSQL alternatives (HECHT; JABLONSKI, 2011; CORBELLINI *et al.*, 2017). NoSQL systems usually hold semi-structured and unstructured data, mainly focusing on distributing the data storage and avoiding processing complex join operations required when processing graph queries in a RDBMS. They also avoid following a strict schema for structuring the data, imposed by the relational model, which results in not efficiently storing and managing semi-structured data (MOHAN, 2013). The NoSQL alternatives are divided into four categories (LEAVITT, 2010; CATTELL, 2011; CORBELLINI *et al.*, 2017): Key-Value, Column-oriented, Document-oriented, and Graph-oriented.

Graph-oriented NoSQL Systems, also referred to in this manuscript as NoSQL databases, store data in a graph-like structure, usually representing the edges as direct references to the disk data instead of using the list of edges as in RDBMSs (CORBELLINI *et al.*, 2017). An example of such NoSQL databases is the Neo4j², an open-source project based on Lucene indexes to store and access data from vertices and their relationships. Studies revealed that Neo4j is more efficient than a relational database (*i.e.*, MySQL) when dealing with traversal queries, *i.e.*, queries looking for shortest paths with lengths bigger than three (GROLINGER *et al.*, 2013; VUKOTIC *et al.*, 2014; VICKNAIR *et al.*, 2010). However, it is essential to mention that the performance of Neo4j is negatively impacted by the size of the query result, which can achieve huge sizes for large graphs and path lengths (VUKOTIC *et al.*, 2014). Other examples of Graph-oriented NoSQL databases includes InfiniteGraph³, InfoGrid⁴, HypergraphDB⁵, and AllegroGraph⁶.

The primary language to manipulate and retrieve graph data from the Neo4j graph-oriented NoSQL database is named Cypher⁷. Source code 1 exemplifies a SSSP query expressed in Cypher language⁸. It starts in Line 1 with a clause MATCH, which sets the vertex v_i , the vertex with the identifier $id = 'i'$, as the source of the query. Then, Line 2 calls the algorithm for computing shortest paths starting from v_i , using the property named cost as the distance or weight of the edge when computing the path length. Line 3 returns the vertices found at the end of each path, with their respective path length value. Finally, Line 4 returns these results, renaming the vertices found as destination, and the path length as distance.

Another alternative to RDBMSs and NoSQL systems are the new database systems that have emerged named NewSQL (STONEBRAKER, 2012). NewSQLs are systems that follow the Relational Model and are prepared for distributed architectures, always maintaining the

² <<https://neo4j.com/>>

³ <<http://www.objectivity.com/products/infinitegraph/>>

⁴ <<http://infogrid.org/trac/>>

⁵ <<http://hypergraphdb.org/>>

⁶ <<https://franz.com/agraph/allegrograph/>>

⁷ <<https://neo4j.com/developer/cypher/>>

⁸ <<https://neo4j.com/docs/graph-algorithms/current/labs-algorithms/single-source-shortest-path/>>

transactional properties ACID (*i.e.*, Atomicity, Consistency, Isolation and Durability) of an RDBMS (CORBELLINI *et al.*, 2017). In other words, they allow the RDBMS to have horizontal scalability, maintaining the fault tolerance provided by NoSQL (GROLINGER *et al.*, 2013) systems. Examples of the NewSQL system include VoltDB⁹, NuoDB¹⁰ and Clustrix¹¹.

```
1: MATCH (vi : V {id : 'i'})
2: CALL algo.shortestPath.deltaStepping.stream(vi, 'cost', 3.0)
3: YIELD vertexId, distance
4: RETURN algo.asNode(vertexId).id AS destination, distance
```

Source code 1 – Example of an SSSP query, expressed in Cypher language, executed in Neo4j.

Among these examples, VoltDB stands out, being considered much faster for managing data than several RDBMSs (*e.g.*, PostgreSQL, MySQL, DB2, and SQL Server) in a range of applications (STONEBRAKER; WEISBERG, 2013). VoltDB stores all data in main memory, without the need to perform buffer management, but its performance may degrade without sufficient primary memory. That is, the NewSQL alternative may not perform well when it is required to perform swap operations to disk. For processing graph queries and managing graph data, we highlighted VoltDB because of an existing proposal named Graph Relational Fusion (GRFusion) (HASSAN *et al.*, 2017; HASSAN *et al.*, 2018b; HASSAN *et al.*, 2018a). We provide the details of the GRFusion proposal in Section 2.3.2.3 while describing the alternatives to process graph queries in RDBMSs.

2.3 Graph Management in RDBMSs

In the previous sections, we covered the topics referring to RDBMSs in Section 2.1, and graph management and computations in Section 2.2. This section combines these two topics by describing the main methods and approaches to manage graph data in RDBMSs.

There are three main storage schemas for graph data management employing RDBMSs: (a) triple store schema; (b) binary tables; and (c) n-ary tables (LEVANDOSKI; MOKBEL, 2009). The triple store schema (a) represents the data from the Resource Description Framework (RDF), stored following with the Entity–Attribute–Value model (EAV). RDF organizes data as triples <subject,property,value>. The subject is the identifier *id* of the entity, the property represents an attribute of the entity, whose value is denoted by *value*. In this schema, both attributes and relationships between the entities are described as properties (NEUMANN; WEIKUM, 2010). Binary tables (b) consists of grouping each distinct property value into a

⁹ <<https://www.voltodb.com/>>

¹⁰ <<https://www.nuodb.com/>>

¹¹ <<https://www.clustrix.com/>>

specific table. Multiple binary tables are then named by the property value, and only require two columns: subject and value. The binary tables correspond to storing column-oriented tables, requiring to join tuples from different property tables, using their respective subject value, to retrieve all the properties from the entities stored. Storing these multiple properties as different columns in the same table leads to the n-ary representation (c) (LEVANDOSKI; MOKBEL, 2009).

Figure 2 illustrates an example of a property graph with two entities v_1 and v_2 . Each entity's property may include its name, age, and the university where he or she works. For the triple store schema in Figure 2(a), we illustrate the respective RDF table, where each triple defines a different property in a distinct tuple. Figure 2(b) illustrates the binary table schema, in which the properties name, university and age are in separated tables. Figure 2(c) shows the n-ary representation of the property graph, organizing the properties into different columns. Notice that this last representation may lead to empty columns, represented using the NULL value.

Figure 2 – Property graph example representing the three storage schemas: (a) the RDF table; (b) binary tables; and (c) an n-ary table.

Source: Elaborated by the author.

Storing the edges as a property in the n-ary representation may result in sparse tables. Hence, the tables are usually separated into the vertex V and edge E tables. The table E is implemented as a many-to-many relationship table. It follows the Edge List (CORMEN *et al.*, 2009) representation, storing each edge in a single row. There are other representations for edges, such as Adjacency Matrix (MARUHASHI *et al.*, 2012) and Adjacency-List (CORMEN *et al.*, 2009). Each of these representations leads to different computational complexities to process

graph queries and other analyses. Remarkably, the Adjacency-List representation is considered the most efficient for retrieving a vertex neighborhood, operation pretty common while executing graph queries since it already provides them organized in a proper list. Therefore, in this Ph.D. research, we focused on enriching the table E with adjacency-list storage approaches, as we highlighted in our hypothesis and research questions (see Section 1.2).

2.3.1 Hybrid Storage Approaches based on Adjacency-Lists

This section details the hybrid approaches combining the Edge Lists and Adjacency-List storage approaches in RDBMSs. The straightforward approach is to store the Adjacency-List in a column of vector or array data type (CHEN, 2013). In this approach, each tuple in the table stores the neighborhood's edges in a single column. The authors that proposed it focused on scenarios in which all vertices have a small $degree_v$ value, which is not true considering that the graph usually follows a power-law distribution (see Section 2.2.1). Moreover, the authors did not explore scenarios where the neighborhood column size is larger than the disk page size configured for the RDBMS (see Section 2.1.2). However, such a straightforward approach does not meet the First Normal Form (1NF) property from the Relational Model (ELMASRI; NAVATHE, 2016) because it is storing a multi-valued attribute. Additionally, the authors also did not explore how to adapt the graph query execution to employ this hybrid representation.

Another hybrid approach is the DB2RDF (BORNEA *et al.*, 2013), which proposes to group sets of properties into the same columns for the n-ary table representation. DB2RDF limits the n value in the n-ary table representation to a maximum of k properties per row. This strategy focused on minimizing the number of NULL values, such as exemplified with the property age in Figure 2(c). If an entity has more properties than k , they are split into several rows repeating the same value in the subject column. Since the same column can store distinct properties, instead of only having the column value, each property is represented by a pair of columns denoting its property name and the respective value. A graph coloring algorithm is employed to avoid overloading columns with predicates that usually co-occur together, thus using a hash function to determine which columns will store each property value. The analysis from the authors focused on queries from four benchmarks, aiming at comparing the query optimizations performed by SPARQL¹² and by the RDBMS (*i.e.*, internally optimizing the SQL query). However, these queries primarily focused on retrieving properties from the entities, not exploring the paths in these graphs as required for the graph queries explored in this research (see Section 2.2.2).

The SQLGraph (SUN *et al.*, 2015) extended DB2RDF by incorporating columns storing JavaScript Object Notation (JSON). SQLGraph proposes a schema that employs relational tables to store the adjacency information from the graph's vertices. Both vertex and edge properties were stored into separated tables, one for the vertices and the other one for the edges. Each group of properties per-vertex or per-edge were stored into a single JSON object, identified by its *id*

¹² <https://www.w3.org/TR/rdf-sparql-query/>

value. If the graph has multiple edge sets, each with a different semantic, they are all stored into a single edge table in the SQLGraph proposal. SQLGraph also mixed the SQL and Gremlin¹³ programming languages to perform graph traversals and update operations. This proposal’s main problem is storing the edge’s properties in a separated table, requiring several lookups in it while computing the shortest paths in a SSSP query, for example.

Despite not being a hybrid storage approach for an RDBMS, the GraphChi framework (KYROLA; BLELLOCH; GUESTRIN, 2012) is a disk-based system for efficiently computing graph algorithms. GraphChi splits the edges into intervals, namely shards, where each shard stores the edges sorted by their source vertex identifier id_{src} . The framework requires that every shard fits entirely in memory. GraphChi does not follow an RDBMS-based approach. However, it motivates our research to adapt its specialized storage approach to an RDBMS. We explored this idea by employing it with the Clustered Tables physical design (see Section 2.1.2) and reported both the proposal and results in Section 4.2.

All in all, this section reviews the main topics for the **Question 2** of this Ph.D. research. We tackled these hybrid representations with our proposals and analyses described in Chapter 4.

2.3.2 Graph Query Processing in a RDBMS

Figure 3 illustrates the management and execution of queries over graph-like data stored in an RDBMS. Figure 3(a) exemplifies some expected outputs from these queries, such as exploring vertex properties, retrieving shortest-paths information, ranking and calculating graph metrics, investigating communities, among others. Figure 3(b) refers to executing these queries using recursive queries, and it is detailed in Section 2.3.2.1. Figure 3(c) denotes frameworks to extract the graph data from the RDBMS, performing the queries using customized implementations employing, for example, Python or C/C++. Section 2.3.2.2 describes these frameworks. Figure 3(d) shows approaches that create a graph layer or view over the data, executing the graph queries directly over it. Section 2.3.2.3 details the existing graph layer or view approaches.

2.3.2.1 Recursive Query Approaches

Recursion is a fundamental feature for many programming languages (ORDONEZ; CABRERA; GURRAM, 2017; PRZYMUS *et al.*, 2010). In SQL, recursive queries are primarily expressed employing Common Table Expressions (CTEs). CTEs have been part of the SQL standard since 1999 (FLORATOS *et al.*, 2018), and it is supported by most of the RDBMSs such as Microsoft SQL Server, PostgreSQL, SQLite, and Oracle. However, it is not currently available in a range of columnar DBMSs (ORDONEZ; CABRERA; GURRAM, 2017), such as MonetDB, Vertica, and C-Store. In general, a CTE defines a temporary result table manipulated by SQL statement. Therefore, a recursive CTE is a table containing the results of repetitively joining the

¹³ <<https://tinkerpop.apache.org/gremlin.html>>

edge table E with the temporary result table from the previous iteration. The quantity of join operations performed in a CTE resulting table corresponds to the max path size found so far in the graph (see Section 2.2.2).

Figure 3 – Approaches for executing graph queries in an RDBMS, with their results (a), divided into: (b) recursive queries; (c) extraction approaches; and (d) graph layers or views.

Source: Elaborated by the author.

A recursive query is one of the most challenging query classes, and it demanded several studies to improve its execution performance (GAO *et al.*, 2011; GAO *et al.*, 2014; FAN; RAJ; PATEL, 2015; JINDAL *et al.*, 2015; ORDONEZ, 2005; BURZANSKA; STENCEL; WISNIEWSKI, 2009; ORDONEZ, 2010; ORDONEZ; GURRAM; RAI, 2014; FLORATOS *et al.*, 2018; ZHAO; YU, 2017). These queries are important in the context of this Ph.D. research since they enable executing, for example, the following graph queries through SQL commands in a RDBMS (CABRERA; ORDONEZ, 2016; CABRERA; ORDONEZ, 2017): Reachability, SSSP (using the Bellman-Ford algorithm), Weakly Connected Components (WCC), and PageRank.

FEM (Frontier, Expand and Merge) framework (GAO *et al.*, 2011; GAO *et al.*, 2014) is one of the earliest works that translate analytical graph queries into SQL statements. The FEM framework creates a temporary result table for storing the query results and focuses on processing SSSP queries. Its approach follows the Bellman-Ford algorithm, iteratively expanding

the set of visited vertices and finding new vertices or shortest paths to previously founded ones. It processes each recursive step by following three steps: (F) fetches the frontier vertices in the list of previously visited vertices; (E) expands the frontier vertices; and (M) merges the newly expanded vertices with the visited ones. Its optimizations include performing a bi-directional search and partitioning the edge table based on its weight or distance property used for the path lengths. However, the bi-directional search requires inputting both starting and ending vertices in the query for shortest paths, which can be too restrictive for general SSSP queries. Additionally, the edge table partitioning relies on distributed processing to speed up the execution performance since the expansion step (E) employs all the partitions in each recursive step.

Extending the FEM framework, the Grail (Graph Analysis in Legacy Systems) framework (FAN; RAJ; PATEL, 2015), as well as its adaptation for the Vertica¹⁴ RDBMS (JINDAL *et al.*, 2015), follows a similar approach. The Grail framework included the PageRank and WCC graph analytic algorithms to execute with SQL statements. It uses Giraph and GraphLab to coordinate its asynchronous execution in a distributed architecture. The framework uses messages to share the temporary result table among the different machines. The adaptation for the Vertica RDBMS (JINDAL *et al.*, 2015) follows the same idea as the Grail framework of coordinating the execution and exchanging messages. However, it focuses on minimizing the exchanged messages and on optimizing the query execution plan created by Vertica.

Notice that the main optimizations on recursive queries are related to reducing the intermediary result tables as soon as possible and to using index structures to optimize the join operations performed. Regarding reducing the size of intermediary results, the literature provides two main rules (ORDONEZ, 2005; BURZANSKA; STENCEL; WISNIEWSKI, 2009; ORDONEZ, 2010; ORDONEZ; GURRAM; RAI, 2014):

- Early filtering the selection conditions: if the query has any WHERE clause, the predicates in it need to be evaluated as soon as possible in a query employing recursive CTEs; and
- Eliminating duplicate rows: whenever the expansion step finds new vertices or paths, they need to be reduced as soon as possible. In a SSSP query, for example, this rule refers to removing paths that already have been detected that are longer than other paths found in the intermediary result table.

Incorporating these rules to recursive CTEs led to the WITH+ proposal (ZHAO; YU, 2017). The WITH+ introduces four relational algebra operators to efficiently execute recursive queries: MV-join, MM-join, anti-join, and union-by-update. The most relevant operator for recursive CTEs is the union-by-update because it asserts the duplicate row elimination rule. It overcomes the main problem of CTEs, limited by only appending new results to the intermediary result table, by enabling their update instead. Therefore, WITH+ proposal implemented ten

¹⁴ <<https://www.vertica.com>>

graph algorithms using the relational algebra: SSSP, CC, PageRank, HITS, TopoSort, K-core, Maximal-Independent-Set, Label-Propagation, Maximal-Node-Matching, and Keyword-Search. However, their contribution does not consider specialized storage approaches focused on this Ph.D. research. Moreover, we also observed similarities between the SSSP, CC, and PageRank query types presented, which we used for optimizing their execution as explored in Section 3.2.

Lastly, the SQLoop (FLORATOS *et al.*, 2018) also enforces the duplicate row elimination rule by proposing the ITERATIVE operation. The ITERATIVE operation replaces the RECURSIVE one by updating the values instead of only appending them. Such a replacement is similar to the union-by-update operator in the WITH+ proposal. The SQLoop also parallelizes the query execution by partitioning the intermediary result table and executing the query over each of them using different database connections. Their limitation is similar to the WITH+ proposal by not considering specialized storage approaches as proposed in this Ph.D. research.

2.3.2.2 Extraction Frameworks

GraphGen is a framework for extracting the graph-like data from an RDBMS using a datalog-like language (XIROGIANNOPOULOS; KHURANA; DESHPANDE, 2015; XIROGIANNOPOULOS; SRINIVAS; DESHPANDE, 2017; XIROGIANNOPOULOS; DESHPANDE, 2017). It materializes the graph in main memory using a Compressed Sparse Row (CSR) representation. GraphGen also employs a series of in-memory graph condensed deduplications for further compressing its representation. It then executes graph queries over the CSR representation, such as the SSSP, CC, and PageRank queries. The framework employs the NetworkX¹⁵ Python library for executing these queries. However, the authors stated that it could be adapted to other libraries or programming languages as well.

EmptyHeaded (ABERGER *et al.*, 2016; ABERGER *et al.*, 2017) is a graph extraction engine over data stored in a RDBMS also using a datalog-like language. The EmptyHeaded engine translates every SQL query to a Generalized Hypertree Decomposition (GHD), which is a Directed Acyclic Graph (DAG) that represents a query plan for the worst-case optimal join algorithm. It proposes a new join engine architecture designed to exploit the low-level layouts necessary to increase Single-Instruction Multiple Data (SIMD) parallelism. The queries supported by EmptyHeaded are the SSSP, PageRank, 3-clique (*i.e.*, triangle), and 4-clique. EmptyHeaded employs C++ programming language to manage the GHD and executed the graph queries utilizing it.

Other extraction frameworks are Ringo (PEREZ *et al.*, 2015), pgRouting (MILER; MEDAK; ODOBAŠIĆ, 2014), and Apache AGE¹⁶. Ringo is similar to the GraphGen framework by representing the graph in memory. It extracts the graph into an in-memory interactive graph analytical system, implemented in Python and C++ programming languages. The difference

¹⁵ <<https://networkx.github.io/>>

¹⁶ <<https://age.apache.org/>>

between Ringo and GraphGen is that the former employs the Stanford Network Analysis Platform (SNAP)¹⁷ to execute the graph queries. pgRouting is a shortest-path implementation for the PostgreSQL using its extension named PostGIS¹⁸. It extracts the graph using a predefined SQL query to fetch the data from the table E and materialize it in memory to execute the query over it. Finally, Apache AGE is a graph extension architecture yet under development, based on AgensGraph¹⁹ to parse, transform, optimize and execute graph queries written in Cypher over data stored in PostgreSQL.

2.3.2.3 Approaches employing Graph Layers or Views

GRFusion (HASSAN *et al.*, 2017; HASSAN *et al.*, 2018b; HASSAN *et al.*, 2018a) is a materialized graph view over the VoltDB NewSQL DBMS, as previously presented in Section 2.2.3. Its objective is to compute and materialize paths, storing them in a view. Such a view is employed to solve only the graph queries for finding short/shortest paths (see Table 2). To compute this view, GRFusion defines an in-memory structure with ROWIDS pointing to the data in the tables V and E . All computations are executed using structures implemented in C++ programming language, adapted into the source code of VoltDB. Once GRFusion creates a materialized view, it is possible to perform all DML commands over it, such as selection, projection, and especially join. The join operation provides interoperability of this view with other tables in the VoltDB, a non-existent feature in the extraction approaches discussed in the previous section. One of its main limitations is that GRFusion only answers one type of graph query: the Reachability and SSSP queries. Moreover, it relies on memory to efficiently manage its C++ structures while computing the graph queries, being problematic for large graph data.

DB2 Graph (TIAN *et al.*, 2019; TIAN *et al.*, 2020) is a graph layer over the IBM Db2TM RDBMS. It introduces a polymorphic table function named `graphQuery`, which computes the graph queries and materializes their results in a table. Afterward, it is possible to perform all DML operations over the resulting table, similarly to the GRFusion materialized graph view. The TinkerPop²⁰ framework, employing Gremlin programming language, executes every computation in the graph queries. Moreover, DB2 Graph adopts the rules for optimizing recursive queries discussed in Section 2.3.2.1, especially for reducing intermediary results by eliminating duplicates and executing the predicates early. However, DB2 Graph is limited to only one RDBMS because it relies on a polymorphic table function to retrieve the graph query results.

Other graph layers or views includes G-SPARQL (SAKR; ELNIKETY; HE, 2012; SAKR; ELNIKETY; HE, 2014), SQL Graph Database²¹, and Graphite (PARADIES; LEHNER; BORNHÖVD, 2015). All in all, these graph layers overcome the extraction approach's limitations

¹⁷ <<http://snap.stanford.edu/ringo>>

¹⁸ <<http://postgis.net/>>

¹⁹ <<https://wiki.postgresql.org/wiki/AgensGraph>>

²⁰ <<https://tinkerpop.apache.org/>>

²¹ <<https://docs.microsoft.com/en-us/sql/relational-databases/graphs/>>

by providing interoperability between the results of the graph queries with the other tables in the RDBMS. However, these layers or views are mainly tied with a specific database system, providing customized structures and algorithms that were not generalized to other DBMSs.

2.4 Final Considerations

In this chapter, we presented the main concepts related to this Ph.D. research. It started covering basic concepts from RDBMSs and graph data management in Sections 2.1 and 2.2, respectively. In Section 2.3 we combined these concepts by detailing approaches, methods, and frameworks for managing graph data in RDBMSs. Particularly, the Sections 2.3.1 and 2.3.2.1 are related with this Ph.D. thesis hypothesis and associated questions.

Section 2.3.2.1 introduced the concept of recursive queries and their optimizations. We also discussed its limitations and insights that we tackled during the development of this Ph.D. research. We expanded the definitions of recursive queries in Chapter 3, detailing our proposals and analyses related to **Question 1** of the hypothesis (see Section 1.2). Regarding the specialized storage approaches studied for **Question 2**, we detailed its state-of-the-art in Section 2.3.1, referring to its limitations, and discussed our insights over it, which led to the proposals and analysis in Chapter 4.

RECURSIVE QUERIES ANALYSIS

This chapter expands the definition of Recursive Queries and introduces our proposals to process them. Section 3.1 presents both a formal definition and implementations of Recursive Queries in RDBMSs, and discusses an application scenario employing the results of such queries combined with the data from other tables present in the RDBMS schema. Section 3.2 summarizes our proposal of sharing join operations among these recursive queries, regarding **Question 1** from this Ph.D. research: “*Is it possible to improve the performance of executing multiple graph queries, such as Single-Source Shortest Path (SSSP), Connected Components (CC), and PageRank (PR), by combining them into a single recursive query which employs an enhanced result table T containing the result of each graph query in a distinct column?*”. Finally, Section 3.3 concludes this chapter.

3.1 Recursive Query Definition

A Recursive Query is a query divided into two parts, and both are expressed in Equation 3.1. Equation 3.1a represents the first part named starting-point query, which initializes table T^0 with a subset of vertices $V' \in V$. The initialization considers any query executed over the table of vertices V as long as it projects the vertex identifiers and other columns employed in the query. We express in Equation 3.1a a query selecting the rows based on the predicates $attr_1 = x$, $attr_2 = y$, and others. The attributes $attr_1$ and $attr_2$ are columns from table V , and the letters x and y represent query-specific values for the respective column compared. Equation 3.1b represents the second part, named recursive-step query. The recursive-step query refers to the expansion step (see Section 2.3.2.1) and consists of materializing T^i by joining the results of the previous iteration T^{i-1} with the table of edges E .

$$\begin{cases} T^0 \leftarrow \pi_{id, \dots} (\sigma_{(attr_1 = x \wedge attr_2 = y \wedge \dots)} V) & (3.1a) \\ T^i \leftarrow \pi_{id_{tgt}, \dots} (T^{i-1} \bowtie_{(id = id_{src})} E) & (3.1b) \end{cases}$$

Based on Equation 3.1, we formally define a Recursive Query $Q^{\circlearrowleft}$ in Definition 1. A Recursive Query $Q^{\circlearrowleft}$ combines the results from the intermediary tables $T^0 \cup T^1 \cup \dots \cup T^{n-1} \cup T^n$. The value of n is denoted as the stopping criteria for the Recursive Query. It corresponds to whenever the values in the current recursive-step T^n is equal to the previous iteration T^{n-1} .

Definition 1 ($Q^{\circlearrowleft}$). A Recursive Query $Q^{\circlearrowleft}$ is defined as $Q^{\circlearrowleft} = \bigcup T^i$, with $0 \leq i \leq n$. The query stops after the n -th iteration when no new or different result returns from the join operation, *i.e.*, $T^{n-1} = T^n$.

3.1.1 Implementation using CTEs

Equation 3.2 presents the general CTE syntax for implementing a Recursive Query $Q^{\circlearrowleft}$. The equation starts with the WITH RECURSIVE statement, followed by the name of the materialized result table $Q^{\circlearrowleft}$. The result table data is build using the Definition 1, employing the SQL operator UNION ALL to combine the intermediary result tables. Finally, Q_f is the final query that employs the $Q^{\circlearrowleft}$ in its FROM statement to perform the desired analysis or retrieval.

$$\text{WITH RECURSIVE } Q^{\circlearrowleft} \text{ AS } (T^0 \text{ UNION ALL } \dots \text{ UNION ALL } T^n) \quad Q_f \quad (3.2)$$

To exemplify Recursive Queries' implementation using CTEs based on the SQL-99 standard (EISENBERG; MELTON, 1999), we provide the SQL implementations for both PostgreSQL and Oracle RDBMSs. Source code 2 shows it example in the PostgreSQL. Line 1 creates the table $Q^{\circlearrowleft}$ with the vertex identifier column id and its other attributes. Lines 2 and 4 are the starting-point query and the recursive-step query, respectively. Line 6 illustrates the final query Q_f , which retrieves the data from the materialized table $Q^{\circlearrowleft}$. Notice that the starting-point query in Line 2 retrieves a single vertex by its identifier value id . In the example, we used the letter x to denote any possible value of id . Furthermore, the starting-point query not necessarily retrieves only a single vertex. It can retrieve a subset of vertices V' of any size, which we explored its variations in Section 4.2.2.2.

```

1:    WITH RECURSIVE  $Q^{\circlearrowleft}(id, \dots)$  AS (
2:        SELECT  $id, \dots$  FROM V WHERE  $id = x$ 
3:        UNION ALL
4:        SELECT  $id_{tgt}, \dots$  FROM  $Q^{\circlearrowleft}$  JOIN E ON  $Q^{\circlearrowleft}.id = E.id_{src}$ 
5:    )
6:    SELECT  $id, \dots$  FROM  $Q^{\circlearrowleft}$ ;

```

Source code 2 – Recursive Query in PostgreSQL RDBMS using CTEs.

Source code 3 shows the same implementation in Oracle RDBMS. Line 1 shows that the Oracle does not requires the RECURSIVE statement when declaring the CTE. Lines 2 to 4

are equal between Source codes 2 and 3. The key differences between these two codes are in Lines 5 and 6, which specify the execution in more detail, such as the row processing order and the cycles' verification. Line 5 specifies that the recursion will return the child rows before processing the sibling rows, following a Depth First Search (DFS) specified by the `DEPTH FIRST BY` statement. Another possibility would be to use the `BREADTH FIRST BY` statement to return sibling rows before processing child rows. The `SET rank` statement generates an enumeration, which is applied to the attribute `id` to keep track of the order in which new tuples come at each recursive-step query execution. Line 6 validates the existence of cycles in the DFS, creating an attribute `cycle` in the CTE with value one (*i.e.*, true) if a cycle was detected and 0 (*i.e.*, false) otherwise. Lines 7 and 8 is the final query Q_f , ordering the rows from Q^Q by their rank value.

```

1:   WITH  $Q^Q(id, \dots)$  AS (
2:       SELECT id, ... FROM V WHERE id = x
3:       UNION ALL
4:       SELECT idtgt, ... FROM  $Q^Q$  JOIN E ON  $Q^Q.id = E.id_{src}$ 
5:   ) SEARCH DEPTH FIRST BY id SET rank
6:   CYCLE id SET cycle TO 1 DEFAULT 0
7:   SELECT id, ... FROM  $Q^Q$ 
8:   ORDER BY rank;
```

Source code 3 – Recursive Query in Oracle RDBMS using CTEs.

3.1.2 Implementation using PL/pgSQL

Despite CTE being present in the SQL since 1999, these queries do not have the optimizations discussed in Section 2.3.2.1. Particularly, because the CTE implementation relies on the `UNION ALL` statement, it only appends new results in each recursive-step query execution, not removing old and unused intermediary results. Hence, the primary way to overcome this limitation is by implementing these queries using Procedural Language for SQL (PL/pgSQL). SQL Source code 4 presents the implementation of a SSSP query using PL/pgSQL in PostgreSQL RDBMS. Notice that we opted to use temporary tables to manage the iterations and continuously reducing the number of tuples that will be joined in each recursive-step query execution.

Line 1 declares the name of the procedure `SSSP_Query`. It receives two parameters, the starting vertex v^{start} identifier, and the maximum quantity for the recursive-step query executions `MAX_REC_TIMES`. Line 2 specifies that the procedure returns a table containing the identifiers `id` from the vertex found in the SSSP query and their respective path length, or distance, to v^{start} . Line 4 declares a `RECORD` data type to serve as a flag to determine if the recursion will stop. Line 5 declares the integer `REC_TIMES` to count the number of executed recursive-step queries, comparing them to the `MAX_REC_TIMES`. Lines 7 to 9 is the starting-point query, initializing the result table `T` with distance 0 to v^{start} and ∞ , or unreachable, to the other vertices. Lines 11 to 13

```

1: CREATE FUNCTION SSSP_Query( $v^{start}$  INT, MAX_REC_TIMES INT)
2: RETURNS TABLE(id INT, distance DOUBLE PRECISION) AS $$
3: DECLARE
4:   end_condition RECORD;
5:   REC_TIMES INT := 1;
6: BEGIN
7:   CREATE TEMP TABLE T(id, distance) AS
8:     SELECT V.id::INT,  $\infty$ ::DOUBLE PRECISION FROM V;
9:   UPDATE T SET distance = 0.0 WHERE T.id =  $v^{start}$ ;
10:
11:   CREATE TEMP TABLE T' (id INT, distance DOUBLE PRECISION);
12:   CREATE TEMP TABLE Diff (id INT, distance DOUBLE PRECISION);
13:   CREATE TEMP TABLE Delta(id INT);
14:
15:   WHILE REC_TIMES  $\leq$  MAX_REC_TIMES LOOP
16:     INSERT INTO T'
17:       SELECT E.idtgt, MIN(T.distance + E.w)
18:       FROM T JOIN E ON T.id = E.idsrc
19:       GROUP BY E.idtgt;
20:
21:     SELECT INTO end_condition * FROM T' LIMIT 1;
22:     IF end_condition IS NULL THEN EXIT; END IF;
23:
24:     INSERT INTO Diff
25:       SELECT T.id, T.distance - T'.distance
26:       FROM T JOIN T' ON T.id = T'.id;
27:     SELECT INTO end_condition * FROM Diff LIMIT 1;
28:     IF end_condition IS NULL THEN EXIT; END IF;
29:     INSERT INTO Delta
30:       SELECT id FROM Diff WHERE distance  $\neq$  0 LIMIT 1;
31:     SELECT INTO end_condition * FROM Delta;
32:     IF end_condition IS NULL THEN EXIT; END IF;
33:
34:     TRUNCATE T;
35:     INSERT INTO T SELECT * FROM T';
36:     TRUNCATE T';
37:     TRUNCATE Diff;
38:     TRUNCATE Delta;
39:
40:     REC_TIMES := REC_TIMES + 1;
41:   END LOOP;
42:
43:   RETURN QUERY EXECUTE 'SELECT * FROM T';
44: END;
45: $$ LANGUAGE plpgsql;

```

Source code 4 – PL/pgSQL procedure to compute the SSSP query in PostgreSQL RDBMS.

initializes additional tables to help the execution of the recursive-step query. The loop starts at Line 15 and stops if a maximum number of iterations has been reached, or if no new shortest paths are found, validated from Lines 24 to 32 employing the tables *Diff* and *Delta*.

Lines 16 to 19 show the recursive-step query, which joins the current result table *T* with the edge table *E* to compute the new result table *T'*. Since the query computes the shortest paths, each iteration of the recursive-step query computes the minimum aggregation function over the sum of the shortest distances found in the current iteration *T.distance* plus the edge length or weight *E.w*. Lines 21 and 22 validates if new vertices are found in the current iteration. Lines 34 to 38 prepares the table for the next recursive-step query, copying the data from table *T'* to *T* and cleaning up the additional data. Line 40 increments the counter *REC_TIMES* for the next iteration. Finally, Line 43 returns the data in the resulting table *T* from the procedure *SSSP_Query*.

Concerning the PL/pgSQL procedure in Source code 4, and in every SQL query evaluated in this Ph.D. project, we always employed proper indexes over table *E*. First of all, the PK of this table is defined over the pair of attributes *id_{src}* and *id_{tgt}*, which results in creating an index over them. Moreover, we defined an index over the attribute *id_{src}* for the predicate *T.id = E.id_{src}* on Line 18 from Source code 4. Another index created over table *E* considered the triple *id_{src}*, *id_{tgt}*, and *w*, allowing the use of Index-only scans¹ to speed up the join operation. With these three indexes, we left the choice of whether to use indexes to the RDBMS query optimizer. For table *T* we employed temporary tables and, despite having an index created over its attribute *id* beforehand, our experiments reported the usage of indexes only over table *E* whenever it was possible. Such behavior was likely due to the low selectivity of the queries.

Section 3.1.3 covers an application scenario that employs an SSSP query, combining its results with data from other tables to perform more complex analysis. The interoperability in this combination is one of the reasons that motivated this Ph.D. research, as previously discussed in Section 1.1. Furthermore, other graph queries, such as PageRank and CC (see Section 2.2.2), are executed using different parameters for the starting-point and recursive-step queries. We considered their similarities to speed up their execution, as detailed in Section 3.2.

3.1.3 Application Scenario example

The purpose of performing graph queries in RDBMSs is to enable data interoperability in different applications. When combining results from both graph and traditional queries, more complex and elaborated analyses may arise. To exemplify, we summarize the G-FranC, a dataset proposed and explored in Scabora *et al.* (2019). This section details the dataset, discussing its motivation, and presenting examples of analyses encompassing both graph and traditional queries that were performed over G-FranC dataset.

Joining criminal data with the topology of a city is widely explored in the literature,

¹ <https://www.postgresql.org/docs/current/indexes-index-only-scans.html>

used to describe both transportation and collective human behaviors (KANG *et al.*, 2013), or comparing them against other urban indicators from the city (PORTA *et al.*, 2009). Notice that criminal activity is also an urban indicator. Hence, it can be employed to analyze the impact of unlawful activities, ranging from minor offenses to felonies, over collective human behavior. Several studies focus on appraising criminal activities (BETTENCOURT *et al.*, 2010; FITTERER; NELSON; NATHOO, 2014; BOGOMOLOV *et al.*, 2014; DERYOL *et al.*, 2016; NIETO-CHAUPIS, 2018a; NIETO-CHAUPIS, 2018b; FERREIRA; RUBIANO; MOJICA-NAVA, 2018; SPADON *et al.*, 2017; SPADON *et al.*, 2017), and they have the potential to aid decision-making activities regarding public policies planning for safety improvement.

Based on this motivation, G-FranC is a dataset that combines the structure of a complex network of urban streets and criminal activities from San Francisco – CA, USA, which occurred from the year 2003 to 2018. Both criminal and network data were combined in a unified and structured schema stored in an RDBMS by following the Extraction, Transformation, and Loading (ETL) data processes (KIMBALL; CASERTA, 2011). The ETL process consists of (i) Extracting the data from both domains; (ii) Transforming them by removing invalid coordinates and multiple edges between the same pair of vertices (keeping only the one with the smallest distance); and (iii) Loading the data by providing the proper SQL scripts. On the one hand, the complex network is represented as a georeferenced graph, as defined in Section 2.2.1, in which each vertex have its coordinates of latitude l_{at}^v and longitude l_{on}^v , and each edge have foreign keys for the source and target vertex identifiers (id_{src} and id_{tgt} , respectively), and the distance, in meters, of the street segment. It was extracted using the OSMnx² Python library on July 1st, 2019, resulting in $|V| = 9,559$ street intersections, and $|E| = 26,689$ unique street segments.

On the other hand, the criminal activity was provided by the *San Francisco OpenData* (SFO) initiative³, composed by reports made by police officers and by citizens. There are more than 2.2 million reports, each one with the coordinates where it occurred, from January 1st, 2003 to May 15th, 2018. Every report has the following attributes: (i) coordinates, represented by its latitude l_{at}^c and longitude l_{on}^c ; (ii) category; (iii) textual description of the activity; (iv) date of occurrence (*i.e.*, *datetime*); (v) weekday when its occurred; (vi) textual description of its resolution; and (vii) the police district that processed the report. Figure 4 summarizes the schema of G-FranC dataset, composed of three tables: the vertex table for the street intersections, the edge table for the street segments, and the crime table for the criminal reports.

The process of mapping each criminal report to the nearest street intersection uses a K-Nearest Neighbors (KNN) query, with $k = 1$, which assigns the closest vertex identifier to the attribute id_{fk}^v in the crime table. The distance function employed in the KNN query was the great-circle distance ($\mathcal{D}$), calculated between the coordinates of each crime c to each vertex $v \in V$. Equation 3.3 defines $\mathcal{D}$, which is calculated over the Earth's surface and corresponds to

² Available at <<https://github.com/gboeing/osmnx>>

³ Available at <<https://data.sfgov.org/Public-Safety/Police-Department-Incident-Reports-Historical-2003/tmnf-yvry>>.

the real distance between georeferenced elements. All latitude and longitude values employed in the equation are represented in radians because the distance function is derived from the Spherical Law of Cosines (WYLIE, 2011). In the equation, l_{at}^c and l_{at}^v are the latitude values from the crime c and the vertex v , respectively. Analogous, Δ^{lon} refers to the difference between their longitude values l_{on}^c and l_{on}^v . The Earth's radius $\mathcal{R}$ is equal to 6,371 kilometers. Finally, Table 3 shows the top 7 most frequent crime categories present in G-FranC dataset, with their respective number of occurrences and examples of criminal activities in the category.

Figure 4 – Schema proposed for G-FranC, highlighting the more relevant attributes in each table.

Source: Adapted from Scabora *et al.* (2019).

$$\mathfrak{D} = \mathcal{R} \times \cos^{-1} (\sin(l_{at}^c) \sin(l_{at}^v) + \cos(l_{at}^c) \cos(l_{at}^v) \cos(\Delta^{lon})) \quad (3.3)$$

Table 3 – The Top 7 most frequent crime categories in the G-FranC dataset, with the number of occurrences and criminal activity examples for each category.

#	Category	Occurrences	Examples
1	Larceny/Theft	480,420	Theft from private properties and shoplifting.
2	Other Offenses	309,320	Traffic violation and resisting arrest.
3	Non-criminal	238,313	Violation of civil sidewalks and fire or death reports.
4	Assault	194,685	Threats against life and physical child abuse.
5	Vehicle Theft	126,587	Reports about a stolen automobile, truck, or motorcycle.
6	Drug/Narcotic	119,621	Narcotics possession and marijuana transportation.
7	Vandalism	116,058	Malicious mischief.

Source: Scabora *et al.* (2019).

After finishing the mapping process, we have every vertex v enhanced with a list of criminal activities that occurred in it. The criminal activity list present in each vertex allows evaluating criminal communities in the city by discovering clusters in the data. The Nerstrand⁴ is a multi-thread cluster detection tool-set for discovering clusters by maximizing their modularity (LASALLE; KARYPIS, 2015). It considers the graph's topology, using its edges, and the

⁴ Available at <<https://github.com/dlasalle/nerstrand>>.

weight measurement in each vertex. Specifically, using the quantities of criminal activity per category as the vertex's weight renders multiple community analyses. Figure 5 illustrates the Top 4 largest communities identified by the Nerstrand tool-set considering different values of weights for the vertices in the graph.

Figure 5 – The Top 4 largest communities identified considering their quantity of criminal activities as weight. In (a), all weight values are set as 0, being the communities identified without considering any criminal activity, and in (b), (c), and (d), the vertex's weights corresponds to the number of crimes from the categories Larceny/Theft, Other Offenses and Non-Criminal.

Source: Research data.

Figure 5(a) shows the communities detected by employing the weight value 0 in all vertices, which corresponds to the clusters without considering any criminal activity. Figure 5(b), Figure 5(c), and Figure 5(d) shows these communities for the first three of the Top 7 categories presented in Table 3, respectively. Other community analyses can be derived from the G-FranC dataset. The quantities (*i.e.*, weight values) employed for the community analysis illustrated

in Figure 5 were calculated using the aggregate function COUNT over the foreign key id_{fk}^v and without any predicate. However, it is possible to filter the criminal activity quantity based on their other attributes showed in the crime table of the schema in Figure 4, such as by different periods from the datetime attribute or by the police district that processed the report.

For the neighborhood analysis employing Recursive Queries, we evaluated the proportion of criminal activities in the neighborhood areas of a given location (*i.e.*, vertex). This analysis starts by selecting the closest vertices from the given location by executing the SSSP query previously presented in the Source code 4. The results from the SSSP queries are then combined with the data in the crime table from Figure 4. To validate the proportion of criminal activities and avoid computing larger path lengths than needed, we limited its value to a max value max^{dist} , in kilometers. The restriction refers to adding `HAVING MIN(T.distance + E.w) ≤  $max^{dist}$` predicate after Line 19 of Source code 4 to filter paths above this value.

Figure 6 – Neighborhood analysis over the vertex with the highest number of crime occurrences (black star), showing the regions around it up to 0.5 km (in orange), 1 km (in purple), and 2 km (in green). It is possible to observe variations in the Top 7 most recurrent criminal categories in each case, as well as in their respective percentage values.

Source: Research data.

Figure 6 illustrates the neighborhood analysis performed over the vertex with the highest number of crime occurrences, represented as a black star on the map. In the bar plot located on the right of the map, we show the proportion of Top 7 categories of criminal activities reported in it, under the title of Center Query. Based on this location, we also retrieved the vertices in the range of 0.5 km (in orange), 1 km (in purple), and 2 km (in green) around it. Notice that there are variations in the Top 7 categories retrieved in each case and differences in their respective percentage values. Such observation motivates diverse analysis that combines graph data with

data from other tables in the RDBMS schema, such as exploring the similarity among crime categories studied in specific locations in a city and their neighborhood regions.

Additionally, this neighborhood analysis may result in the execution of multiple SSSP queries to explore a range of given locations with high criminal activity. Such a scenario motivated our proposal to optimize analysis based on multiple graph queries in a graph. In the next section, we detail this optimization proposal developed during this Ph.D. research. Particularly, in Section 3.2.1 we discuss more generic scenarios for graph queries over georeferenced graphs.

3.2 Sharing Join Operations in Recursive Queries

This section goal is speeding up the execution of multiple graph queries by sharing their intermediary result table T while executing the recursive-step queries (see Section 3.1). We proposed SHARq (*SHARED Recursive queries*) framework, which materializes the table T , employed on the recursive-step query, and enriching it with extra columns to foster its usability for subsequent queries. By sharing, we mean to reuse partial results throughout different tasks and analyses. We assume that graph queries such as CC, SSSP, and PR, which are frequently processed, as previously discussed in Table 2, are prone to be executed in a single database section. The SHARq proposal was initially explored in [Scabora et al. \(2021\)](#). The remainder of this section discusses its main findings.

Figure 7 – Overview of SHARq. (a) Graph queries in an RDBMS. (b) The common approach, based on recomputing common recursive joins. (c) Outline of the methodology, where SHARq reuses recursive join results among multiple queries. (d) The observed reduction of processing time.

Source: Elaborated by the author.

Figure 7 summarizes the general idea of SHARq. The framework takes into account the fact that the graph query types CC, SSSP, and PR, required in the workload from Figure 7(a), execute analogous join operations. Naively, as shown in Figure 7(b), these join operations will be computed multiple times, instead of having their output reused for multiple graph queries. Figure 7(c) illustrates the SHARq framework, which consists of combining these queries and extending the results with additional measurement columns, one for each query, enabling different computations simultaneously without re-evaluating the same join operations. SHARq aims at computing potentially useful extra measurements that depend on the same recursive join operations. Hence, the query can be executed faster than the usual execution sequence, avoiding almost redundant computations of costly analogous join operations.

3.2.1 Motivation for Sharing

Recursive Queries obtain a result table T containing the graph query results to be explored or combined with other analyses. For instance, when executing a SSSP query for the starting vertex $v_i \in V$, the current shortest path lengths are materialized in table $T_{v_i}^{SSSP} = \{t_{v_i, v_1}^{SSSP}, \dots, t_{v_i, v_{|V|}}^{SSSP}\}$. Each tuple in this table $t_{v_i, v_k}^{SSSP} = \langle v_k, w_{ik} \rangle$ keeps track of the path lengths or weights w_{ik} from vertex v_i to every vertex v_k , with $1 \leq k \leq |V|$. To motivate the SHARq usefulness, we present two application scenarios that require executing multiple graph queries: sharing among alike recursive queries and sharing among different recursive queries.

Sharing among alike recursive queries. We call alike queries those sharing the same sequence of retrieval operations but changing the comparison predicates. Suppose that someone wants to find a place to treat a set of patients far away from any existing hospital or healthcare facility in a city. Such a task would be employed for urban planning, for example, to improve healthcare facilities' accessibility. By providing the distances between hospitals and patients within a city or urban region, it is possible to group patients and explore their characteristics. Let a set of patients $\mathcal{P} = \{p_1, p_2, \dots, p_m\}$, where each patient $p_j = \langle id, age_group, \dots, id^v \rangle$, with $1 \leq j \leq m$, has an identifier id , his/her age group and other attributes, a foreign key id^v to the vertex where he/she is located. Let $\mathcal{H} = \{h_1, h_2, \dots, h_n\}$ be a set of hospitals, where each hospital $h_i = \langle id, vacancies, \dots, id^v \rangle$, with $1 \leq i \leq n$, has an identifier, some properties such as the quantity of vacancies in it, the vertex that corresponds to its location, and so on.

To filter the subset of patients far from every hospital, one must run SSSP queries starting from each hospital $h_i \in \mathcal{H}$. Using the PL/pgSQL procedure in Source code 4, the query regarding hospital h_1 initializes the (partial) result table $T_{h_1}^{SSSP}$ with the distance (measurement) to the starting vertex h_1 set to zero and the distance to the remaining vertices set to ∞ . The algorithm then performs several iterations using join operations between the result table $T_{h_1}^{SSSP}$ and the edge table E to compute the actual shortest paths (*i.e.*, the recursive-step query). The actual shortest paths are returned in the result table $T_{h_1}^{SSSP}$. The same execution is repeated for each of the remaining hospitals. The final analysis, expressed in Source code 5, combines the

distances from every hospital $h_i \in \mathcal{H}$, filtering patients whose shortest distance to a hospital is greater or equal than a threshold ε , which indicates being “far away”. The expected query output may be, for example, the number of patients that are far away per age range.

The major problem with Source code 4 is the repetitive iterations based on the joins executed for every SSSP subquery. Notice that, even starting from different hospitals $h_i \in \mathcal{H}$, they all share analogous join operations. With SHARq, we propose to group the join operations into a single one and cumulatively evaluate the shortest path distance values retrieved so far. The framework creates an enriched result table including the columns $w_1, w_2, \dots, w_n$ corresponding to the shortest distances so far from each patient to every hospital $h_1, h_2, \dots, h_n$, respectively. SHARq performs the join operation only once, simultaneously updating the shortest paths regarding all columns $w_1, w_2, \dots, w_n$. Thus, it performs a single join operation per SSSP iteration instead of a join operation per SSSP iteration per query (from each specific starting vertex). The computed shortest distances from all n hospitals is then stored in a table $T_{\mathcal{H}}^{SSSP}$. Therefore, the intended query is rewritten in Source code 6, which employs the $T_{\mathcal{H}}^{SSSP}$.

```

1: SELECT  $\mathcal{P}$ .age_range, COUNT(*) qte
2: FROM  $\mathcal{P}$  JOIN  $T_{h_1}^{SSSP}$  ON  $\mathcal{P}.id^v = T_{h_1}^{SSSP}.id$ 
3:      JOIN  $T_{h_2}^{SSSP}$  ON  $\mathcal{P}.id^v = T_{h_2}^{SSSP}.id$ 
4:      ...
5:      JOIN  $T_{h_n}^{SSSP}$  ON  $\mathcal{P}.id^v = T_{h_n}^{SSSP}.id$ 
6: WHERE  $T_{h_1}^{SSSP}.w \geq \varepsilon$  AND ... AND  $T_{h_n}^{SSSP}.w \geq \varepsilon$ 
7: GROUP BY  $\mathcal{P}.age\_range$ ;

```

Source code 5 – Query for retrieving the set of patients in a city that are “far away” from the n hospitals $h_1, h_2, \dots, h_n$.

```

1: SELECT  $\mathcal{P}.age\_range$ , COUNT(*) qte
2: FROM  $\mathcal{P}$  JOIN  $T_{\mathcal{H}}^{SSSP}$  ON  $\mathcal{P}.id^v = T_{\mathcal{H}}^{SSSP}.id$ 
3: WHERE  $T_{\mathcal{H}}^{SSSP}.w_1 \geq \varepsilon$  AND ... AND  $T_{\mathcal{H}}^{SSSP}.w_n \geq \varepsilon$ 
4: GROUP BY  $\mathcal{P}.age\_range$ ;

```

Source code 6 – Query speeded up by simultaneously computing the shortest paths to the set of hospitals $\mathcal{H}$.

Other application scenarios regarding social networks consider a set of researchers $\mathcal{R}$ and their mutual distances based on the number m of co-authored or cited works. If this graph is modeled using a bipartite projection (see Section 2.2.1), the weight measurement is defined as $1 \div m$, and we are interested in researchers that are close (or distant) to every researcher in $\mathcal{R}$. Therefore, we may rewrite query Source code 6 for this context calculating $T_{\mathcal{R}}^{SSSP}$. The same idea can also be explored in healthcare systems by modeling the relationships among concepts such as diseases and treatments as a graph. Table 1 presents other examples applicable here.

Sharing among different recursive queries. The second scenario considers grouping different graph query types, which are also based on analogous join operations. For the application domain of researcher social networks, we can consider a multicriteria analysis combining these query types. Shortest path queries retrieve close collaborators for a given subset of researchers, while the Connected Component query in this domain allows identifying sub-collaboration networks for further analysis. A PageRank query computes every vertex's importance based on the importance of its interconnections. Combining those queries enables selecting researchers from a sub-collaborative network (CC) and identify who is the nearest (*i.e.*, having a high rate of collaborations – SSSP) and most important (*i.e.*, has the highest rank values – PR) to them.

3.2.2 The Common Join Operation

Since SHARq framework defines an enriched result table, it enables updating the measurements of all graph queries in the same join operation. Therefore, it reduces the total number of joins $n_{\bowtie}$ by a factor of up to $n_{\bowtie}/n_q$, where n_q is the number of grouped queries. Regarding the multiple SSSP queries, SHARq computes the SSSP query from the entire subset of vertices $V' = \{v_1, v_2, \dots, v_n\}$ using an enriched table $T_{V'}^{SSSP}$, demanding only a single join $T_{V'}^{SSSP} \bowtie E$ per algorithm iteration regardless the number of simultaneous queries. This rationale also applies to the other graph queries CC and PR, as these queries also require analogous join operations.

Definition 2 sets the concept of *Recursive Join* employed in SHARq. This *Recursive Join* corresponds to the part of the recursive-step query, which is similar among the graph queries SSSP, CC, and PR. A recursive-step query relies on a table T with a column to store the vertex identifier (id) and at least one measurement column (w), as exemplified in Source code 4.

Definition 2 (Recursive Join). The *Recursive Join* is a join operation in a Recursive Query defined as $T' = T \bowtie E$, with join predicate $T.id = E.id_{src}$, where T and T' correspond to the results tables, respectively, at the current and the next step iteration of the Recursive Query.

The difference in these three query types is two-folded: the values initialized in T , in the starting-point query, and the aggregation function employed in each recursive-step query performed. We detail this difference in each of the three graph queries in the following:

- **SSSP query:** the measurement column w^{SSSP} is initialized with distance 0 to v_i , and ∞ (meaning unreachable) to all other vertices. Equation 3.4 presents the aggregation function for the SSSP query, previously discussed in Section 3.1.2, which accumulates the minimal distance to reach each vertex.

$$T'.w^{SSSP} = \mathcal{F}_{agg}^{SSSP} = \min(T.w^{SSSP} + E.w) \quad (3.4)$$

- **CC query:** the measurement column w^{CC} refers to the connected component's identifier to which the vertex belongs. The starting-point query assigns the same vertex identifier id to

the respective measurement column w^{CC} . Equation 3.5 presents the aggregation function for the CC query, which employ a max (or min) operation in each recursive-step query. Thus, by taking the max (or min) of w^{CC} we iteratively combine vertices from the same component and assign to them a single w^{CC} value.

$$T'.w^{CC} = \mathcal{F}_{agg}^{CC} = \begin{cases} \max(T.w^{CC}) \\ \min(T.w^{CC}) \end{cases} \quad (3.5)$$

- **PR query:** the measurement column w^{PR} refers to the respective vertex's rank in the PageRank. The initialization assigns the value $\frac{1}{|V|}$ to every vertex. To facilitate the calculus, we use an additional column named *deg*, which stores each vertex outdegree. Equation 3.5 presents the aggregation function for the PR query. It employs a damping factor $d \in \mathbb{R}$, $d \in [0, 1]$. At each recursive-step query executed, the rank of a vertex v_i is updated based on the rank of the vertices v_j such that there is a directed edge (v_i, v_j) .

$$T'.w^{PR} = \mathcal{F}_{agg}^{PR} = \left(d * \text{sum} \left(\frac{T.w^{PR}}{T.deg} \right) \right) + \left(\frac{1 - d}{|V|} \right) \quad (3.6)$$

Despite using different aggregation functions, every query employs the same GROUP BY operation in each recursive-step query execution. This grouping operation, as previously exemplified Line 19 from Source code 4, asserts the optimization rules for the Recursive Queries (see Section 2.3.2.1) to eliminate duplicated or unused rows in each iteration. All these three graph queries (*i.e.*, CC, SSSP, and PR) group the data based on the target vertex identifier id_{tgt} in table E . Hence, SHARq enables performing the same grouping operation, but employing a different aggregation function for each graph query executed. By adding extra columns to table T and adapting the initialization and step functions accordingly, simultaneous queries share the same *Recursive Join*, resulting in a reduction in the total number of join operations executed.

3.2.2.1 Adapted Algorithm for SHARq

Algorithm 1 describes the *Recursive Join* execution of multiple graph queries proposed by SHARq framework. The algorithm receives both tables V and E , and a set of graph queries $\mathcal{Q} = \{ Q_1, Q_2, \dots, Q_n \}$. Each query Q_i , with $1 \leq i \leq n$, can be either a SSSP, a CC, or a PR query. We omitted the parameters specific for each graph query for simplicity, such as the damping factor d in the PR query and the starting vertex v^{start} in a SSSP query. The algorithm outputs the result table T containing measurements w_i for every Q_i , and it is divided into the initialization and recursive parts, explained as follows.

Line 1 initializes table T by projecting the column of vertex identifiers id of V table. Lines 2 to 5 initialize the measurement columns w_i for each graph query $Q_i \in \mathcal{Q}$, accordingly. The

algorithm starts its loop by performing *Recursive Join* operations, converging every measurement to their final values iteratively. Line 6 computes table T' according to the *Recursive Join* as expressed by Definition 2 while iterating over the aggregation functions $\mathcal{F}_{agg}^1, \mathcal{F}_{agg}^2, \dots, \mathcal{F}_{agg}^n$, according to the respective graph queries $Q_1, Q_2, \dots, Q_n$. Each $\mathcal{F}_{agg}^i$, with $1 \leq i \leq n$, can be any aggregation function defined in Equations 3.4, 3.5, or 3.6. Line 7 filters the vertices with the differences between the measurement values in the current T and the new T' . Both SSSP and CC queries have $\xi = 0$, evaluating whenever there is a change in the shortest path length values or in the connected components. For the PR query, we employ a small value for ξ (e.g., 10^{-11} used in the experiments) to check whether the rank of every vertex has presented minor variation, which is the convergence criterion. Line 8 updates table T with the values in table T' . Line 9 is the loop condition, which continues the recursion whenever there are vertices retrieved in the selection operation σ of Line 7. Finally, Line 10 returns table T with all measurements for $\mathcal{Q}$.

Algorithm 1 – Algorithm employed in SHARq framework for multiple graph queries.

```

1: Input:  $V, E, \mathcal{Q} = \{ Q_1, Q_2, \dots, Q_n \}$ 
2: Output: Table  $T$  with all measurements  $w_i$  corresponding to  $Q_i$ 
3: /* Initialization: */
4:  $T \leftarrow \Pi_{(id)} ( V )$ 
5: for all  $Q_i \in \mathcal{Q}$  do
6:   Initialize  $w_i$  column and add it to  $T$ 
7: end for
8: /* Recursive Join operation: */
9: repeat
10:    $T' \leftarrow E.id_{tgt} \mathcal{V}(\mathcal{F}_{agg}^1, \mathcal{F}_{agg}^2, \dots, \mathcal{F}_{agg}^n) \left( \begin{array}{c} T \\ T.id = E.id_{src} \end{array} E \right)$ 
11:    $\Delta T \leftarrow \sigma_{(|T'.w_i - T.w_i| \leq \xi)} \left( \begin{array}{c} T' \\ T'.id = T.id \end{array} T \right)$ 
12:    $T \leftarrow T'$ 
13: until  $\Delta T = \emptyset$ 
14: return  $T$ 
```

3.2.3 Experimental Evaluation using SHARq

Table 4 presents the 18 datasets used for evaluating SHARq, which are divided into three different categories. The first category refers to Healthcare systems, and represents the relationships between clinical findings and diagnoses (*i.e.*, concepts). It was acquired using the Common Data Model of the Observational Medical Outcomes Partnership (OMOP CDM) (LIMA *et al.*, 2019), having patient's clinical data⁵. To generate datasets with different cardinalities, we extracted six cohorts by filtering their concepts and relationships. The keywords employed to filter the concepts are “cardiac”, “alcohol”, “vaccin” (to include both words vaccination and

⁵ <<https://www.ohdsi.org/wp-content/uploads/2017/10/OHDSI-Symposium-2017-VM-Download.pdf>>

vaccine), “fracture”, “syringe”, and “inject”. The edge’s weight attribute w in table E was set as $1/n_r$, where n_r is the number of relationships between every concept’s pair.

Table 4 – The 18 datasets employed to evaluate the SHARq framework, divided into three categories: Healthcare (OMOP CDM), Social Network (SN), and Urban Streets (US).

category	keyword	dataset name	$ V $	$ E $
Healthcare (OMOP CDM)	cardiac	OMOP-cardiac	2,783	4,139
	alcohol	OMOP-alcohol	10,921	17,499
	vaccin	OMOP-vaccin	15,417	32,733
	fracture	OMOP-fracture	26,307	113,904
	syringe	OMOP-syringe	64,726	525,816
	inject	OMOP-inject	362,727	2,460,343
Social Network (SN)	SQL	SN-sql	3,334	11,046
	relational	SN-relational	6,572	19,438
	topology	SN-topology	10,151	28,669
	query	SN-query	19,329	69,077
	database	SN-database	35,795	115,685
	network	SN-network	188,622	810,456
Urban Street (US)	10 kilometers	US-10km	2,970	8,342
	San Francisco (SF)	US-SF	9,559	31,414
	30 kilometers	US-30km	11,435	31,362
	Seattle	US-Seattle	15,668	64,232
	50 kilometers	US-50km	21,607	57,262
	90 kilometers	US-90km	65,670	171,012

Source: Elaborated by the author.

The second category refers to Social Network (SN) data and considers an academic network describing collaborations among researchers provided by the ArnetMiner system (TANG *et al.*, 2008). It has a total of 1,712,433 researchers and 4,258,615 collaborations through co-authorship (*i.e.*, edges) between them⁶. Each researcher has an identifier, followed by its name, list of affiliations, number of publications and citations, H-index, P-index, and a list of research topics of interest. The edge’s weight attribute w in table E is set as the distance between each researcher’s pair given by $1 \div m$, where m is the number of publications co-authored between them. Higher values of m result in shorter distances between two researchers. To variate the cardinality in this category, we filtered a subset of researchers by choosing different research topics. For example, regarding the topic “sql”, we filtered researchers interested in subjects related to SQL queries, NoSQL databases, and so on. The selected topics were “sql”, “relational”, “topology”, “query”, “database” and “network”.

The third category refers to Urban Streets (US), and they are Georeferenced Graphs (see Section 2.2.1) acquired using the OSMnx Python library. We randomly selected two cities for the experiments: San Francisco (SF) and Seattle. The other four datasets from the US category

⁶ <<https://www.aminer.org/aminernetwork>>, last accessed on 13 July 2020

targeted a scalability evaluation. We set a random coordinate (*i.e.*, latitude and longitude) and retrieve the induced subgraphs around it up to a maximum distance. The maximum distances employed were 10, 30, 50, and 90 kilometers. The edge's weight attribute w in table E were the distance between each pair of vertices, in meters.

All the experiments in this section were carried out using the RDBMS PostgreSQL version 10.12, in a machine equipped with an Intel Core i7-8700 @ 3.20GHz processor, 16GB of DDR4 2,400 MT/s RAM, SATA 6.0 Gb/s 7,200RPM hard disk, and Linux operating system Ubuntu 18.04. Every graph query's correctness was verified by comparing the results with those obtained by the NetworkX (see Section 2.3.2.2). For the SSSP and CC query types, every SHARq's result matches the corresponding NetworkX's one. For the PR query type, every rank value computed by SHARq is close to the NetworkX library results. Every query was executed 40 times, and the average time was computed by discarding 30% of the values – 15% from the best, and 15% from the worst values. For SSSP queries, the random starting vertex was chosen. For PR queries, we employed the damping factor $d = 0.85$, and $\xi = 10^{-11}$ to approximate every rank's result.

3.2.3.1 Application Scenarios for SHARq

This section presents analyses for the three dataset categories, illustrating cases that benefit from SHARq framework. These cases either require simultaneous alike queries or simultaneous varying query types (*i.e.*, CC, SSSP, and PR queries), as motivated in Section 3.2.1.

Scenario 1: Social Network category. To evaluate the case of sharing among alike queries, consider a query to identify and explore potential new researchers related to a specific subject, as follows. $\mathcal{Q}_A$: *In a collaboration network where researchers are ranked by their h-index metrics, consider the top-20 researchers interested in “sql”. Using each one of them as the starting point in an SSSP query, retrieve the farthest ones, with distances greater than ϵ , categorizing them according to their h-index and number of citations.*

Regarding the data shared among different query types, consider the following example: $\mathcal{Q}_B$: *Given a researcher, search for its closest as well as its most influential researchers, which corresponds to SSSP and PR queries, respectively. If some researchers have never collaborated with the given researcher, identify their sub-collaboration network with a CC query.*

Scenario 2: Healthcare Category. An example of sharing alike queries in this category is the following analysis. $\mathcal{Q}_C$: *Given a list of 10 clinical findings related to cardiac problems, retrieve the closest treatments and diagnoses through SSSP queries, where the closest ones refer to concepts occurring at a larger number of relationships. Based on the closest treatments and diagnoses, with distances lower or equal to ϵ , investigate the patients related to them.*

Regarding sharing among different queries, we executed the following example. $\mathcal{Q}_D$: *Given a cardiac clinical finding, search for the closest and the most important diagnoses and*

treatments through SSSP and PR queries, respectively. If such a clinical outcome has never been related to other cardiac issues, identify the diagnoses and treatments for these other issues through a CC query. Investigate the patients related to the most influential treatments and diagnoses retrieved, including other cardiac sub-network(s), if any.

Scenario 3: Urban Street Category We have already presented examples from this category that demands multiple alike queries (in Section 3.2.1, with the Source codes 5 and 6, and in Section 3.1.3 with the neighborhood analyses over criminal activities). We paraphrase it here as follows. $\mathcal{Q}_E$: Retrieve the patients far away from the top-10 healthcare facilities having higher quantities of vacancies in a city, with every shortest distance being greater or equal to ϵ . Analyze the patients in terms of age range distribution.

With respect to sharing among different queries, consider the following analysis. $\mathcal{Q}_F$: Given a hospital, retrieve the shortest distances between it and the other hospitals or healthcare facilities in the city, aiming at identifying facilities that are considered to be close to each other if the distance between them is lower than or equal to ϵ . Then, analyze the importance of each facility's location by means of a PR query. Finally, also identify potentially inaccessible points in the city due to obstructed street segments through a CC query.

3.2.3.2 Performance Evaluation

This section presents results comparing the performance of SHARq and WITH+ competitor (see Section 2.3.2.1). WITH+ is considered state-of-the-art for recursive queries in RDBMSs.

Performance of Sharing Among Alike Queries. Figure 8 compares the execution time results for $\mathcal{Q}_A$, $\mathcal{Q}_C$, and $\mathcal{Q}_E$, for the Social Network, Healthcare and Urban Streets categories, respectively, when starting SSSP queries from 5, 10, 15 and 20 vertices simultaneously (*e.g.*, researchers, clinical findings, or healthcare facility's locations). For the Healthcare category in the first column of Figure 8, the reductions provided by SHARq varied from 33% to 71%. Regarding the SN category shown in the second column of Figure 8, our proposal's time reductions varied from 34% to 62% for all six datasets. In the Urban Street category, presented in the third column of Figure 8, the query processing time reductions varied from 84% to 89%. These reductions illustrate the improvement provided by sharing *Recursive Join* operations across alike graph queries with SHARq framework. Moreover, it is important to notice that the highest reductions occurred for the Urban Street category, which further motivates our example in Section 3.1.3.

Performance of Sharing Among Different Queries. Figure 9 compares the execution time results of $\mathcal{Q}_B$, $\mathcal{Q}_D$, and $\mathcal{Q}_F$, for the Social Network, Healthcare and Urban Streets category, respectively, regarding different graph queries. To obtain the same query results, it is required either to execute all the corresponding graph queries, paying the sum of their required time, or just using SHARq. The reductions provided by SHARq framework for the Social Network category, in the first column of Figure 9, the reductions ranged from 44% to 49% when combining SSSP with CC, 4% to 23% for PR + SSSP, 7% to 22% for PR + CC, and 8% to 37% for all

Figure 8 – The execution time reductions for 5, 10, 15, and 20 combined SSSP queries observed with SHARq framework in 18 datasets, organized by their category in different columns.

Figure 9 – The execution time reductions for the combinations of SSSP, CC, and PR queries observed with SHARq framework in 18 datasets, organized by their category in different columns.

three graph queries. Regarding the Social Network category, in the second column of Figure 9 ranged from 42% to 53% when combining SSSP + CC, 12% to 26% for PR + SSSP, 9% to 22% for PR + CC, and 10% to 31% for all three queries. Finally, for the Urban Street category in the third column of Figure 9, the reductions ranged from 20% to 44% when combining PR with CC, 28% to 40% for SSSP + CC, 2% to 42% for SSSP + PR, and 31% to 54% for all queries.

All in all, from Figures 8 and 9 it is possible to observe the potential of sharing intermediary results from Recursive Queries. The query processing time reduction reached almost 90% for multiple SSSP queries when combining up to 20 of them. For the combinations of SSSP, CC, and PR queries, the reductions reached 54%. However, in some cases, such combinations were not so promising because they achieved less than 10%. These less promising results were observed mainly in combinations where one query requires much more join operations than the other one, which motivates one of the future works discussed in Section 5.2 regarding removing the columns from T whenever their respective query finishes. Even presenting small improvements in some combinations, SHARq always outperformed its competitor *WITH+*.

3.2.3.3 Trade-off when Including Extra Columns

This section explores the trade-off between the number of disk accesses and the total amount of *Recursive Join* operations required to obtain the additional columns. An enriched table T proposed in SHARq framework charges a cost to manage the extra measurement columns. Additional columns lead to a larger temporary result table T, increasing the number of disk accesses required to manage it. Hence, there is a trade-off to reduce the number of *Recursive Join* operations paid by increasing each operation's individual cost.

Figure 10 shows that the reduction in the number of join operations is significantly larger than the small individual increase in the number of disk accesses, so the overall query processing time is always reduced (as also shown in Figures 8 and 9). Notice that points near to the top of the plot represent the best results in terms of less *Recursive Join* operations, while values close to the right-hand side represent the highest costs in each operation. Each plot in Figure 10 shows a combination of different graph queries, and each marker depicts one of the three categories.

Regarding the combination of the SSSP and CC queries, the Healthcare category datasets provided both the smaller number join operations and the smaller overhead compared to the other categories. However, in the other query combinations, the Healthcare category provided the smallest join operation reductions. The combinations SSSP + PR, CC + PR, and ALL benefited more in the Urban Streets category regarding a smaller number of join operations, despite having large access overheads. Furthermore, in the combinations that include the PR query, the Social Network category provided the worst results by incurring both in a larger number of *Recursive Join* operations and in a larger access overhead. Nonetheless, we can see similar overall behavior among the three categories for SSSP + PR, CC + PR, and ALL. Summarizing, it is possible to conclude that some query combinations are more favorable than others to certain categories

when sharing join operations, but SHARq can always speed up the query processing.

Figure 10 – Comparison between the overhead introduced by additional measurement columns in each *Recursive Join* operation and the total number of operations in SHARq framework.

Source: Research data.

3.3 Final Considerations

This chapter expanded the Recursive Queries' context, presenting its definition, implementation, and application scenarios examples. We also cover our proposal named SHARq, which enriches the temporary result table *T* employed in the Recursive Queries. By adding extra measurement columns to the table *T*, we enable sharing diverse computations without re-evaluating the same join operations. Employing a total of 18 datasets, divided into three categories, we showed query processing time reduction of up to 89% when executing 20 SSSP queries, and of up to 54% when combining the graph queries SSSP, CC, and PR. All in all, we addressed the **Question 1** by proposing a framework that combines multiple graph queries into a single Recursive Query employing an enhanced result table *T*.

ADJACENCY-LIST REPRESENTATIONS IN AN RDBMS

This chapter presents specialized storage approaches for the table of edges E proposed during this Ph.D. research. It helps us answer the **Question 2**: “*Is it possible to increase the recursive join operation performance when processing graph queries by enhancing the edge table E representation using adjacency-based storage approaches?*”. Section 4.1 details the Edge- k method, which groups the tuples in the edge table. Section 4.2 describes *Edge Clustered Table*, another method for grouping edges, but managing them on pages in the disk, and thus organizing them into the same disk pages of blocks. Finally, Section 4.3 concludes this chapter.

4.1 Tuple-Level Grouping: Edge- k

Edge- k is a specialized storage approach that groups multiple edges into a single table’s row or tuple. It consists of employing additional columns in the edge table E , which enables grouping each vertex neighborhood into a reduced number of tuples. The resulting table from the Edge- k method is named table Edge- k , so, in the remaining of this section, we use Edge- k to refer to the proposed method, whereas table Edge- k denotes the resulting table based on the Edge- k method. With this method, we overcome the problem discussed in existing graph representations in RDBMSs (see Section 2.3), about storing each edge in a separated tuple. Table Edge- k was initially explored in Scabora *et al.* (2017) and extended in Scabora *et al.* (2018). The remainder of this section discusses the main findings concerning both publications.

4.1.1 The Edge- k Method and Table

The Edge- k method implements each row e_k following the Definition 3.

Definition 3 (e_k). Let the neighborhood of an vertex id_{src} be $(id_{tgt}^1, id_{tgt}^2, \dots, id_{tgt}^k)$, where $id_{src}, id_{tgt}^i \in V$, $1 \leq i \leq k$, and the weight of the edge from id_{src} to every vertex id_{tgt}^i be $(w^1, w^2, \dots, w^k)$, respectively. The Edge- k table's row e_k is defined as $e_k = \{id_{src}, \langle id_{tgt}^1, w^1 \rangle, \dots, \langle id_{tgt}^k, w^k \rangle\}$

In table Edge- k , a tuple e_k can store up to k edges starting from the same id_{src} vertex. For each edge's property, such as the distance or weight between every pair of vertices, an additional column is included for each distinct target vertex id_{tgt} . To keep the example simple, we only refer to one edge's property: its weight w . Hence, for each target vertex id_{tgt}^i , we have its respective weight w^i , with $1 \leq i \leq k$. Figure 11 illustrates the table Edge- k over an example of graph.

Figure 11 – Graph example and its representation: (a) a conventional edge table, with one edge per row; and (b) the table Edge- k , with each edge e_k storing up to k edges in a single row.

Source: Adapted from Scabora *et al.* (2017).

By employing the table Edge- k to group and reduce the number of rows in the edge table, we aim at: (i) minimize the quantity of I/O operations when processing SSSP queries; (ii) decrease the number of data blocks used to store the graph due to less space required for the row headers; and (iii) ensure that at least part of the vertex's neighborhood will be contiguously stored in disk, which can also help achieve higher query execution performance. The main idea of table Edge- k is to store each vertex's adjacencies using an adjacency-list representation that spans to multiple rows, with a predefined number k of edges per row.

It is important to notice that most of the vertex's neighborhood will have a size $degree_v$ different than the value of k . Because of these differences, it is necessary to define rules to manipulate edges e_k in the Edge- k table, discussed as follows:

- **[rule(A)] $degree_v = k$:** When $degree_v$ and k are equal, where a single completely-occupied row e_k contains the whole vertex neighborhood (*i.e.*, no requiring any NULL value);
- **[rule(B)] $degree_v < k$:** When $degree_v$ is smaller than k , the vertex's neighborhood is stored in a single row e_k using NULL values to complete it; and

- **[rule(C)] $\text{degree}_v > k$:** When degree_v is larger than k , the first k edges are stored in a single completely-occupied row e_k . Then, the remaining neighborhood (*i.e.*, $\text{degree}_v - k$) will be inserted in an additional row. These rules reapply over the remaining neighborhood while its size is larger than k . The recursion finishes when the remaining neighborhood is equal (*i.e.*, **rule(A)**) or smaller (*i.e.*, **rule(B)**) than k .

Figure 11 illustrates both conventional and Edge- k tables to store an example directed graph, as well as it covers examples of the rules rule(A), rule(B), and rule(C). Figure 11(a) shows the conventional edge table storing only one destination vertex id_{tgt} per row, with its respective property w . Figure 11(b) presents the table Edge- k , storing up to k destination vertices $id_{tgt}^1, id_{tgt}^2, \dots, id_{tgt}^k$ per row, with their respective property $w^1, w^2, \dots, w^k$. Regarding the rules, let the neighborhood of vertex v_0 have a size degree_v greater than k . By following the rule(C), the adjacency list of v_0 needs to be split into two or more rows. In the example, we consider that $\text{degree}_v = 2 * k$, so precisely two rows can store the complete adjacency list of v_0 , where each row follows the rule(A). The v_2 , on the other hand, has its neighborhood size degree_v smaller than k , thus employing the rule(B) and requiring NULL values to complete the edge e_k .

4.1.1.1 Main Parameters for estimating the value of k in the table Edge- k

Table Edge- k provides the reduction in the number of rows at the cost of allowing NULL values, mainly generated by the rule(B). Tuning the value of k is required to avoid tables with either too many rows (*i.e.*, low k value) or too many columns (*i.e.*, high k value). The focus of the table Edge- k is to reduce the overall number of rows required to store the edges while keeping the amount of NULL values in the edge table as small as possible. Minimizing both the number of rows and quantity of NULL values reduces the size required for storing the table and, consequently, improves query execution performance employing the table Edge- k . Aiming at tuning the value of k when using the table Edge- k , we provide equations to estimate the number of rows η_{rows}^k and the quantity of NULL values η_{NULL}^k required to fill the incomplete e_k rows.

The η_{rows}^k takes into account, for each vertex $v \in V$, the amount of rows required to store its complete neighborhood size degree_v . The quantity of rows per vertex v is determined by the smallest integer value that is greater than degree_v divided by the defined value of k . Overall, by summing the number of rows required by each vertex in V , it is possible to estimate η_{rows}^k , as shown in Equation 4.1. While η_{rows}^k takes into account the number of additional rows required whenever degree_v is larger than k , the η_{NULL}^k computes whenever degree_v is smaller than k . Specialty, the η_{NULL}^k depends on the number of NULL values required to fill incomplete e_k rows. Since table Edge- k groups each vertex's neighborhood into chunks of size k , the number of edges occupying in the last row e_k is equal to the rest of the integer division of degree_v by k . Conversely, the unoccupied part of this row e_k is filled with NULL values and corresponds to k minus the occupied part with edges. Whenever $\text{degree}_v > k$, according to rule(C), the row e_k will not have NULL values, but an additional computation is required to segment the degree_v

into chunks of k size. The fragmentation consists of computing the rest of the division of the occupied part by k . Finally, η_{NULL}^k is obtained by summing the unoccupied neighborhood of each vertex $v \in V$, as shown in Equation 4.2.

$$\eta_{\text{rows}}^k = \sum_{v \in V} \left\lceil \frac{\text{degree}_v}{k} \right\rceil \quad (4.1)$$

$$\eta_{\text{NULL}}^k = \sum_{v \in V} (k - (\text{degree}_v \bmod k)) \bmod k \quad (4.2)$$

The optimal storage approach would use a single row to store the complete neighborhood degree_v of each vertex $v \in V$. This case results in a minimal space used for the row overhead imposed by each e_k row header (*i.e.*, internal control data most of RDBMSs) and wasted while filling NULL values. Due to the differences between degree_v and k previously discussed, the table Edge- k would be adequately represented only for the particular case where every vertex has a fixed neighborhood size. Such a premise is invalid for most complex networks. Since the table Edge- k strictly follows the Relational Model, it is possible to compute the exceeding space $\eta_{\text{exceeding}}^k$, in bytes, required both to deal with overheads of every additional e_k row and to represent the NULL values. Equation 4.3 shows how to compute $\eta_{\text{exceeding}}^k$.

$$\eta_{\text{exceeding}}^k = (\eta_{\text{NULL}}^k * \text{size}(\text{NULL})) + (\eta_{\text{rows}}^k - |V|) * (\text{row}^{\text{overhead}} + \text{size}(id_{\text{src}})) \quad (4.3)$$

where $\text{size}()$ is a function to determine the size, in bytes, of the attribute id_{src} and to store a NULL value, and $\text{row}^{\text{overhead}}$ is the row overhead of an RDBMS table. This equation is divided into two parts. The first part computes the space occupied by NULL values given by the product of η_{NULL}^k by the space occupied by each NULL value, in bytes, acquired by the function $\text{size}(\text{NULL})$. The second part considers the wasted space occupied by the additional e_k rows required whenever the neighborhood size is larger than the chosen k value. In other words, $\eta_{\text{rows}}^k - |V|$ corresponds to the number of extra rows in the table Edge- k , which is multiplied by the cost of each additional row. This cost is defined by the row overhead $\text{row}^{\text{overhead}}$ plus the size, in bytes, of replicated vertex id_{src} . Finally, Equation 4.3 sums these two parts to determine the total $\eta_{\text{exceeding}}^k$ space.

Additionally, the result of the function $\text{size}(\text{NULL})$ depends on the RDBMS implementation and representation. If an RDBMS does not compact the NULL values, it will reserve the predefined size, in bytes, of the corresponding attribute. In such a case, $\text{size}(\text{NULL}) = \text{size}(id_{\text{tgt}}^i)$, $\forall i$ in $1 \leq i \leq k$. Moreover, most of the RDBMSs usually apply techniques to save space when storing NULL values, and the real size of $\text{size}(\text{NULL})$ varies according to the product. It is also important to notice that Equation 4.2 only considers the vertex's identifiers when computing the η_{NULL}^k . If additional edge properties are present (*e.g.*, weight or distance), the value of η_{NULL}^k must be multiplied by the number of properties employed. Although η_{rows}^k , η_{NULL}^k , and $\eta_{\text{exceeding}}^k$ help

users to tune the storage of graphs for every scenario, the table Edge- k depends on choosing a value for k to allow shorter graph query processing times.

4.1.2 Graph Management Adaptations with Edge- k

This section covers the two adaptations required to manage the table Edge- k and process SSSP queries over it. Section 4.1.2.1 presents the modifications required to process the operations of insert, update, and delete, as well the bulk loading operation to completely load the table Edge- k . Section 4.1.2.2 depicts the adaptation in the SSSP query while employing table Edge- k .

4.1.2.1 Table Management and Bulk Loading

Regarding the insert, update, and delete operations, it is important to notice that table Edge- k introduces a minor overhead compared to the conventional edge table. Executing those operations over the table Edge- k is more expensive because they demand not only locating a row e_k , but also finding its respective column id_{tgt}^i in that row, $1 \leq i \leq k$. Each operation is implemented as a distinct PL/SQL procedure, detailed as follows.

Insert and Update operations. The insert operation requires searching for an unoccupied column (*i.e.*, $id_{tgt}^i = \text{NULL}$) in a row e_k of the table Edge- k that already is storing edges from the same id_{src} vertex. If an unoccupied spot is available, the new edge will be written in the corresponding column along with its properties. Otherwise, a new e_k row is created, inserting the new edge in it and filling up the remaining spots with NULL values. The update operation has a similar approach; however, instead of finding an unoccupied column, it searches for a column pointing to a specific id_{tgt} vertex and updating its property values. We assume that the update operation refers solely to changing some property (*e.g.*, its weight or distance) of a given edge. Updating either the id_{src} or the id_{tgt} of an edge is equivalent to its deletion and recreation.

Delete operation. This is a more complicated operation because it focuses on maximizing the number of complete rows (*i.e.*, without any NULL values). Consider the deletion of an edge $e = \langle v_1, v_2 \rangle$, that is, between the vertices $id_{src} = v_1$ and $id_{tgt} = v_2$. This edge is deleted by assigning a NULL value to the column id_{tgt}^i corresponding to v_2 in a row e_k where $id_{src} = v_1$. There are two cases for the delete operation: (i) either the row e_k is incomplete, or all rows where $id_{src} = v_1$ are complete. In this case, it just swaps the value in the column corresponding to v_2 with the value from the last occupied column in the same row; and (ii) there is an incomplete row e_k' . In this case, the swap operation occurs between e_k and e_k' , replacing the last occupied column in row e_k' with the value from the column corresponding to v_2 in row e_k . Afterward, the columns corresponding to v_2 and their respective properties are filled with NULL values. If the operation removes the last id_{tgt}^i vertex in the row e_k , then this row is removed from table Edge- k .

Bulk loading process. The workaround to overcome the overhead introduced by Edge- k is executing a bulk loading operation. The operation receives as input a comma-separated (CSV)

file, containing each edge in a distinct row as usually used in a conventional edge table (*i.e.*, the list of edges representation). Then, the technique is executed in three steps:

- **Sort:** Performs a disk-based sort operation over the lines of the CSV file based on the id_{src} column (*i.e.*, the source vertex identifier of each edge). Such sorting is required to optimize the second step by allowing a sequential read of the CSV file;
- **Cast:** Converts the sorted CSV file, containing one edge per line, into another CSV file containing up to k edges per line. This step consists of sequentially reading the CSV file lines and combine them in the same row as long as the id_{src} vertex remains the same and the current number of edges in the row is less than k value; and
- **Load:** Execute the loading operation in the RDBMS using the converted CSV file. Examples of this operation includes the COPY¹ command in PostgreSQL and the SQL*Loader² in Oracle RDBMS.

4.1.2.2 Graph Navigation Query Adaptation

As previously mentioned, the SSSP query is a search-find greedy-based algorithm. The query starts at a user-given vertex id_{src} , from which it will iteratively expand the vertex's neighborhood (or frontier) until it reaches a given stop criterion (for more details, see Section 3.1.2). Executing the SSSP query using the table Edge- k requires an adaptation in the Source code 4. In short, since the table Edge- k groups multiple edges in the same row during the *Recursive Join* operation (see Definition 1), it is necessary to ungroup the rows in the new temporary result table T' before starting a new recursive iteration.

```

1: INSERT INTO T'
2:   SELECT idsrc, MIN(w) w
3:   FROM (
4:     SELECT UNNEST(array[idtgt1, idtgt2, ..., idtgtk]) idsrc,
5:           T.w + UNNEST(array[w1, w2, ..., wk]) w
6:     FROM T JOIN E ON T.id = E.idsrc
7:   ) TUNNEST
8:   WHERE idsrc IS NOT NULL
9:   GROUP BY idsrc

```

Source code 7 – Adaptation of the recursive-step query when employing table Edge- k .

The adaptation consists of modifying the Lines 16 to 19 in the Source code 4. The experiments using the SSSP query described in this section were executed using the PostgreSQL RDBMS, so the ungroup operation was implemented using the UNNEST³ function. Source code 7

¹ <<https://www.postgresql.org/docs/current/sql-copy.html>>

² <<https://docs.oracle.com/en/database/oracle/oracle-database/19/sutil/oracle-sql-loader.html>>

³ <<https://www.postgresql.org/docs/current/functions-array.html>>

present this modification. The focus is in creating a temporary table T^{UNNEST} , which consists in performing the UNNEST function over the target vertices id_{tgt}^i and their respective weights w^i , $1 \leq i \leq k$. Additionally, due to the presence of NULL values, it is also required to filter these target vertices by using the WHERE clause in Line 8 from the Source code 7. The remaining of the query follows the original Source code 4, which employs the MIN aggregation function over the table T^{UNNEST} to create the temporary result table of the next recursive step T' . Such adaptation can be employed in other RDBMSs using analogous operators. For example, using the UNPIVOT⁴ operator, from Oracle RDBMS, achieves similar results.

4.1.3 Experimental Evaluation using the table Edge- k

This section presents experiments employing the table Edge- k divided into three parts. Section 4.1.3.1 shows the comparison of elapsed times when performing the insert, update, delete and bulk load operations presented in Section 4.1.2.1. Section 4.1.3.2 aims at analyzing both the dataset size reduction, achieved by minimizing η_{rows}^k , and the time reductions while executing the adapted SSSP query (see Source code 7). Section 4.1.3.3 compares the table Edge- k , which is a specialized tuple-level storage approach, with the Graph-oriented NoSQL System named Neo4j (see Section 2.2.3).

We evaluated the table Edge- k on both real and synthetic datasets. The real dataset is from a research network (see Table 1) extracted from the *Digital Bibliography & Library Project*⁵ (DBLP). We collected the data from the DBLP dataset on March 16th, 2017, and it has more than 1.9 million vertices (researchers) and more than 19 million edges (*i.e.*, co-authorship in the same publication). We illustrate this graph in Figure 12, where the edge's weight w denotes the distance between a pair of researchers. The weight is determined by $w = 1 \div n^{\text{publications}}$, with $0 \leq w \leq 1$, where $n^{\text{publications}}$ is the number of shared publications between the pair of researchers. Weight values close to 0 represent high proximity (*i.e.*, higher number of shared publications and stronger relationships, as illustrated using stronger lines). In contrast, weights close to 1 represent weaker relationships (*i.e.*, lower proximity, as illustrated using thinner lines).

The synthetic datasets were generated using the Newman-Watts-Strogatz and Erdős-Rény models previously described in Section 2.2.1, both generated using the NetworkX Python library. For the Newman-Watts-Strogatz model, we fixed the number of shortcuts X^{Newman} and varied the number of vertices $|V|$. For the Erdős-Rény model, we fixed the number of vertices $|V|$ and varied the probability p^{Erdos} of creating edges between every pair of vertices. Table 5 details all datasets' characteristics and includes the parameters explored in the synthetic datasets. The characteristics are the number of vertices $|V|$ and edges $|E|$, and the minimum, maximum, and average neighborhood's sizes (*i.e.*, $degree_v$) present in each dataset.

⁴ <https://oracle-base.com/articles/11g/pivot-and-unpivot-operators-11gr1>

⁵ Available at <http://dblp.uni-trier.de/xml/>

Figure 12 – The DBLP dataset, representing the direct and indirect relationships between the researchers through their shared publications.

Source: Adapted from Scabora *et al.* (2018).

Table 5 – Characteristics of both real and synthetic datasets explored with table Edge- k .

characteristic	DBLP	Newman-Watts-Strogatz $X^{Newman} = 200$			Erdős-Rényi		
					$p^{Erdos} = 1\%$	$p^{Erdos} = 3\%$	$p^{Erdos} = 5\%$
$ V $	1,909,226	9,000	18,000	27,000	9,000	9,000	9,000
$ E $	19,194,624	1,979,410	3,959,970	5,940,340	810,570	2,428,600	4,048,600
$\min(degree_v)$	1	203	204	205	56	214	369
$\max(degree_v)$	2,100	245	238	247	126	334	536
$\text{avg}(degree_v)$	10.05	220.12	219.99	220.01	90.06	269.84	449.84

Source: Adapted from Scabora *et al.* (2017).

Hardware and Software specifications. The hardware consists of a machine equipped with an Intel® Core™ i7-2600 @ 3.40GHz processor, 8GB of DDR3 1333MHz RAM memory, two SATA 6Gb/s 7200RPM hard disks set up in RAID 0, and Linux operating system Fedora release 25. The RDBMS employed in the experiments of this section is the PostgreSQL with version 9.5.6. For the Neo4j specifications, the Java Heap Size was dynamically calculated based on available system resources. For the memory configuration, we reserved 4GB for the operating system, 2GB for the Java Virtual Machine heap size (sufficient memory for the query execution in the Neo4j), and the remaining 2GB to the page cache size. No additional index structure nor other optimizations were employed in the Neo4j.

4.1.3.1 Overhead in Insert, Update, Delete and Bulk Load Operations

To measure the overhead in the individual insert, update, and delete operations using the table Edge- k discussed in Section 4.1.2.1, we analyzed each one over the DBLP dataset. The analysis consists of randomly generating 1,000 new edges and inserting them, followed by their respective update and delete operations. For the values of the parameter k , we employed $k = 1, 5, 10$, and 15 . Notice that $k = 1$ is a particular case where the table Edge- k corresponds to a

conventional edge table. Figure 13 shows the average time and the standard deviations, calculated based on the time to manipulate the 1,000 random edges. When comparing the table Edge- k to a conventional table, the overall overhead is about 50% for the insert, update, and delete operations. These operations take five milliseconds (ms) more than the base time of 10ms required by the conventional edge table.

Figure 13 – The average elapsed times for the insert, update, and delete operations when using the conventional edge table (*i.e.*, with $k = 1$) and the table Edge- k using $k = 5, 10$, and 15.

Source: Adapted from Scabora *et al.* (2018).

Moreover, one can notice that the elapsed times for the table Edge- k in each operation are similar despite the k value. This similarity indicates that the overhead is heavily influenced by searching the affected column id_{tgt}^i after loading all rows related to id_{src} from the disk. Hence, varying the k value does not substantially impact the processing time because the search for the column is performed in the main memory. Additionally, an overlap exists in the standard deviations of the table Edge- k times while varying the k value.

Based on the Sort, Cast, and Load steps introduced in Section 4.1.2.1, we compared the bulk loading time between the conventional edge table and the table Edge- k over the DBLP dataset. Figure 14 shows the average bulk loading processing times with k varying from 1 to 30. Notice that only the Load step is required for the conventional edge table (*i.e.*, $k = 1$). Although executing fewer steps than the table Edge- k , it presented higher processing times to complete. Considering the $k = 1$ (*i.e.*, the conventional edge table) and $k = 2$, one can observe a processing time reduction of 35%. As the k value increases, the processing time reduction stabilizes around $k = 10$ with a reduction of approximately 70%. Moreover, the processing time reduction during the bulk loading process is mainly related to the sharp decrease in the η_{rows}^k (see Equation 4.1).

Figure 14 – The average processing time of the bulk loading processes varying k . The bulk loading process is performed following the steps: Sort, Cast, and Load.

Source: Research data.

4.1.3.2 Reductions in Storage Size and Query Processing Times

The potential for the benefits of the table Edge- k mainly relies on the reduction of storage size. Based on Equation 4.1, by reducing the total number of row η_{rows}^k , it is possible to decrease the dataset storage size. Figure 15 show the dataset size in each dataset while varying the k value from 1 to 50. In every dataset, one can notice that the principal reduction occurs in the first values of k , especially with values between 2 and 15. For higher values of k , most of the neighborhood $degree_v$ of each vertex occupies a single e_k row. Hence, the reduction size stabilizes beyond this point. Consequently, higher k values result only in increasing the quantity of NULL values required η_{NULL}^k . As NULL values occupy an irrelevant space in the PostgreSQL RDBMS, having additional NULL values does not impact the total storage size, which is why the dataset size reduction remains stable even for higher k values.

Figure 15a shows the sizes of the DBLP dataset. As the value of k varies from 1 to 10, one can observe sharp size reductions. With k values greater than 20, the size reduction stabilizes, having a total reduction of around 70% when compared to the conventional edge table (*i.e.*, $k = 1$). Figures 15b and 15c presents the dataset sizes for both synthetic datasets Newman-Watts-Strogatz and Erdős-Rényi, respectively. In each one of them, varying the number of vertices $|V|$ (in Newman-Watts-Strogatz) and the probability p^{Erdos} (in Erdős-Rényi), their reduction behavior is similar. That is, in every synthetic dataset and their respective parameters, the table Edge- k achieved a size reduction ranging from 78% to 80% for $k \geq 30$.

Figure 15 – The dataset size reduction, in megabytes, achieved by the Edge- k table in both real and synthetic datasets while varying the k value from 1 to 50.

Source: [Scabora et al. \(2018\)](#).

Following, we analyzed the SSSP query processing times over the table Edge- k . The starting point for this query is the vertex having the largest neighborhood size $degree_v$. Figure 16 presents the average query-time to execute a SSSP query while varying k . For the DBLP dataset, we analyzed the execution times from 2 to 5 iterations of the SSSP query. The number of iteration refers to the quantity of *Recursive Join* operations, which consists of the number of times executing the SQL query in Source code 7. We discarded the 1st iteration alone because the same outcome could be obtained by directly querying the edge table, filtering it by the vertex id_{src} , and no performing any join operation. For both synthetic datasets, the shortest path to all vertices has been found using at most four iterations. Therefore, we executed the complete SSSP query without limiting the number of iterations.

Figure 16 – The average SSSP query processing time in three scenarios for varying k in the table Edge- k : the DBLP dataset with the 2nd, 3rd, and 4th iterations, and in both Newman-Watts-Strogatz and Erdős-Rényi synthetic datasets varying their respective parameters.

Source: Adapted from [Scabora et al. \(2018\)](#).

Figure 16a presents the average query-time to execute a SSSP query varying k when considering the 2nd, 3rd, and 4th iterations for the DBLP dataset. The maximum reduction

achieved is around 66% in the 2nd iteration compared to the conventional edge table (*i.e.*, $k = 1$). Regarding the average query times from all three iterations, the reduction was of up to 58% for $k = 18$. A similar behavior can be observed in Figures 16b and 16c, while varying their respective parameters. In Figure 16b the reduction varies from 50% to 54% for $k \geq 10$, and in Figure 16c the highest average reduction was of around 52% at $k = 30$. Notice that the behavior presented in every dataset is analogous to the dataset size reduction previously discussed in Figure 15. Based on their similar behavior, we calculated the Pearson coefficient between the number of disk pages of the table Edge- k (see Section 2.1.2), and the average elapsed time of SSSP queries while varying k value. For all datasets, the Pearson correlation varied from 0.83 to 0.94. Accordingly, reducing the number of rows η_{rows}^k , more specifically, the table Edge- k size (in megabytes), also lowers the query processing time. As expected, reducing the dataset size minimizes the number of I/O operations and optimizes the query executions over it.

Additionally, we also carried a scalability analysis to identify the highest query processing time reduction achieved regarding the number of iterations (in the DBLP dataset) and the parameter employed while generating the synthetic graphs Newman-Watts-Strogatz and Erdős-Rény. Figure 17 summarizes the largest reduction of the SSSP query processing time achieved in each dataset. Figure 17a shows the reductions in the DBLP dataset, whereas increasing the number of iterations results in decreasing the query time reductions. This implies that the SSSP query performance tends to degrade as we iteratively perform *Recursive Join* operations. Nevertheless, executing many joins is not usually well-suited for an RDBMS, as previously discussed. Considering the application scenario explored, denoting researchers and their collaborations, we focused on the more recurrent situation in which one searches for collaborations that have up to four researchers in its shortest path. Hence, it would not be particularly interesting to execute more SSSP query iterations in this scenario.

Figure 17 – The highest SSSP query processing time reductions using Edge- k varying: (a) the number of iterations in the DBLP dataset; (b) the numbers of vertices $|V|$ for Newman-Watts-Strogatz dataset; and (c) the probability p^{Erdos} for Erdős-Rény dataset.

Source: Scabora *et al.* (2018).

Figures 17b and 17c shows the reductions in both synthetic datasets Newman-Watts-Strogatz and Erdős-Rény, respectively. In both figures, one can observe an increase in the SSSP

query processing time reduction, showing that our proposal provides even better results as the volume of data grows regarding the number of vertices (in Newman-Watts-Strogatz dataset) and the number of edges (in Erdős–Rényi dataset). That is, more massive graphs allow higher reductions in both SSSP query processing time and dataset size occupied by the edge table. All in all, these experimental results show the potential of the table Edge- k to achieve good performance improvements in analyses based on shortest paths. For the DBLP dataset, the reduction in SSSP query processing time decreases as the number of iterations increases. However, this issue is already known in many RDBMS-based solutions that require several join operations.

4.1.3.3 Comparison with Neo4j NoSQL Database

To further evaluate the previous experiments, we compared the Edge- k and conventional edge tables with the Graph-oriented NoSQL System named Neo4j database. Due to the sharp reduction in both dataset size and SSSP query processing time for k values up to 15, the comparison performed in this section considered the table Edge- k using $k = 10$. We implemented both distribution degree and SSSP queries in the Neo4j using the Java API. The implementation considered a fair comparison between both databases by executing the same algorithm.

As previously described (see Section 2.2.2), the distribution degree query returns the neighborhood $degree_v$ sizes of each vertex $v \in V$ and counting the size frequencies. This query implementation in Neo4j consists of performing a loop through all vertices, counting the number of neighbors for each vertex. Using the conventional edge table, such query relies in the SQL functions GROUP BY (on the id_{src}) and COUNT (on the id_{tgt}). For the Edge- k table, this query requires adapting the COUNT function to consider multiple edges in the same row e_k . Figure 18 shows the elapsed time in each approach, where even Neo4j outperforming the conventional edge table, the table Edge- k outperformed both competitors.

Figure 18 – The average elapsed time for executing distribution degree queries comparing both Edge- k and conventional edge tables with the Neo4j database.

Source: Research data.

Regarding the SSSP query type, its implementation in Neo4j database employed the *HashMap* class⁶ to represent the temporary table *T* used in the *Recursive Join* operations. This analysis includes the variation of the starting vertex id_{src} used in the SSSP query. The variation enables exploring distinct behaviors related to the number of vertices already computed in the current and previous recursive steps. For example, if one starts a query with a remote vertex, which is a vertex containing only a few vertices in its neighborhood, the SSSP query will require more recursive steps to reach a considerable amount of vertices. Hence, the *Recursive Join* operation will have smaller *T* tables, resulting in shorter processing times.

Based on this reasoning, we randomly chose twenty distinct starting vertices. Half of them have a small neighborhood size, and the other half have larger neighborhood sizes, which are near to the maximum size. We executed a SSSP query for each starting vertex, varying the number of iterations from 1 to 15. At the 15th iteration, we stopped the query because their result table did not change anymore. Each SSSP query iteration of each starting vertex was executed three times in order to minimize any query processing time deviation. Figure 19 shows the elapsed times per SSSP query iteration.

Figures 19a to 19c show a boxplot graphic for the approaches using the table *Edge-k*, the conventional edge table and in the Neo4j NoSQL, respectively. Comparing Figures 19a and 19b, one can observe that both have a similar uniform growth in the elapsed time as the number of iteration increases. For the Neo4j in Figure 19c, there is a sharp growth in the elapsed times regarding the first 5 iterations. This behavior is a consequence of intensive disk read operations, caused by the sharp growth in the number of reached vertices per iteration. After the 5th iteration, the tendency flattens because Neo4j has already load most of the graph data in memory. Such finding is related to the performance of Neo4j regarding the size of intermediary query results, as previously discussed in Section 2.2.3.

Additionally, Figure 19d shows the average time for all three approaches, varying the starting vertex and the number of iterations. It is possible to notice that in the first iteration, the SSSP query executed in Neo4j was 10 times faster than both conventional edge and *Edge-k* tables. However, in the following iterations, due to the sharp growth observed in Figure 19c, both RDBMS-based solutions outperformed Neo4j. In the 15th iteration, Neo4j finally outperformed the RDBMS-based solutions, but with an execution time very similar to the table *Edge-k*. Notably, the table *Edge-k* always outperformed the conventional edge table for any number of iterations. These results are another interesting finding because, in this section, we evaluated the SSSP query with different starting vertices and explored another query as well, the degree distribution, for which the table *Edge-k* enabled better results.

⁶ <<https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>>

Figure 19 – The elapsed time comparison for the SSSP query execution regarding the Neo4j and the conventional edge and Edge- k tables, while varying the starting vertex and the number of iterations from 1 to 15.

Source: Adapted from [Scabora et al. \(2018\)](#).

4.1.4 Final Remarks over Edge- k

Table Edge- k represents our first specialized storage approach for grouping edges into the same rows. It tackles the problem of a prevalence of a singular approach to store the edges of a graph and perform queries or analyses. We proposed the table Edge- k , an alternative storage approach for grouping each vertex neighborhood in a few or even in a single tuple. Moreover, to maintain robustness and maintain compatibility to the Relational Model, we established a predefined maximum number k when grouping multiple edges in a single row. Consequently, we observed a reduction in both dataset size and processing times for the SSSP and distribution degree queries. In fact, using the table Edge- k enabled reductions of over 50% in the DBLP, Newman-Watts-Strogatz, and Erdős-Rényi datasets. Those results highlight the positive impact

of the table *Edge-k* compared to existing RDBMS-based frameworks (*i.e.*, using the conventional edge table) and the Graph-oriented NoSQL System named Neo4j database.

4.2 Block-Level Grouping: Clustered Tables

This section presents another contribution for a specialized storage approach: grouping multiple edges into pages or blocks of the disk. In this section, we explore the adjacency-list representation of edges using Clustered Tables' physical designs. By using Clustered Tables, we tied the neighborhood of each edge into chunks or blocks, which enables grouping edges originated from the same vertex in the same disk's pages. Consequently, the physical design provided by the Clustered Tables enabled reducing both Recursive Query execution times and the overall quantity of I/O operations required by them. The analysis employing Clustered Tables was explored in [Scabora *et al.* \(2020\)](#). The remainder of this section discusses its main findings.

4.2.1 The Clustered Tables Proposal and Methodology

There are two possible physical designs for the Clustered Table proposal. The first design consists of grouping each vertex and its neighborhood into a single disk page. We named it as *Combined Tables*, which consists of defining a cluster to store the data from both vertex and edge tables, *i.e.*, tables *V* and *E*, respectively, into the same chunks or blocks in the disk. The second design consists of reorganizing only the rows of the table *E*, creating chunks to group them based on the source vertex attribute id_{src} . We named it *Edge Clustered Table*, which is the most similar to the *Edge-k* technique (see Section 4.1): however, although *Edge-k* groups multiple edges into the same tuple and *Edge Clustered Table* is based on the same reasoning, the latter groups edges into the same cluster. Figure 20 illustrates the conventional tables alongside with both proposed physical designs named *Combined Tables* and *Edge Clustered Table*.

Figure 20(a) shows the conventional tables, having both vertices (in table *V*) and edges (in table *E*) with no optimization. Figure 20(b) shows the *Combined Tables* physical design, where we define a cluster for each vertex identifier (*i.e.*, attribute *id* from table *V*), storing the vertex tuple together with all its attributes. Within the same cluster, we store all tuples from table *E* where $id_{src} = id$, consisting of the edges from the neighborhood of the respective vertex *id*. *Combined Tables* aims at optimizing the performance of the first *Recursive Join* operation (see Definition 1) carried out over tables *V* and *E*, corresponding to executing Egonet queries (see Section 2.2.2). Figure 20(c) shows the *Edge Clustered Table* physical design, which defines the groups based on the source vertex identifier id_{src} , and without modifying the table *V*. *Edge Clustered Table* enables reducing the processing time of recursive queries because most of the adjacency list of every vertex will be stored in adjacent regions in the disk.

In both *Combined Tables* and *Edge Clustered Table* designs, we employed the physical data organization named Clustered Tables, from Oracle RDBMS, to group the data in disk

according to its cluster key (see Section 2.1.2). Source code 8 presents the creation of tables V and E using Clustered Tables physical design. Line 1 creates the cluster named `clustered` over the attribute `id`, which is named `cluster` key of the Clustered Table. The size of each cluster is denoted as `cluster_size` and varies in each physical design explored in this section (see Section 4.2.1.1 for further details). Line 2 creates a index over the `cluster` key `id` to optimize queries using it. Lines 3 to 7 and 8 to 14 creates the tables V and E employing Clustered Tables. The difference when using the Clustered Tables physical designs is the Lines 7 and 14, where we specify a cluster when creating each table. Notice that the Source code 8 define both tables in the same cluster, which refers to the *Combined Tables* physical design. For the *Edge Clustered Table*, we remove the cluster specification at Line 7 since this physical design only employs the Clustered Tables over table E .

Figure 20 – An example schema for storing graph data and its three physical representations: (a) conventional representation for the tables V and E ; (b) grouping the tuples from both tables into clusters using Clustered Tables; and (c) grouping only the tuples from table E with the Clustered Table’s physical design.

Source: Adapted from Scabora *et al.* (2020).

Nevertheless, the SQL adaptation required to employ Clustered Tables’ physical designs is performed only during the creation of tables V and E . Compared to table *Edge- k* , the Clustered Tables do not require any adaptation regarding the insert, update, and delete operations, neither modifying the *Recursive Join* operation previously detailed in Section 4.1.2. Next, we discuss methods to determine the value of cluster size employed in the Line 1 of the SQL Source code 8, regarding both *Combined Tables* and *Edge Clustered Table* physical designs.

```

1: CREATE CLUSTER clustered (id INTEGER) SIZE cluster_size;
2: CREATE INDEX clustered_idx ON CLUSTER clustered;
3: CREATE TABLE V (
4:     id            INTEGER PRIMARY KEY,
5:     latitude      FLOAT,
6:     longitude     FLOAT
7: ) CLUSTER clustered (id);
8: CREATE TABLE E (
9:     id_src        INTEGER,
10:    id_tgt         INTEGER,
11:    distance       FLOAT,
12:    time           FLOAT,
13:    CONSTRAINT E_pk PRIMARY KEY (id_src, id_tgt)
14: ) CLUSTER clustered (id_src);

```

Source code 8 – Creation of tables *V* and *E* using the Clustered Tables physical designs.

4.2.1.1 Cluster Size estimation for employing Clustered Tables' physical designs

We employ two techniques to estimate the cluster sizes. For the *Combined Tables* physical design, we defined a cluster size large enough to store the vertex data and all of its adjacent edges. Hence, since it aims to speed up Egonet queries' execution, this cluster size enables the maximum reduction in the I/O operations. Distinctly, for the *Edge Clustered Table* physical design, we had chosen not to store all the neighborhood into a single cluster. The cluster sizes for *Edge Clustered Table* are based on the Pareto principle, using the 80/20 rule (see Section 2.2).

The Pareto principle relies upon the intuition that 20% of a population concentrates 80% of all the income, and it is considered a representative distribution that appears in several scenarios regarding complex networks or graph data (see Section 2.2.1). Based on this principle, we define cluster sizes corresponding to the 20th and 80th percentiles of the vertex distribution degree values, which we refer to as $degree^{20th}$ and $degree^{80th}$, respectively. Accordingly, we adopt the Pareto principle to create two cluster size configurations, one for storing 20% of the adjacency list and another for the 80% of the adjacency list. Equation 4.4 show how to calculate the cluster size $cluster_size^{ith}$ for any i -th percentile when employing the *Edge Clustered Table*.

$$cluster_size^{ith} = \left(degree^{ith} \times maxsize(e) \right), 0 \leq i \leq 100 \quad (4.4)$$

where $maxsize(e)$ is the maximum size, in bytes, of the tuples from table *E*, and $degree^{ith}$ is the degree value in the i -th percentile of the vertex distribution degree. The $maxsize(e)$ is given by $maxsize(e) = \max_{e \in E} (size(e))$, where $size(e)$ is a function that returns the size, in bytes, of the tuple $e \in E$. Due to the fact that we adopt the Pareto principle, we only used the parameters $i = 20$ and $i = 80$. However, this equation can be calculated using other percentiles values.

4.2.1.2 Application Scenario for Clustered Tables

The application scenario explored with Clustered Tables refers to the Urban Streets (see Table 1), denoting a Georeferenced Graph (see Section 2.2.1). The dataset explored, named *DIMACS*, is available by the 9-th implementation challenge for shortest paths⁷. *DIMACS* dataset is composed of $|V| = 23,947,347$ vertices and $|E| = 57,708,624$ edges. Each vertex $v \in V$ has an identifier (*id*) and its coordinates (latitude and longitude), where the data types are integer for attribute *id* and float for both attributes latitude and longitude. Each edge $e \in E$ has the identifiers for source and target vertices (id_{src} and id_{tgt} , respectively, both of integer data type). In the *DIMACS* dataset, there are two weights in each edge: the distance (in meters) and the estimated transit time over this street segment. Both weights are represented as float data type.

The application scenario evaluated over the *DIMACS* dataset consists of performing SSSP queries. As an example, consider an application in which one is searching for the closest patient's residence from it. In this scenario, we start a SSSP query by retrieving the vertex identifier *id* where the hospital in the starting-point query (see Equation 3.1a). Then, in each recursive-step query (see Equation 3.1b), we visit new vertices searching for the location of these closest patients in the graph up to a maximum distance. The complete SSSP query was previously presented in Source code 4, which retrieves the vertices up to a maximum distance as already illustrated in the application scenario of Section 3.1.3. Notice that the physical design optimization proposed, using Clustered Tables, does not modify any SQL statement in Source code 4 because it is present only while creating the tables from the schema.

4.2.2 Experimental Evaluation using Clustered Tables

This section covers three analyses performed over the Clustered Tables physical design proposed, and it is divided into three parts. Section 4.2.2.1 evaluates SSSP query performance when starting from a single starting-point while varying the maximum distance. In Section 4.2.2.2, we analyze the performance of the *Recursive Join* operation in the first recursive step when varying the starting-point query selectivity. Finally, Section 4.2.2.3 combines the analysis of both Sections 4.2.2.1 and 4.2.2.2 by evaluating the overall SSSP query performance by varying both starting-point query selectivity and maximum distance.

Dataset Setup. Figure 21 shows the main statistics from the *DIMACS* dataset. Figure 21a shows the vertex distribution degree, which varies from 1 to 9. Considering the *Edge Clustered Table* physical design, we highlighted the values for $degree^{20th}$ and $degree^{80th}$, which correspond to the values 2 and 3, respectively. Figure 21b shows a histogram of the edge tuple sizes, in bytes, from the table *E*, which vary from 16 to 25. Based on Figure 21 statistics, we calculated the cluster sizes for both *Combined Tables* and *Edge Clustered Table* physical designs. In *Combined Tables*, the cluster size is 256 bytes, which is large enough

⁷ <<http://users.diag.uniroma1.it/challenge9/download.shtml>>

to accommodate all nine edges, each one having 25 bytes, plus the vertex tuple size in table V , whose maximum value is 21. In *Edge Clustered Table*, $cluster_size^{20th} = 2 \times 25 = 50$ bytes and $cluster_size^{80th} = 3 \times 25 = 75$ bytes, according to Equation 4.4. Consequently, Figure 21c shows the resulting size of all physical designs analyzed. Notice that conventional design, with no optimization, is less space-consuming than both *Edge Clustered Table* physical designs by approximately 19%. The difference between the percentiles of 20% and 80% in *Edge Clustered Table* is minimal, consisting of just 5 Megabytes (MB). Finally, *Combined Tables* physical design demands above three times more space than all other designs.

Figure 21 – The statistics obtained from the *DIMACS* dataset, (a) vertex distribution degree, (b) histogram of the edge tuple sizes, and (c) structure sizes for the Clustered Tables.

(a) Vertex distribution degree.

(b) edge tuple sizes.

(c) Structure sizes for the *DIMACS* dataset.

Source: Adapted from Scabora *et al.* (2020).

Hardware and Software specifications. The experiments were carried out using Oracle RDBMS 19c Enterprise Edition 19.3.0.0.0 on Docker Engine 19.03.1, running in a computer equipped with an Intel Core i7-8700 @ 3.20GHz processor, 16GB of DDR4 2,400 MT/s RAM memory, SATA 6.0 Gb/s 7,200RPM hard disk, and Linux operating system Ubuntu 18.04.

4.2.2.1 Recursive Query starting from a Single Point

This section evaluates the SSSP query processing performance when the starting-point query retrieves only one element. The analysis corresponds to starting from a single public facility (*e.g.*, a hospital). This is in the case for $|T^0| = 1$ in Equation 3.1a). We executed a SSSP query for five randomly chosen starting vertices, 100 times each, in all explored physical designs. In this analysis, we compared the query performance based on two measurements: (i) the average query processing time; and (ii) the number of buffer gets. For the query processing time (i), we discarded 30% of the shortest and 30% of the longest values before computing the average in order to filter out noisy results. For the number of buffer gets (ii), which consists of the number of data block requests by Oracle RDBMS, getting the block directly from main memory (if the block is already there), or reading it from disk if needed. In other words, the number of buffer gets measures the amount of I/O operations in the SSSP query. Both measurements (i) and (ii) were acquired from the internal management table named V\$SQL⁸. Figure 22 shows the results of both measurements.

Figure 22 – The average SSSP query processing time (a) and the number of buffer gets (b) over the DIMACS dataset employing Clustered Tables starting from a single point.

Source: Adapted from Scabora *et al.* (2020).

⁸ <https://docs.oracle.com/cd/B19306_01/server.102/b14237/dynviews_2113.htm>

For all four physical designs (*i.e.*, conventional, *Combined Tables*, *Edge Clustered Table 20%*, and *Edge Clustered Table 80%*), we evaluated the SSSP query execution by varying the maximum reachable distance from 500 meters to 40 kilometers. Figure 22a shows the average times, where *Edge Clustered Table 20%* is faster than other physical designs. It achieved a maximum time reduction of 22.5% when compared to conventional design, 24.0% when compared to *Edge Clustered Table 80%*, and 21.1% when compared to *Combined Tables*. Figure 22b shows that all physical designs employing Clustered Tables reduced the required data blocks when processing recursive queries by approximately 61% when compared to conventional design. However, when the SSSP query retrieves paths with lengths greater than 20 kilometers, the number of buffer gets converges for all physical designs. For paths larger than 20 kilometers, which corresponds to approximately 24 *Recursive Join* operations, the number of required data blocks is similar in all evaluated physical designs. Nevertheless, *Edge Clustered Table 20%* remains faster than all other evaluated designs in recursive queries starting from a single location.

4.2.2.2 Egonet queries starting from Multiple Points

Next, we evaluate the first *Recursive Join* operation in the SSSP query processing when having multiple starting-point vertices. This type of query, which is also known as the Egonet query, aims at retrieving the neighborhood of a vertex (see Section 2.2.2). Due to the variation in the starting-point query selectivity, we are exploring the case when $|T^0| > 1$ in Equation 3.1a. Figure 23 shows the variation of the starting-point query selectivity and its impact over the Egonet query processing performance considering the different physical designs. Figure 23a shows the vertex distribution degree of each subset vertices retrieved for each selectivity value.

Figure 23 – The average query processing time of Egonet queries for the four evaluated Clustered Tables’ physical designs, (a) the vertex distribution degree of each subset of vertices while varying the selectivity, (b) the corresponding query processing times.

Source: Adapted from Scabora *et al.* (2020).

Each curve in Figure 23a corresponds to a selectivity value, where each subset of vertices follows a similar distribution compared to the vertex degree distribution of all vertices in the graph. Based on these selectivity values, Figure 23b shows the average query processing time of the Egonet query for each subset of vertices. One can observe that, for selectivity values near 0.83, *Combined Tables* physical design reduced query processing time by 50.0% when compared to the conventional design. With respect to both *Edge Clustered Table* physical designs, the query processing time reduced by 37.0% in *Edge Clustered Table* 20% and by 35.6% in *Edge Clustered Table* 80%. Considering only the Clustered Tables physical designs, the *Combined Tables* reduced the query processing time by 45.8% when compared to *Edge Clustered Table* 20%. This analysis explored the potential of Clustered Tables physical design while focusing only on the first *Recursive Join* operation in the SSSP query.

4.2.2.3 Recursive Query starting from Multiple Points

The third analysis combines the studies from Sections 4.2.2.1 and 4.2.2.2. It consists of evaluating the overall SSSP query processing time when starting from multiple vertices. Hence, executed the SSSP query starting from subsets of vertices with sizes $|T^0| = 5,000, 10,000, 20,000, 50,000,$ and $100,000$. Figure 24 shows the average SSSP query processing time for the multiple subset of vertices T^0 when varying the maximum distance from 1 to 5 kilometers.

One can observe that the *Edge Clustered Table* 20% physical design had the best performance in all graphics, enabling an average query processing time reduction, when compared to the Conventional design, of up to 9.3% when $|T^0| = 5,000$, 10.2% when $|T^0| = 10,000$, 15.0% when $|T^0| = 20,000$, 19.9% when $|T^0| = 50,000$, and 10.7% when $|T^0| = 100,000$. Moreover, the time reduction increased when $|T^0|$ varied from 5,000 to 50,000, and after 50,000 it starts to decrease. Such behavior occurred because too many clusters were accessed in the *Recursive Join* operation due to a more significant reduction of the query selectivity value.

Concluding, the *Combined Tables* physical design presented the worst performance in all graphics from Figure 24. Even though this physical design improved the execution performance in the first *Recursive Join* operation, it did not reduce the overall SSSP query processing time. Finally, *Edge Clustered Table* 80% physical design has a similar query processing performance when compared to the Conventional design in all graphics from Figure 24.

4.2.3 Final Remarks over Clustered Tables approaches

Both *Combined Tables* and *Edge Clustered Table* physical designs correspond to our second specialized storage approach for grouping multiple edges into the same disk's pages. By using Clustered Tables, we introduced a storage method based on adjacency-lists, which reorganizes the data within the tables. With these physical designs, we were able to optimize the execution of SSSP queries inside an RDBMS without requiring SQL query modifications, as present in the *Edge-k* proposal. The optimization refers to reducing disk I/O operations, and, as

a consequence, the query processing time was remarkably shorter. When starting from a single vertex, the results showed a reduction of up to 22.5% in the SSSP query processing time when compared to no physical design. We also observed a reduction of up to 61% in the number of disk's pages accessed when using Clustered Tables physical design. For Egonet queries, the physical designs enabled a reduction of up to 50% in the query processing time. Moreover, when starting from multiple vertices simultaneously, our second specialized storage achieved a reduction of approximately 20% for the SSSP query processing time. All in all, the best results came when using the *Edge Clustered Table* physical design with $cluster_size^{20th}$.

Figure 24 – The average query processing times of the SSSP query for $|T^0| = 5,000, 10,000, 20,000, 50,000,$ and $100,000$.

Source: Adapted from [Scabora et al. \(2020\)](#).

4.3 Final Considerations

In this chapter, we explored adjacency-based specialized storage approaches over the edge table E . We proposed both tuple-based and block-based specializations, under the names of table *Edge- k* and *Edge Clustered Table* physical design, and experimentally showed improvements in graph query executions' performance. On the one hand, the table *Edge- k* provided the highest performance improvements, but requiring adaptations in the recursive queries executed, as described in Source code 7. On the other hand, the *Edge Clustered Table* physical design also enables performance improvements without requiring any modifications in both recursive queries and insert, update, and delete operations, however, providing fewer execution time reductions when compared with table *Edge- k* . All in all, we addressed the **Question 2** by proposing and validating two adjacency-based specialized storage approaches when dealing with graph queries executed using Recursive Queries.

CONCLUSION

This Ph.D. research focuses on the development of novel specialized storage approaches for optimizing the execution of Recursive Queries in RDBMSs while processing graph queries. Specifically, we presented three main contributions that we summarize next, together with the research questions addressed during this research and the overall thesis. This chapter is organized as follows. Section 5.1 highlights the main contributions of this thesis. Section 5.2 describes future works opened for research as a result of this Ph.D. project. Section 5.3 summarizes the list of publications originated from this research and from collaborations that occurred during this research execution.

5.1 Contributions

As stated in Chapter 1, the main goal of this Ph.D. project was answering the following question: “*How can we improve the performance of queries over graph data executed in a RDBMS?*”. To tackle the problem, we divided it into two questions: modifying each of the tables employed by the Recursive Queries, the table of edges E , and the temporary result table T . From this goal, we produced three main contributions:

1. The SHARq framework, for enriching the temporary result table T employed in the recursive queries to enable reusing it across multiple graph queries. By adding extra columns to table T , we can reuse the join operations, supporting the execution of multiple SSSP queries and combining the graph query types SSSP, CC, and PR. The enriched table T provides increased efficiency to execute graph queries, thus answering the **Question 1** from this Ph.D. research;
2. The Edge- k method, which groups multiple edges from the same vertex neighborhood into the same tuple. The resulting table, named table Edge- k , stores a predefined maximum number k of edges by using extra columns for storing their destination vertex identifiers

and respective properties. Employing table *Edge-k* in recursive queries, in particular for the distribution degree and the SSSP queries, significantly improved their performance. Hence, this contribution successfully answered the **Question 2** from this Ph.D. research; and

3. The *Edge Clustered Table* physical design, which groups multiple edges into a single disk page using Clustered tables. This third contribution shares similar characteristics with the table *Edge-k*. However, the *Edge Clustered Table* proposal only requires simple modeling adaptations when creating the tables without modifying the recursive queries. It provides reductions of both time and disk I/O operations while executing SSSP and Egonet queries. The *Edge Clustered Table* also answers the **Question 2**, showing that there may be several alternatives for adjacency-list storage approaches, which can hopefully be further explored in the future.

Based on the three contributions, this Ph.D. research improved recursive queries' performance both horizontally and vertically. By horizontally, we mean optimizing batches of multiple graph queries, sharing their join operations executed recursively. The term vertically refers to reducing the elapsed time and the computer resource usage in each join operation from an individual graph query. Accordingly, our results allow stating that our thesis was successfully answered:

Thesis. *By specializing data structures (i.e., tables, indexes, and internal data representations) in a RDBMS using adjacency-based storage approaches, it is possible to improve the performance of graph queries that depends on the several join operations involved in recursive query executions.*

In fact, we observed that adjacency-list specialized storage approaches optimized the execution of graph queries represented by recursive queries in all the applications evaluated. The contributions discussed here and the results achieved during this Ph.D. research point out a range of future developments, detailed in the next section.

5.2 Future Work

Some of the main future works envisioned from this Ph.D. research are the following:

- Regarding the SHARq framework, its extension would be proposing a continuous operator for executing the join operations from recursive queries. The continuous operator consists of continuously performing join operations between the result table *T* and the edge table *E* whenever graph queries are required, employing query rewriting techniques to add

(when needed) and subtract (when finished) every graph query performed on the fly with additional columns;

- Regarding table *Edge-k* and *Edge Clustered Table* physical design, the most straightforward future work would be evaluating how to automatically determine their parameters, such as k and *cluster_size* values. The first step in this direction would be exploring: other graph types, different datasets, and varying the number of properties stored in the edge table (instead of only a single weight value per edge);
- Finally, combining the enhancements of tables T (in SHARq) and the edge table (with table *Edge-k* and *Edge Clustered Table* physical design) would be beneficial to integrate the efficient execution of graph queries in RDBMS. Hence, combining all these techniques proposed during this Ph.D. research will reduce the gap for efficiently executing graph queries in RDBMSs.

5.3 List of Publications

The publications in this section are divided into main publications, referring to the ones directly resulted from this Ph.D. research, and the collaborative publications originated from contributions with the research group GBdI (Data Bases and Images Group). Both main and collaborative publications helped in the learning process and formation of the Ph.D. candidate. They are listed as follows.

Main publications:

- **SBBD 2017 (SCABORA *et al.*, 2017)**: Lucas C. Scabora; Paulo H. Oliveira; Daniel S. Kaster; Agma J. M. Traina; Caetano Traina-Jr. **Relational graph data management on the edge: Grouping vertices' neighborhood with Edge-k**. In the 32th Brazilian Symposium on Databases, SBC, 2017, p. 124–135.
- **JIDM 2018 (SCABORA *et al.*, 2018)**: Lucas C. Scabora; Paulo H. Oliveira; Gabriel Spadon; Daniel S. Kaster; José F. Rodrigues-Jr.; Agma J. M. Traina; Caetano Traina-Jr. **Cutting-edge Relational Graph Data Management with Edge-k: From One to Multiple Edges in the Same Row**. In the Journal of Information and Data Management, 9(1), p.20–35.
- **SBBD-DSW 2019 (SCABORA *et al.*, 2019)**: Lucas C. Scabora; Gabriel Spadon; Lucas S. Rodrigues; Mirela T. Cazzolato; Marcus V. S. Araujo; Elaine P. M. de Sousa; Agma J. M. Traina; José F. Rodrigues-Jr.; Caetano Traina-Jr. **G-FranC: a dataset of criminal activities mapped as a complex network in a relational DBMS**. In Dataset Show Case of the 34th Brazilian Symposium on Databases, SBC, 2019, p. 366–376.

- **SAC 2020 (SCABORA *et al.*, 2020)**: Lucas C. Scabora; Gabriel Spadon; Paulo H. Oliveira; José F. Rodrigues-Jr.; Caetano Traina-Jr. **Enhancing recursive graph querying on RDBMS with data clustering approaches**. In Proceedings of the 35th SIGAPP Symposium on Applied Computing, ACM, 2020, p. 404–411.
- **SAC 2021 (SCABORA *et al.*, 2021)**: Lucas C. Scabora; Gabriel Spadon; Mirela T. Cazzolato; Daniel S. Kaster; Agma J. M. Traina; José F. Rodrigues-Jr.; Caetano Traina-Jr. **SHARq: Sharing Recursive Queries in Relational Databases**. In Proceedings of the 36th SIGAPP Symposium on Applied Computing, ACM, 2021, p. 336–339.

Collaborative contributions:

- **SBBB 2017 (OLIVEIRA *et al.*, 2017b)**: Paulo H. Oliveira; Lucas C. Scabora; Mirela T. Cazzolato; Marcos V. N. Bedo; Agma J. M. Traina; Caetano Traina-Jr. **MAMMOSET: An Enhanced Dataset of Mammograms**. In Proceedings of the Satellite Events of the 32nd Brazilian Symposium on Databases, SBC, 2017, p. 256–266.
- **SAC 2017 (OLIVEIRA *et al.*, 2017)**: Paulo H. Oliveira; Lucas C. Scabora; Caetano Traina-Jr.; Daniel S. Kaster. **One index to dominate them all: domain indexes for improving queries across multiple tables**. In Proceedings of the 32th SIGAPP Symposium on Applied Computing, ACM, 2017, p. 900–905.
- **CBMS 2017 (OLIVEIRA *et al.*, 2017a)**: Paulo H. Oliveira; Lucas C. Scabora; Mirela T. Cazzolato; Willian D. Oliveira; Agma J. M. Traina; Caetano Traina-Jr. **Efficiently Indexing Multiple Repositories of Medical Image Databases**. In Proceedings of the 30th International Symposium on Computer-Based Medical Systems, IEEE, 2017, p. 286–291.
- **CBMS 2017 (CAZZOLATO *et al.*, 2017)**: Mirela T. Cazzolato; Lucas C. Scabora; Alceu F. Costa; Marcos R. Nesso-Jr.; Luis F. M. Oliveira; Daniel S. Kaster; Caetano Traina-Jr.; Agma J. M. Traina. **BREATH: Heat Maps Assisting the Detection of Abnormal Lung Regions in CT Scans**. In Proceedings of the 30th International Symposium on Computer-Based Medical Systems, IEEE, 2017, p. 248–253.
- **ICCS 2017 (SPADON *et al.*, 2017)**: Gabriel Spadon; Lucas C. Scabora; Paulo H. Oliveira; Marcus V. S. Araujo; Bruno B. Machado; Elaine P. M. de Sousa; Caetano Traina-Jr.; José F. Rodrigues Jr. **Behavioral Characterization of Criminality Spread in Cities**. In Procedia Computer Science, 108, 2017, p. 2537–2541.
- **ITNG 2017 (SPADON *et al.*, 2017)**: Gabriel Spadon; Lucas C. Scabora; Marcus V. S. Araujo; Paulo H. Oliveira; Bruno B. Machado; Elaine P. M. de Sousa; Caetano Traina-Jr.;

- José F. Rodrigues-Jr. **Complex Network Tools to Understand the Behavior of Criminality in Urban Areas**. In *Information Technology - New Generations*, Springer International Publishing, 2018, p. 493–500.
- **SBBD 2018 (SPADON *et al.*, 2018)**: Gabriel Spadon; Lucas C. Scabora; Marcos R. Nesso-Jr.; Caetano Traina-Jr.; José F. Rodrigues-Jr. **Caracterização Topológica de Redes Viárias por Meio da Análise de Vetores de Características e Técnicas de Agrupamento**. In *Proceedings of the 33rd Brazilian Symposium on Databases, SBC*, 2018, p. 157–168.
 - **SAC 2018 (CHINO *et al.*, 2018)**: Daniel Y. T. Chino; Lucas C. Scabora; Caetano Traina-Jr.; Agma J. M. Traina. **BoSS: image retrieval using bag-of-superpixels signatures**. In *Proceedings of the 33rd SIGAPP Symposium on Applied Computing, ACM*, 2018, p. 309–312.
 - **CBMS 2018 (CHINO *et al.*, 2018)**: Daniel Y. T. Chino; Lucas C. Scabora; Mirela T. Cazzolato; Ana Elisa Serafim Jorge; Caetano Traina-Jr.; Agma J. M. Traina. **ICARUS: Retrieving Skin Ulcer Images through Bag-of-Signatures**. In *Proceedings of the 31st International Symposium on Computer-Based Medical Systems, IEEE*, 2018, p. 82–87.
 - **CBMS 2018 (NESSO-JR. *et al.*, 2018)**: Marcos R. Nesso-Jr.; Mirela T. Cazzolato; Lucas C. Scabora; Paulo H. Oliveira; Gabriel Spadon; Jéssica Andressa de Souza; Willian D. Oliveira; Daniel Y. T. Chino; José F. Rodrigues-Jr.; Agma J. M. Traina; Caetano Traina-Jr. **RAFIKI: Retrieval-Based Application for Imaging and Knowledge Investigation**. In *Proceedings of the 31st International Symposium on Computer-Based Medical Systems, IEEE*, 2018, p. 71–76.
 - **CBMS 2019 (ZABOT *et al.*, 2019b)**: Guilherme F. Zabet; Mirela T. Cazzolato; Lucas C. Scabora; Bruno S. Façal; Agma J. M. Traina; Caetano Traina-Jr. **UCORM: Indexing Uncorrelated Metric Spaces for Concise Content-Based Retrieval of Medical Images**. In *Proceedings of the 31nd International Symposium on Computer-Based Medical Systems, IEEE*, 2019, p. 306–311.
 - **CMPB 2019 (CAZZOLATO *et al.*, 2019)**: Mirela T. Cazzolato; Lucas C. Scabora; Marcos R. Nesso-Jr.; Luis F. M. Oliveira; Alceu F. Costa; Daniel S. Kaster; Marcel Koenigkamt-Santos; Paulo M. de A. Marques; Caetano Traina-Jr.; Agma J. M. Traina. **dp-BREATH: Heat maps and probabilistic classification assisting the analysis of abnormal lung regions**. *Journal of Computer Methods and Programs in Biomedicine*, 173, 2019, p. 27–34.
 - **SBBD 2019 (CAZZOLATO *et al.*, 2019)**: Mirela T. Cazzolato; Lucas S. Rodrigues; Lucas C. Scabora; Guilherme F. Zabet; Guilherme Q. Vasconcelos; Daniel Y. T. Chino; Ana Elisa Serafim Jorge; Robson L. F. Cordeiro; Caetano Traina-Jr.; Agma J. M. Traina.

- A DBMS-Based Framework for Content-Based Retrieval and Analysis of Skin Ulcer Images in Medical Practice.** In Proceedings of the 34th Brazilian Symposium on Databases, SBC, 2019, p. 109–120.
- **ISM 2019 (ZABOT *et al.*, 2019a):** Guilherme F. Zabot; Mirela T. Cazzolato; Lucas C. Scabora; Agma J. M. Traina; Caetano Traina-Jr. **Efficient Indexing of Multiple Metric Spaces with Spectra.** In International Symposium on Multimedia, IEEE, 2019, p. 169–176.
 - **J-BHI 2019 (OLIVEIRA *et al.*, 2019):** Paulo H. Oliveira; Lucas C. Scabora; Mirela T. Cazzolato; Willian D. Oliveira; Rafael S. Paixão; Agma J. M. Traina; Caetano Traina-Jr. **Employing Domain Indexes to Efficiently Query Medical Data From Multiple Repositories.** Journal of Biomedical and Health Informatics, 23(6), 2019, p. 2220–2229.
 - **CMPB 2020 (CHINO *et al.*, 2020):** Daniel Y. T. Chino; Lucas C. Scabora; Mirela T. Cazzolato; Ana Elisa Serafim Jorge; Caetano Traina-Jr.; Agma J. M. Traina. **Segmenting skin ulcers and measuring the wound area using deep convolutional networks.** Journal of Computer Methods and Programs in Biomedicine, 191, 2020, p. 105376.
 - **CBMS 2020 (FLOREZ *et al.*, 2020):** Alexander Y. C. Florez; Lucas C. Scabora; Sihem Amer-Yahia; José Fernando Rodrigues-Jr. **Augmentation Techniques for Sequential Clinical Data to Improve Deep Learning Prediction Techniques.** In Proceedings of the 33rd International Symposium on Computer-Based Medical Systems, IEEE, 2020, p. 597–602.
 - **CBMS 2020 (CAZZOLATO *et al.*, 2020):** Mirela T. Cazzolato; Jonathan S. Ramos; Lucas S. Rodrigues; Lucas C. Scabora; Daniel Y. T. Chino; Ana Elisa Serafim Jorge; Paulo M. de A. Marques; Caetano Traina-Jr.; Agma J. M. Traina. **Semi-Automatic Ulcer Segmentation and Wound Area Measurement Supporting Telemedicine.** In Proceedings of the 33rd International Symposium on Computer-Based Medical Systems, IEEE, 2020, p. 356–361.

BIBLIOGRAPHY

ABERGER, C. R.; LAMB, A.; OLUKOTUN, K.; RÉ, C. Mind the Gap: Bridging multi-domain query workloads with EmptyHeaded. **Proceedings of the VLDB Endowment**, v. 10, n. 12, p. 1849–1852, 2017. Citations on pages [30](#) and [50](#).

ABERGER, C. R.; TU, S.; OLUKOTUN, K.; RÉ, C. EmptyHeaded: A relational engine for graph processing. In: **Proceedings of the SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2016. p. 431–446. Citations on pages [30](#) and [50](#).

AHMED, A.; THOMO, A. Computing source-to-target shortest paths for complex networks in RDBMS. **Journal of Computer and System Sciences**, v. 89, p. 114–129, 2017. Citation on page [30](#).

AL-AMIN, S. T.; ORDONEZ, C.; BELLATRECHE, L. Big data analytics: Exploring graphs with optimized SQL queries. In: **Proceedings of the Database and Expert Systems Applications - DEXA International Workshops, BDMICS, BIOKDD, and TIR**. [S.l.]: Springer, 2018. (Communications in Computer and Information Science, v. 903), p. 88–100. Citation on page [30](#).

ALBERT, R.; BARABÁSI, A.-L. Statistical mechanics of complex networks. **Reviews of Modern Physics**, American Physical Society, v. 74, p. 47–97, 2002. Citation on page [39](#).

BARABÁSI, A. L.; PÓSFAL, M. **Network science**. [S.l.]: Cambridge University Press, 2016. ISBN 9781107076266. Citations on pages [29](#), [39](#), [40](#), and [41](#).

BETTENCOURT, L. M. A.; LOBO, J.; STRUMSKY, D.; WEST, G. B. Urban Scaling and Its Deviations: Revealing the Structure of Wealth, Innovation and Crime across Cities. **PLoS ONE**, Public Library of Science (PLoS), v. 5, n. 11, p. e13541, nov 2010. Citation on page [58](#).

BOCCALETTI, S.; LATORA, V.; MORENO, Y.; CHAVEZ, M.; HWANG, D.-U. Complex networks: Structure and dynamics. **Physics Reports**, v. 424, n. 4, p. 175–308, 2006. Citation on page [42](#).

BOGOMOLOV, A.; LEPRI, B.; STAIANO, J.; OLIVER, N.; PIANESI, F.; PENTLAND, A. Once Upon a Crime. In: **Proceedings of the 16th International Conference on Multimodal Interaction - ICMI '14**. [S.l.]: ACM, 2014. v. 1, p. 427–434. ISBN 9781450328852. ISSN 0198-9715. Citation on page [58](#).

BORNEA, M. A.; DOLBY, J.; KEMENTSIETSIDIS, A.; SRINIVAS, K.; DANTRESSANGLE, P.; UDREA, O.; BHATTACHARJEE, B. Building an efficient RDF store over a relational database. In: **Proceedings of the SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2013. p. 121–132. Citation on page [46](#).

BRANDES, U. On variants of shortest-path betweenness centrality and their generic computation. **Social Networks**, v. 30, n. 2, p. 136–145, 2008. Citation on page [42](#).

BURZANSKA, M.; STENCEL, K.; WISNIEWSKI, P. Pushing predicates into recursive SQL common table expressions. In: **Advances in Databases and Information Systems, ADBIS**. [S.l.]: Springer, 2009. (Lecture Notes in Computer Science, v. 5739), p. 194–205. Citations on pages [48](#) and [49](#).

CABRERA, W.; ORDONEZ, C. Unified algorithm to solve several graph problems with relational queries. In: **Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)**. [S.l.: s.n.], 2016. Citation on page [48](#).

_____. Scalable parallel graph algorithms with matrix–vector multiplication evaluated with queries. **Distributed and Parallel Databases**, Springer, v. 35, n. 3–4, p. 335–362, 2017. Citations on pages [31](#) and [48](#).

CATTELL, R. Scalable SQL and NoSQL data stores. **SIGMOD Records**, ACM, v. 39, n. 4, p. 12–27, 2011. Citations on pages [29](#) and [43](#).

CAZZOLATO, M. T.; RAMOS, J. S.; RODRIGUES, L. S.; SCABORA, L. C.; CHINO, D. Y. T.; JORGE, A. E. S.; MARQUES, P. M. de A.; TRAINA-JR., C.; TRAINA, A. J. M. Semi-automatic ulcer segmentation and wound area measurement supporting telemedicine. In: **Proceedings of the 33rd International Symposium on Computer-Based Medical Systems - CBMS**. [S.l.]: IEEE, 2020. p. 356–361. Citation on page [106](#).

CAZZOLATO, M. T.; RODRIGUES, L. S.; SCABORA, L. C.; ZABOT, G. F.; VASCONCELOS, G. Q.; CHINO, D. Y. T.; JORGE, A. E. S.; CORDEIRO, R. L. F.; TRAINA-JR., C.; TRAINA, A. J. M. A dbms-based framework for content-based retrieval and analysis of skin ulcer images in medical practice. In: **Proceedings of the 34th Brazilian Symposium on Databases - SBBD**. [S.l.]: SBC, 2019. p. 109–120. Citation on page [105](#).

CAZZOLATO, M. T.; SCABORA, L. C.; COSTA, A. F.; NESSO-JR., M. R.; OLIVEIRA, L. F. M.; KASTER, D. S.; TRAINA-JR, C.; TRAINA, A. J. M. BREATH: heat maps assisting the detection of abnormal lung regions in CT scans. In: **Proceedings of the 30th International Symposium on Computer-Based Medical Systems - CBMS**. [S.l.]: IEEE, 2017. p. 248–253. Citation on page [104](#).

CAZZOLATO, M. T.; SCABORA, L. C.; NESSO-JR., M. R.; OLIVEIRA, L. F. M.; COSTA, A. F.; KASTER, D. S.; KOENIGKAM-SANTOS, M.; MARQUES, P. M. de A.; TRAINA-JR., C.; TRAINA, A. J. M. *dp-breath*: Heat maps and probabilistic classification assisting the analysis of abnormal lung regions. **Journal of Computer Methods and Programs in Biomedicine - CMBP**, v. 173, p. 27–34, 2019. Citation on page [105](#).

CHEN, R. Managing massive graphs in relational DBMS. In: **Proceedings of the International Conference on Big Data**. [S.l.]: IEEE Computer Society, 2013. p. 1–8. Citations on pages [30](#) and [46](#).

CHINO, D. Y. T.; SCABORA, L. C.; CAZZOLATO, M. T.; JORGE, A. E. S.; TRAINA-JR., C.; TRAINA, A. J. M. ICARUS: retrieving skin ulcer images through bag-of-signatures. In: **Proceedings of the 31st International Symposium on Computer-Based Medical Systems - CBMS**. [S.l.]: IEEE, 2018. p. 82–87. Citation on page [105](#).

_____. Segmenting skin ulcers and measuring the wound area using deep convolutional networks. **Journal of Computer Methods and Programs in Biomedicine - CMBP**, v. 191, p. 105376, 2020. Citation on page [106](#).

CHINO, D. Y. T.; SCABORA, L. C.; TRAINA-JR., C.; TRAINA, A. J. M. Boss: image retrieval using bag-of-superpixels signatures. In: **Proceedings of the 33rd SIGAPP Symposium on Applied Computing - SAC**. [S.l.]: ACM, 2018. p. 309–312. Citation on page [105](#).

CODD, E. F. A relational model of data for large shared data banks. **Communications of the ACM**, v. 13, n. 6, p. 377–387, 1970. Citation on page [29](#).

CORBELLINI, A.; MATEOS, C.; ZUNINO, A.; GODOY, D.; SCHIAFFINO, S. N. Persisting big-data: The NoSQL landscape. **Information Systems**, v. 63, p. 1–23, 2017. Citations on pages [29](#), [43](#), and [44](#).

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms, 3rd Edition**. [S.l.]: The MIT Press, 2009. ISBN 978-0-262-03384-8. Citations on pages [42](#) and [45](#).

CORTESI, A.; HALDER, R. Abstract interpretation of recursive queries. In: **Proceedings of the 9th International Conference on Distributed Computing and Internet Technology - ICDCIT**. [S.l.]: Springer Berlin Heidelberg, 2013. (Lecture Notes in Computer Science, v. 7753), p. 157–170. Citation on page [30](#).

DERYOL, R.; WILCOX, P.; LOGAN, M.; WOOLDREDGE, J. Crime Places in Context: An Illustration of the Multilevel Nature of Hot Spot Development. **Journal of Quantitative Criminology**, Springer Nature, v. 32, n. 2, p. 305–325, jan 2016. ISSN 0748-4518. Citation on page [58](#).

DOLMATOVA, O.; AUGSTEN, N.; BÖHLEN, M. H. A relational matrix algebra and its implementation in a column store. In: **Proceedings of the International Conference on Management of Data, SIGMOD**. [S.l.]: ACM, 2020. p. 2573–2587. Citation on page [31](#).

EISENBERG, A.; MELTON, J. Sql: 1999, formerly known as sql3. **SIGMOD Record**, ACM, v. 28, n. 1, p. 131—138, 1999. Citation on page [54](#).

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems**. [S.l.]: Pearson, 2016. (Always learning). ISBN 9781292097619. Citations on pages [29](#), [30](#), [35](#), [36](#), [38](#), [39](#), and [46](#).

FAN, J.; RAJ, A. G. S.; PATEL, J. M. The case against specialized graph analytics engines. In: **Proceedings of the 7th Biennial Conference on Innovative Data Systems Research - CIDR**. [S.l.]: Online Proceedings, 2015. Citations on pages [30](#), [31](#), [48](#), and [49](#).

FAN, W.; WU, Y.; XU, J. Functional dependencies for graphs. In: **Proceedings of the International Conference on Management of Data**. [S.l.]: Association for Computing Machinery, 2016. (SIGMOD '16), p. 1843–1857. ISBN 9781450335317. Citation on page [40](#).

FERREIRA, A.; RUBIANO, G.; MOJICA-NAVA, E. Urban Security Analysis in the City of Bogotá Using Complex Networks. In: SPRINGER. **International Conference on Complex Systems**. [S.l.], 2018. p. 424–438. Citation on page [58](#).

FITTERER, J.; NELSON, T.; NATHOO, F. Predictive crime mapping. **Police Practice and Research**, Informa UK Limited, v. 16, n. 2, p. 121–135, oct 2014. ISSN 1561-4263. Citation on page [58](#).

FLORATOS, S.; ZHANG, Y.; YUAN, Y.; LEE, R.; ZHANG, X. SQLoop: High performance iterative processing in data management. In: **Proceedings of the 38th International Conference on Distributed Computing Systems - ICDCS**. [S.l.]: IEEE Computer Society, 2018. p. 1039–1051. Citations on pages 31, 47, 48, and 50.

FLOREZ, A. Y. C.; SCABORA, L. C.; AMER-YAHIA, S.; JUNIOR, J. F. R. Augmentation techniques for sequential clinical data to improve deep learning prediction techniques. In: **Proceedings of the 33rd International Symposium on Computer-Based Medical Systems - CBMS**. [S.l.]: IEEE, 2020. p. 597–602. Citation on page 106.

GALBRUN, E.; PELECHRINIS, K.; TERZI, E. Urban navigation beyond shortest route: The case of safe paths. **Information Systems**, Elsevier BV, v. 57, p. 160–171, apr 2016. Citation on page 32.

GAO, J.; JIN, R.; ZHOU, J.; YU, J. X.; JIANG, X.; WANG, T. Relational approach for shortest path discovery over large graphs. **Proceedings of the VLDB Endowment**, v. 5, n. 4, p. 358–369, 2011. Citation on page 48.

GAO, J.; ZHOU, J.; YU, J. X.; WANG, T. Shortest path computing in relational DBMSs. **Transactions on Knowledge and Data Engineering - TKDE**, IEEE Educational Activities Department, v. 26, n. 4, p. 997–1011, 2014. Citations on pages 30 and 48.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database systems: the complete book**. [S.l.]: Pearson Education India, 2008. Citations on pages 29, 35, and 36.

GROLINGER, K.; HIGASHINO, W. A.; TIWARI, A.; CAPRETZ, M. A. Data management in cloud environments: NoSQL and NewSQL data stores. **Journal of Cloud Computing: Advances, Systems and Applications**, v. 2, n. 1, p. 22, 2013. Citations on pages 43 and 44.

GÜTING, R. H. GraphDB: modeling and querying graphs in databases. In: BOCCA, J. B.; JARKE, M.; ZANIOLO, C. (Ed.). **Proceedings of the 20th International Conference on Very Large Data Bases - VLDB**. [S.l.]: Morgan Kaufmann Publishers Inc., 1994. p. 297–308. Citation on page 30.

HAN, J.; KAMBER, M.; PEI, J. **Data Mining: Concepts and Techniques, 3rd edition**. Morgan Kaufmann, 2011. ISBN 978-0123814791. Available: <<http://hanj.cs.illinois.edu/bk3/>>. Citation on page 39.

HASSAN, M. S.; KUZNETSOVA, T.; JEONG, H. C.; AREF, W. G.; SADOOGHI, M. Empowering in-memory relational database engines with native graph processing. **CoRR**, abs/1709.06715, 2017. Available: <<http://arxiv.org/abs/1709.06715>>. Citations on pages 30, 44, and 51.

_____. Extending in-memory relational database engines with native graph support. In: **Proceedings of the 21st International Conference on Extending Database Technology – EDBT**. [S.l.]: OpenProceedings.org, 2018. p. 25–36. Citations on pages 30, 44, and 51.

_____. GRFusion: Graphs as first-class citizens in main-memory relational database systems. In: **Proceedings of the SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2018. p. 1789–1792. Citations on pages 30, 44, and 51.

HECHT, R.; JABLONSKI, S. NoSQL evaluation: A use case oriented survey. In: **International Conference on Cloud and Service Computing**. [S.l.]: IEEE Computer Society, 2011. p. 336–341. Citation on page 43.

HONG, J.; PARK, K.; HAN, Y.; RASEL, M. K.; VONVOU, D.; LEE, Y. Disk-based shortest path discovery using distance index over large dynamic graphs. **Information Sciences**, v. 382-383, p. 201–215, 2017. Citation on page 30.

JANKOV, D.; LUO, S.; YUAN, B.; CAI, Z.; ZOU, J.; JERMAINE, C.; GAO, Z. J. Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. **SIGMOD Rec.**, v. 49, n. 1, p. 43–50, 2020. Citation on page 31.

JINDAL, A.; MADDEN, S.; CASTELLANOS, M.; HSU, M. Graph analytics using vertica relational database. In: **Proceedings of the International Conference on Big Data**. [S.l.]: IEEE Computer Society, 2015. p. 1191–1200. Citations on pages 30, 31, 48, and 49.

KANG, C.; SOBOLEVSKY, S.; LIU, Y.; RATTI, C. Exploring human movements in singapore: a comparative analysis based on mobile phone and taxicab usages. In: **Proceedings of the 2nd ACM SIGKDD International Workshop on Urban Computing**. [S.l.: s.n.], 2013. Citation on page 58.

KARDES, H.; GUNES, M. H. Structural graph indexing for mining complex networks. In: **Proceedings of the 30th International Conference on Distributed Computing Systems Workshops - ICDCSW**. [S.l.]: IEEE, 2010. p. 99–104. Citation on page 42.

KIM, J. H.; CANDAN, K. S.; SAPINO, M. L. Pagerank revisited: On the relationship between node degrees and node significances in different applications. In: **Proceedings of the Workshops of the EDBT/ICDT Joint Conference**. [S.l.]: CEUR-WS.org, 2016. (CEUR Workshop Proceedings, v. 1558). Citation on page 42.

KIMBALL, R.; CASERTA, J. **The data warehouse ETL toolkit: practical techniques for extracting, cleaning, conforming, and delivering data**. USA: John Wiley & Sons, 2011. Citation on page 58.

KLEPPMANN, M. **Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems**. [S.l.]: "O'Reilly Media, Inc.", 2017. Citation on page 29.

KYROLA, A.; BLELLOCH, G. E.; GUESTIN, C. GraphChi: Large-scale graph computation on just a PC. In: **Proceedings of the 10th Conference on Operating Systems Design and Implementation - OSDI**. [S.l.]: USENIX Association, 2012. p. 31–46. Citation on page 47.

LASALLE, D.; KARYPIS, G. Multi-threaded modularity based graph clustering using the multilevel paradigm. **J. Parallel Distrib. Comput.**, Academic Press, Inc., Orlando, FL, USA, v. 76, n. C, p. 66–80, 2015. ISSN 0743-7315. Available: <<https://doi-org.ez67.periodicos.capes.gov.br/10.1016/j.jpdc.2014.09.012>>. Citation on page 59.

LEAVITT, N. Will NoSQL databases live up to their promise? **Computer**, IEEE Computer Society, v. 43, n. 2, p. 12–14, 2010. Citation on page 43.

LEVANDOSKI, J. J.; MOKBEL, M. F. RDF data-centric storage. In: **International Conference on Web Services - ICWS**. [S.l.]: IEEE Computer Society, 2009. p. 911–918. Citations on pages 44 and 45.

LIMA, D. M.; RODRIGUES-JR, J. F.; TRAINA, A. J.; PIRES, F. A.; GUTIERREZ, M. A. Transforming two decades of epr data to OMOP CDM for clinical research. In: **MEDINFO: Health and Wellbeing e-Networks for All - Proceedings of the 17th World Congress on**

Medical and Health Informatics. [S.l.]: IOS Press, 2019. v. 264, p. 233–237. Citation on page [67](#).

MA, H.; SHAO, B.; XIAO, Y.; CHEN, L. J.; WANG, H. G-SQL: fast query processing via graph exploration. **Proceedings of the VLDB Endowment**, v. 9, n. 12, p. 900–911, 2016. Citation on page [30](#).

MARUHASHI, K.; SHIGEZUMI, J.; YUGAMI, N.; FALOUTSOS, C. EigenSP: A more accurate shortest path distance estimation on large-scale networks. In: **Proceedings of the 12th International Conference on Data Mining Workshops - ICDM**. [S.l.]: IEEE Computer Society, 2012. p. 234–241. Citation on page [45](#).

MCCOLL, R. C.; EDIGER, D.; POOVEY, J.; CAMPBELL, D.; BADER, D. A. A performance evaluation of open source graph databases. In: . [S.l.]: Association for Computing Machinery, 2014. (PPAA '14), p. 11–18. ISBN 9781450326544. Citations on pages [30](#) and [41](#).

MILER, M.; MEDAK, D.; ODOBAŠIĆ, D. The shortest path algorithm performance comparison in graph and relational database on a transportation network. **Promet-Traffic&Transportation**, Fakultet prometnih znanosti Sveučilišta u Zagrebu, v. 26, n. 1, p. 75–82, 2014. Citation on page [50](#).

MOHAN, C. History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla. In: **Proceedings of the 16th International Conference on Extending Database Technology - EDBT**. [S.l.]: ACM, 2013. p. 11–16. Citation on page [43](#).

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.; RODRIGUES-JR., J. F.; TRAINA, A. J. M.; TRAINA-JR., C. RAFIKI: retrieval-based application for imaging and knowledge investigation. In: **Proceedings of the 31st International Symposium on Computer-Based Medical Systems - CBMS**. [S.l.]: IEEE, 2018. p. 71–76. Citation on page [105](#).

NEUMANN, T.; WEIKUM, G. The RDF-3X engine for scalable management of RDF data. **The VLDB Journal**, Springer-Verlag, v. 19, n. 1, p. 91–113, 2010. Citation on page [44](#).

NEWMAN, M. E. J.; WATTS, D. J. Renormalization group analysis of the small-world network model. **Physics Letters A**, Elsevier, v. 263, n. 4, p. 341–346, 1999. Citation on page [39](#).

NIETO-CHAUPIS, H. Identification of the Social Duality: Street Criminality and High Vehicle Traffic in Lima City by Using Artificial Intelligence Through the Fisher-Snedecor Statistics and Shannon's Entropy. In: **IEEE International Smart Cities Conference (ISC2)**. [S.l.: s.n.], 2018. p. 1–6. Citation on page [58](#).

_____. Shannon-entropy-based artificial intelligence applied to identify social anomalies in large latin american cities. In: **IEEE 39th Sarnoff Symposium**. [S.l.: s.n.], 2018. p. 1–4. Citation on page [58](#).

OLIVEIRA, P. H.; SCABORA, L. C.; CAZZOLATO, M. T.; OLIVEIRA, W. D.; TRAINA, A. J. M.; TRAINA-JR., C. Efficiently indexing multiple repositories of medical image databases. In: **Proceedings of the 30th International Symposium on Computer-Based Medical Systems - CBMS**. [S.l.]: IEEE, 2017. p. 286–291. Citation on page [104](#).

OLIVEIRA, P. H.; SCABORA, L. C.; CAZZOLATO, M. T.; BEDO, M. V. N.; TRAINA, A. J. M.; TRAINA-JR., C. MAMMOSET: An Enhanced Dataset of Mammograms. In: **Proceedings of the Satellite Events of the 32nd Brazilian Symposium on Databases**. [S.l.]: SBC, 2017. p. 256–266. Citation on page 104.

OLIVEIRA, P. H.; SCABORA, L. C.; CAZZOLATO, M. T.; OLIVEIRA, W. D.; PAIXÃO, R. S.; TRAINA, A. J. M.; TRAINA-JR., C. Employing domain indexes to efficiently query medical data from multiple repositories. **Journal of Biomedical and Health Informatics**, v. 23, n. 6, p. 2220–2229, 2019. Citation on page 106.

OLIVEIRA, P. H.; SCABORA, L. C.; TRAINA-JR., C.; KASTER, D. S. One index to dominate them all: domain indexes for improving queries across multiple tables. In: **Proceedings of the 32nd SIGAPP Symposium on Applied Computing - SAC**. [S.l.]: ACM, 2017. p. 900–905. Citation on page 104.

ORDONEZ, C. Optimizing recursive queries in SQL. In: **Proceedings of the SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2005. p. 834–839. Citations on pages 30, 48, and 49.

_____. Optimization of linear recursive queries in SQL. **IEEE Trans. Knowl. Data Eng.**, v. 22, n. 2, p. 264–277, 2010. Citations on pages 48 and 49.

ORDONEZ, C.; CABRERA, W.; GURRAM, A. Comparing columnar, row and array dbms to process recursive queries on graphs. **Information Systems**, v. 63, p. 66–79, 2017. Citations on pages 31 and 47.

ORDONEZ, C.; GURRAM, A.; RAI, N. Recursive query evaluation in a column DBMS to analyze large graphs. In: **Proceedings of the 17th International Workshop on Data Warehousing and OLAP - DOLAP**. [S.l.]: ACM, 2014. p. 71–80. Citations on pages 48 and 49.

PAPALEXAKIS, E.; HOOI, B.; PELECHRINIS, K.; FALOUTSOS, C. Power-hop: A pervasive observation for real complex networks. **PLOS ONE**, Public Library of Science, v. 11, n. 3, p. 1–11, 03 2016. Available: <<https://doi.org/10.1371/journal.pone.0151027>>. Citation on page 31.

PARADIES, M.; LEHNER, W.; BORNHÖVD, C. GRAPHITE: an extensible graph traversal framework for relational database management systems. In: **Proceedings of the 27th International Conference on Scientific and Statistical Database Management - SSDBM**. [S.l.]: ACM, 2015. p. 29:1–29:12. Citation on page 51.

PASSING, L.; THEN, M.; HUBIG, N.; LANG, H.; SCHREIER, M.; GÜNNEMANN, S.; KEMPER, A.; NEUMANN, T. SQL- and operator-centric data analytics in relational main-memory databases. In: **Proceedings of the 20th International Conference on Extending Database Technology, EDBT**. [S.l.]: OpenProceedings.org, 2017. p. 84–95. Citation on page 31.

PEREZ, Y.; SOSIC, R.; BANERJEE, A.; PUTTAGUNTA, R.; RAISON, M.; SHAH, P.; LESKOVEC, J. Ringo: Interactive graph analytics on big-memory machines. In: **Proceedings of the SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2015. p. 1105–1110. Citation on page 50.

PORTA, S.; STRANO, E.; IACOVIELLO, V.; MESSORA, R.; LATORA, V.; CARDILLO, A.; WANG, F.; SCELLATO, S. Street centrality and densities of retail and services in bologna, italy.

Environment and Planning B Planning and Design, v. 36, p. 1–15, 2009. Citation on page 58.

PRZYMUS, P.; BONIEWICZ, A.; BURZANSKA, M.; STENCEL, K. Recursive query facilities in relational databases: A survey. In: **International Conferences on Database Theory and Application, Bio-Science and Bio-Technology - DTA and BSBT**. [S.l.]: Springer Berlin Heidelberg, 2010. (Communications in Computer and Information Science, v. 118), p. 89–99. Citations on pages 30 and 47.

ROTMENSCH, M.; HALPERN, Y.; TLIMAT, A.; HORNG, S.; SONTAG, D. Learning a health knowledge graph from electronic medical records. **Scientific Reports**, v. 7, n. 1, p. 5994, 2017. Citation on page 31.

SAHU, S.; MHEDHBI, A.; SALIHOGLU, S.; LIN, J.; ÖZSU, M. T. The ubiquity of large graphs and surprising challenges of graph processing. **Proceedings of the VLDB Endowment**, v. 11, n. 4, p. 420–431, 2017. ISSN 2150-8097. Citations on pages 30, 35, and 41.

SAHU, S.; MHEDHBI, A.; SALIHOGLU, S.; LIN, J.; ÖZSU, M. T. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. **The VLDB Journal**, Springer Science and Business Media LLC, v. 29, n. 2-3, p. 595–618, 2019. Citations on pages 30, 35, and 41.

SAKR, S.; ELNIKETY, S.; HE, Y. G-SPARQL: a hybrid engine for querying large attributed graphs. In: **Proceedings of the 21st International Conference on Information and Knowledge Management - CIKM**. [S.l.]: ACM, 2012. p. 335–344. Citation on page 51.

_____. Hybrid query execution engine for large attributed graphs. **Information Systems**, v. 41, p. 45–73, 2014. Citation on page 51.

SCABORA, L. C.; OLIVEIRA, P. H.; KASTER, D. S.; TRAINA, A. J. M.; TRAINA-JR., C. Relational graph data management on the edge: Grouping vertices' neighborhood with Edge- k . In: **Proceedings of the 32nd Brazilian Symposium on Databases - SBBD**. [S.l.]: SBC, 2017. p. 124–135. Citations on pages 33, 75, 76, 82, and 103.

SCABORA, L. C.; OLIVEIRA, P. H.; SPADON, G.; KASTER, D. S.; RODRIGUES-JR., J. F.; TRAINA, A. J. M.; TRAINA-JR., C. Cutting-edge relational graph data management with Edge- k : from one to multiple edges in the same row. **Journal of Information and Data Management - JIDM**, SBC, 2018. Citations on pages 33, 75, 82, 83, 85, 86, 89, and 103.

SCABORA, L. C.; SPADON, G.; CAZZOLATO, M. T.; KASTER, D. S.; TRAINA, A. J. M.; RODRIGUES-JR., J. F.; TRAINA-JR., C. SHARq: Sharing recursive queries in relational databases. In: **Proceedings of the 36th SIGAPP Symposium on Applied Computing - SAC, Virtual Event**. [S.l.]: ACM, 2021. p. 336–339. Citations on pages 33, 62, and 104.

SCABORA, L. C.; SPADON, G.; OLIVEIRA, P. H.; RODRIGUES-JR., J. F.; TRAINA-JR., C. Enhancing recursive graph querying on RDBMS with data clustering approaches. In: **Proceedings of the 35th SIGAPP Symposium on Applied Computing - SAC**. [S.l.]: ACM, 2020. p. 404–411. Citations on pages 34, 90, 91, 94, 95, 96, 98, and 104.

SCABORA, L. C.; SPADON, G.; RODRIGUES, L. S.; CAZZOLATO, M. T.; ARAUJO, M. V. S.; SOUSA, E. P. M. de; TRAINA, A. J. M.; RODRIGUES-JR., J. F.; TRAINA-JR., C. G-FranC: a dataset of criminal activities mapped as a complex network in a relational DBMS.

In: **Dataset Show Case of the 34th Brazilian Symposium on Databases**. [S.l.]: SBC, 2019. p. 366–376. Citations on pages [57](#), [59](#), and [103](#).

SCHUH, S.; CHEN, X.; DITTRICH, J. An experimental comparison of thirteen relational equi-joins in main memory. In: **Proceedings of the SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2016. p. 1961–1976. Citations on pages [30](#) and [36](#).

SEO, K.; AHN, J.; IM, D. Optimization of shortest-path search on rdbms-based graphs. **ISPRS Int. J. Geo Inf.**, v. 8, n. 12, p. 550, 2019. Citation on page [30](#).

SILBERSCHATZ, A.; KORTH, H.; SUDARSHAN, S. **Database Systems Concepts**. 5. ed. [S.l.]: McGraw-Hill, Inc., 2005. Citations on pages [29](#), [35](#), and [36](#).

SIMMEN, D. E.; SCHNAITTER, K.; DAVIS, J.; HE, Y.; LOHARIWALA, S.; MYSORE, A.; SHENOI, V.; TAN, M.; XIAO, Y. Large-scale graph analytics in aster 6: Bringing context to big data discovery. **Proceedings of the VLDB Endowment**, v. 7, n. 13, p. 1405–1416, 2014. Citation on page [31](#).

SPADON, G.; SCABORA, L. C.; ARAUJO, M. V. S.; OLIVEIRA, P. H.; MACHADO, B. B.; SOUSA, E. P. M. de; TRAINA-JR., C.; JR., J. F. R. Complex network tools to understand the behavior of criminality in urban areas. In: **Information Technology - New Generations**. [S.l.]: Springer International Publishing, 2017. p. 493–500. Citations on pages [58](#) and [104](#).

SPADON, G.; SCABORA, L. C.; JR., M. R. N.; TRAINA-JR., C.; RODRIGUES-JR., J. F. Caracterização topológica de redes viárias por meio da análise de vetores de características e técnicas de agrupamento. In: **Proceedings of the 33rd Brazilian Symposium on Databases - SBBD**. [S.l.]: SBC, 2018. p. 157–168. Citation on page [105](#).

SPADON, G.; SCABORA, L. C.; OLIVEIRA, P. H.; ARAUJO, M. V. S.; MACHADO, B. B.; SOUSA, E. P. M. de; TRAINA-JR., C.; JR., J. F. R. Behavioral characterization of criminality spread in cities. **Procedia Computer Science**, v. 108, p. 2537–2541, 2017. Citations on pages [58](#) and [104](#).

STONEBRAKER, M. New opportunities for New SQL. **Communications of the ACM**, ACM, v. 55, n. 11, p. 10–11, 2012. Citation on page [43](#).

STONEBRAKER, M.; WEISBERG, A. The VoltDB main memory DBMS. **IEEE Data Engineering Bulletin**, v. 36, n. 2, p. 21–27, 2013. Citation on page [44](#).

SUN, W.; FOKOUE, A.; SRINIVAS, K.; KEMENTSIETSIDIS, A.; HU, G.; XIE, G. T. SQL-Graph: An efficient relational-based property graph store. In: **Proceedings of the SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2015. p. 1887–1901. Citations on pages [30](#) and [46](#).

TANG, J.; ZHANG, J.; YAO, L.; LI, J.; ZHANG, L.; SU, Z. Arnetminer: Extraction and mining of academic social networks. In: **KDD'08**. [S.l.: s.n.], 2008. p. 990–998. Citation on page [68](#).

TIAN, Y. *et al.* IBM db2 graph: Supporting synergistic and retrofittable graph queries inside IBM db2. In: **Proceedings of the 2020 International Conference on Management of Data – SIGMOD**. [S.l.]: ACM, 2020. p. 345–359. Citations on pages [30](#) and [51](#).

TIAN, Y.; TONG, S.; PIRAHESH, M. H.; SUN, W.; XU, E. L.; ZHAO, W. Synergistic graph and SQL analytics inside IBM db2. **Proceedings of the VLDB Endowment**, v. 12, n. 12, p. 1782–1785, 2019. Citations on pages [30](#) and [51](#).

TSOURAKAKIS, C. E.; DRINEAS, P.; MICHELAKIS, E.; KOUTIS, I.; FALOUTSOS, C. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. **Social Network Analysis and Mining**, v. 1, n. 2, p. 75–81, 2011. Citation on page 42.

VALDURIEZ, P. Join indices. **ACM Transactions on Database Systems**, ACM, v. 12, n. 2, p. 218–246, 1987. Citation on page 37.

VICKNAIR, C.; MACIAS, M.; ZHAO, Z.; NAN, X.; CHEN, Y.; WILKINS, D. A comparison of a graph database and a relational database: a data provenance perspective. In: **Proceedings of the 48th Annual Southeast Regional Conference**. [S.l.]: ACM, 2010. (ACM SE '10), p. 42:1–42:6. Citations on pages 29, 30, and 43.

VUKOTIC, A.; WATT, N.; ABEDRABBO, T.; FOX, D.; PARTNER, J. **Neo4j in Action**. [S.l.]: Manning Publications, 2014. ISBN 9781617290763. Citations on pages 30 and 43.

WYLIE, C. R. **Introduction to Projective Geometry**. [S.l.]: Courier Corporation, 2011. Citation on page 59.

XIROGIANNOPOULOS, K.; DESHPANDE, A. Extracting and analyzing hidden graphs from relational databases. In: **Proceedings of the SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2017. p. 897–912. Citations on pages 30 and 50.

XIROGIANNOPOULOS, K.; KHURANA, U.; DESHPANDE, A. GraphGen: Exploring interesting graphs in relational data. **Proceedings of the VLDB Endowment**, v. 8, n. 12, p. 2032–2035, 2015. Citations on pages 30 and 50.

XIROGIANNOPOULOS, K.; SRINIVAS, V.; DESHPANDE, A. GraphGen: Adaptive graph processing using relational databases. In: **Proceedings of the 5th International Workshop on Graph Data-management Experiences & Systems - GRADES**. [S.l.]: ACM, 2017. p. 9:1–9:7. Citations on pages 30 and 50.

ZABOT, G. F.; CAZZOLATO, M. T.; SCABORA, L. C.; TRAINA, A. J. M.; TRAINA-JR., C. Efficient indexing of multiple metric spaces with spectra. In: **International Symposium on Multimedia - ISM**. [S.l.]: IEEE, 2019. p. 169–176. Citation on page 106.

ZABOT, G. F.; CAZZOLATO, M. T.; SCABORA, L. C.; FAIÇAL, B. S.; TRAINA, A. J. M.; TRAINA-JR., C. UCORM: indexing uncorrelated metric spaces for concise content-based retrieval of medical images. In: **Proceedings of the 31st International Symposium on Computer-Based Medical Systems - CBMS**. [S.l.]: IEEE, 2019. p. 306–311. Citation on page 105.

ZHAO, K.; SU, J.; YU, J. X.; ZHANG, H. Sql-g: Efficient graph analytics by sql. **IEEE Transactions on Knowledge and Data Engineering**, p. 1–1, 2019. Citation on page 30.

ZHAO, K.; YU, J. X. All-in-One: Graph processing in RDBMSs revisited. In: **Proceedings of the SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2017. p. 1165–1180. Citations on pages 30, 31, 48, and 49.