Abou Bekr Belkaid University
Tlemcen Algeria
جامعة أبي بكر بلقايد
تلمسان الجزائر
PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
الجمهورية الجزائرية الديمقراطية الشعبية
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
وزارة التعليم العالي و البحث العلمي
FACULTY OF SCIENCES
DEPARTMENT OF COMPUTER SCIENCES

A Thesis

Presented for obtaining the degree of:

DOCTORATE

In: Computer Sciences

Specialty: Information systems engineering and decision support

By:

Abdelkrim BOUDAOUD

Theme

Towards An Expressive Model For Safe and Efficient Manipulation of Graph Databases

Thesis defended on May, 2026 at Tlemcen in front of the jury composed of:

Prof. Abdelkrim BENAMAR	Full Professor	University of Tlemcen	President
Dr. Houari MAHFOUD	Associate Professor	University of Tlemcen	Supervisor
Prof. Azzedine CHIKH	Full Professor	University of Tlemcen	Co-Supervisor
Prof. Nabil KESKES	Full Professor	ESI - Sidi Bel Abbès	Examiner
Dr. Abdelhamid MALKI	Associate Professor	ESI - Sidi Bel Abbès	Examiner
Dr. Yassamine SELADJI	Associate Professor	University of Tlemcen	Examiner

Academic Year: 2024-2025

Dedication

I proudly dedicate this thesis to:

- To Allah, the source of all wisdom and knowledge, for blessing me with the ability to learn, grow, and persevere in this journey.
- To my beloved family whose unwavering support has been my anchor through every challenge.
- To my teachers, mentors, and colleagues, whose guidance and shared knowledge have shaped my academic path.
- To those who inspired me to persist, even in moments of doubt or difficulty your encouragement kept me moving forward.

– Abdelkrim Boudaoud

Acknowledgements

This thesis represents the culmination of many years of intense work and the realization of a deeply cherished dream. I am profoundly proud to have reached the end of this intellectual marathon and am deeply grateful to be able to celebrate this achievement with my loved ones.

My deepest and most sincere gratitude goes to my thesis supervisor, Dr. Houari Mahfoud. His profound expertise in the subject and his intellectual agility were instrumental in guiding me at critical junctures. His kindness, unwavering encouragement, constant availability, and reassurance during moments of doubt were invaluable to this journey.

I am also immensely grateful to Prof. Chikh Azzedine, whose scientific rigor and meticulous advice pushed me to elevate the quality of my work to a higher standard.

I wish to extend my sincere thanks to the members of the jury for their rigorous evaluation of this work. Their expertise and careful reading have greatly enriched it. I would like to express my particular gratitude to Prof. Benamar Abdlekrim, President of the jury and examiner, as well as to Prof. Keskes Nabil, Dr. Malki Abdelhamid, and Dr. Seladji Yassamine, reviewers of the thesis, for the time they dedicated to attentively assessing my work and for their insightful and constructive comments.

My gratitude also goes to Prof. Ladjel Bellatreche for his warm welcome and invaluable assistance during my internship in France.

A special thanks to all my research colleagues with whom I had the pleasure of working throughout this adventure. Your stimulating discussions, camaraderie, and mutual support were a constant source of inspiration. Together, we overcame challenges

and celebrated successes, creating unforgettable memories.

To my friends, Soulimane Kamni, Sidi Mohammed Kaddour, Zakaria Madi, Adil Achraf Bereksi Reguig, Mustapha Salim Ghitri, Dahmani Sidi Mohammed, Ghellai Badreddine and all the others, who have supported me with love and understanding throughout this demanding academic journey, I thank you from the bottom of my heart. Your presence, your encouragement, and the relaxing moments we shared were essential in maintaining my balance and motivation.

Finally, I wish to dedicate this thesis to my family. Your unconditional support, love, and faith in me have been the driving force that spurred me to persevere, even through the most challenging times. This achievement is as much yours as it is mine. I am profoundly grateful to each of you, and this thesis bears the imprint of your contributions and unwavering support. Thank you from the bottom of my heart for being a part of this academic journey.

– *Abdelkrim Boudaoud*

Abstract

During the past decade, we have witnessed a wide speed use of graph databases for modeling and analyzing highly interconnected data, the relational model quickly reaches its limit when it comes to query and analyze such data. Despite that, the relational model still dominates in the Web. The first motivation of this thesis is to bridge this gap by presenting a framework for mapping and efficiently querying relational data within a graph paradigm, with contributions spanning theoretical foundations, scalability solutions, and practical implementations. First, we establish a rigorous formal foundation by introducing a complete mapping process that maps any relational database, along with its schema, data, constraints, and operations, to an equivalent graph database. This process is the first to holistically guarantee critical mapping properties: *information* and *semantic preservation* to prevent loss of meaning; *query* and *update preservation* through algorithms that automatically convert SQL (relational query language) to Cypher (graph query language); and *monotonicity* for efficient incremental updates. Next, we synthesize these foundations into **R2G-ETL**, an end-to-end software tool that automates the complete mapping process from relational systems to graph systems. As the first tool to enforce all formal preservation properties, it provides a validated and practical pipeline for real-world adoption. The second contribution of this thesis deals with the computational complexity of graph pattern matching on large-scale data. As querying of graph data is based on the subgraph

isomorphism, an NP-Complete problem, querying a large graph data may be time-consuming even with the more efficient systems like Neo4j. One of the possible solutions is to distribute the large graph data into separated machines, execute the query in each machine, collect, and merge the results. To our knowledge, no work has proposed a complete formalization of this problem. We then formalize a distributed querying framework for Cypher, the most widely used graph query language. This framework is centered on three algorithms, the first one handles user query fragmentation for distributed execution; the second one focuses on assembling the partial results returned from each fragment; while The core algorithm coordinates all these operations between the coordinator and the fragments, managing them in a parallel, multi-threaded manner. The distributed framework can integrate seamlessly within any graph database system such as Neo4j, overcoming the limitations of previous approaches.

keywords: Data mapping, Property graph, Graph Pattern Matching, Distributed Graph Pattern Matching, Cypher, Neo4j.

Résumé

Au cours de la dernière décennie, les bases de données orientées graphes se sont largement imposées pour modéliser et analyser des données fortement interconnectées, là où le modèle relationnel atteint rapidement ses limites. Toutefois, ce dernier demeure dominant sur le Web. Cette thèse vise à combler ce fossé en proposant un cadre permettant de transformer et d'interroger efficacement des données relationnelles selon un paradigme graphe, en apportant des contributions à la fois théoriques, techniques et pratiques. Nous introduisons d'abord un processus formel de transformation garantissant la préservation de l'information, de la sémantique, des requêtes et des mises à jour, ainsi que la monotonie des opérations. Ce processus est concrétisé dans R2G-ETL, un outil logiciel assurant une migration automatisée et fiable de bases relationnelles vers des systèmes graphe, en particulier via la conversion de SQL vers Cypher. La seconde contribution porte sur l'éfficacité du requêtage de graphes à grande échelle, reposant sur l'isomorphisme de sous-graphes, un problème NP-complet. Pour y faire face, nous proposons un cadre de requêtage distribué pour Cypher, structuré autour de trois algorithmes : (i) la fragmentation de la requête pour exécution distribuée, (ii) l'assemblage des résultats partiels, et (iii) un algorithme central orchestrant ces opérations entre un coordinateur et les fragments de manière parallèle et multi-threadée. Cette approche améliore considérablement l'efficacité du requêtage dans des systèmes comme Neo4j, tout en posant des fondations théoriques solides.

mots-clés: Mapping de données, graph de propriétés, interrogation des graphes de données, graphes de données distribués, Cypher, Neo4j.

مُلخّص

شهد العقد الأخير انتشارًا واسعًا لقواعد البيانات البيانية لاستخدامها في نمذجة وتحليل البيانات ذات الترابط العالي، حيث يصل النموذج العلاقي إلى حدوده بسرعة عند التعامل مع هذا النوع من البيانات. ومع ذلك، لا يزال النموذج العلاقي هو النموذج السائد على الويب. تهدف هذه الأطروحة إلى سد هذه الفجوة من خلال تقديم إطار يسمح بتحويل واستعلام البيانات العلاقية بكفاءة ضمن نموذج بياني، مع تقديم مساهمات تغطي الأسس النظرية، والحلول التقنية للتوسّع، والتطبيقات العملية.

نقترح أولًا عملية تحويل رسمية تضمن حفظ المعلومات، والمعنى الدلالي، واستمرارية الاستعلامات والتحديثات، بالإضافة إلى خاصية الرتابة التي تتيح دعم التحديثات التزايدية بكفاءة. وقد تم تجسيد هذا النهج ضمن أداة برمجية شاملة تدعى $R2G-ETL$ ، تقوم بأتمتة عملية التحويل من قواعد البيانات العلاقية إلى البيانية، بما في ذلك التحويل من لغة SQL إلى $Cypher$.

تتمثل المساهمة الثانية في معالجة تعقيد استعلام الأنماط البيانية ضمن بيانات ضخمة، وهو ما يرتبط بمشكلة تماثل تحت الرسوم البيانية، وهي مسألة تنتمي لفئة $NP-Complete$. ولمعالجة ذلك، نقترح إطارًا موزّعًا لاستعلامات $Cypher$ يعتمد على ثلاثة خوارزميات أساسية: (١) تجزئة الاستعلام لتنفيذه بشكل موزّع، (٢) تجميع النتائج الجزئية، و(٣) خوارزمية تنسيق مركزية تنظم هذه العمليات بين المنسّق والأجزاء المختلفة بطريقة متوازية ومتعددة الخيوط $(multi-threaded)$. يقدم هذا الإطار حلًا فعالًا وقابلًا للتكيّف مع أنظمة مثل $Neo4j$ ، متجاوزًا قيود الأساليب السابقة.

الكلمات المفتاحيّة: تعيين البيانات ، الرسم البياني ذو الخصائص

List of Publications

Journal Publications

1) Boudaoud, A., Mahfoud, H., Chikh, A. From relational model to property graph: mapping data, schema, constraints and queries. Prog Artif Intell (2025). **(IF=2.4)**

https://doi.org/10.1007/s13748-025-00373-0

International Conference Communications

1) Boudaoud, A., Mahfoud, H., Chikh, A. (2023). Towards a Complete Direct Mapping from Relational Databases to Property Graphs. In: Fournier-Viger, P., Hassan, A., Bellatreche, L. Model and Data Engineering. MEDI 2022. Lecture Notes in Computer Science, vol 13761. Springer, Cham, doi:10.1007/978-3-031-21595-7_16 **(3rd Place Best Paper Award)**

2) A. Boudaoud and H. Mahfoud, "A Framework for Querying Distributed Graph Data Using Cypher," 2025 International Conference on Intelligent Computer Systems, Data Science and Applications (IC2SDA), Blida, Algeria, 2025, pp. 1-8, doi: 10.1109/IC2SDA68097.2025.11331565

National Conference Communications

1) Boudaoud, A., Mahfoud, H. (2025). R2G-ETL : A Tool for Mapping any Relational database to Graph database. In: First National Conference on Innovation in Data Engineering and AI Science. IDEAS 2025.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

BSP Boundary Subpattern

CM Complete Mapping

DM Direct Mapping

GPM Graph Pattern Matching

IM Instance Mapping

IP Information Preservation

MP Monotonicity Preservation

nGQL NebulaGraph Query Language

PGQL Property Graph Query Language

PG Property Graph

QP Query Preservation

SHACL Shapes Constraint Language

SM Schema Mapping

SP Semantic Preservation

SQL Structured Query Language

UP Update Preservation

BFS Breadth-First Search

BS Bounded Simulation

CRs Candidate Regions

DAG Directed Acyclic Graph

DFS Depth-First Search

DS Dual Simulation

EC Embedding Cluster

GDs Graph Databases

GQL Graph Query Language

GS Graph Simulation

GSS Graph Simulation Semantics

RDF Resource Description Framework

RDFS Resource Description Framework Schema

SS Strong simulation

WWW World Wide Web

Introduction

1 General Context

The World Wide Web (WWW), conceptualized by Tim Berners-Lee at CERN (community of scientists from various institutes and universities) in 1989[1], fundamentally transformed global information sharing. Following its public release in 1993, the Web experienced exponential growth, especially with the raise of social networks, leading to an unprecedented explosion of data during the last decades. Data become very large (e.g., Facebook contains billions of users creating billions of information), complex, multi-models, and highly interconnected. This deluge of heterogeneous information quickly overwhelmed classical data storage and retrieval methods, spurring significant research into new database management systems [1].

A key response to this challenge was the vision of the Semantic Web, which aimed to make data not just human-readable but machine-understandable. Central to this vision is the Resource Description Framework (RDF) model [2], a W3C-standardized framework that structures data as subject-predicate-object triples. RDF, along with formal ontologies, provides a powerful mechanism for representing knowledge and meaning, enabling data integration and sophisticated reasoning across diverse domains.

Concurrently, the raise of social networks, e-commerce systems, and

[1] https://home.cern/science/computing/birth-web

recommendation engines highlighted a different but related aspect of data: the paramount importance of connections between entities. These platforms are inherently graph-like, where the value lies not only in the data entities but crucially in the complex web of interactions (relationships) between them. This underscored the need for data models that could efficiently represent these intricate networks and analyze them to extract hidden insights.

This need led to the emergence of the Property Graph (PG) model [3] as a highly effective alternative to RDF for modeling such interconnected data. While RDF excels at formal semantics, its triple-based structure has a key limitation for real-world network implementation: the inherent inability to support properties on relationships. Workarounds like reification introduce complexity and can impair performance. In contrast, the Property Graph model natively supports key-value properties on both nodes and edges, allowing for a more direct and intuitive mapping of real-world scenarios. This flexibility, combined with features like *index-free adjacency* for rapid traversal, has made the Property Graph a leading model for managing social, financial, and other network-based data.

Graph databases utilize a graph data model comprising nodes (entities), edges (relationships), and properties (attributes) to represent, store, and query complex, interconnected data networks [4]. As a category of NoSQL database, they are designed to overcome the limitations of relational models, especially when relationships between entities are a primary focus. Their intuitive model makes them indispensable for modern applications like recommendation systems [5], fraud detection [6], and knowledge graphs [7].

The *Property Graph* model stands out for its flexibility and schema-less nature, making it particularly well-suited for constructing rich and evolving

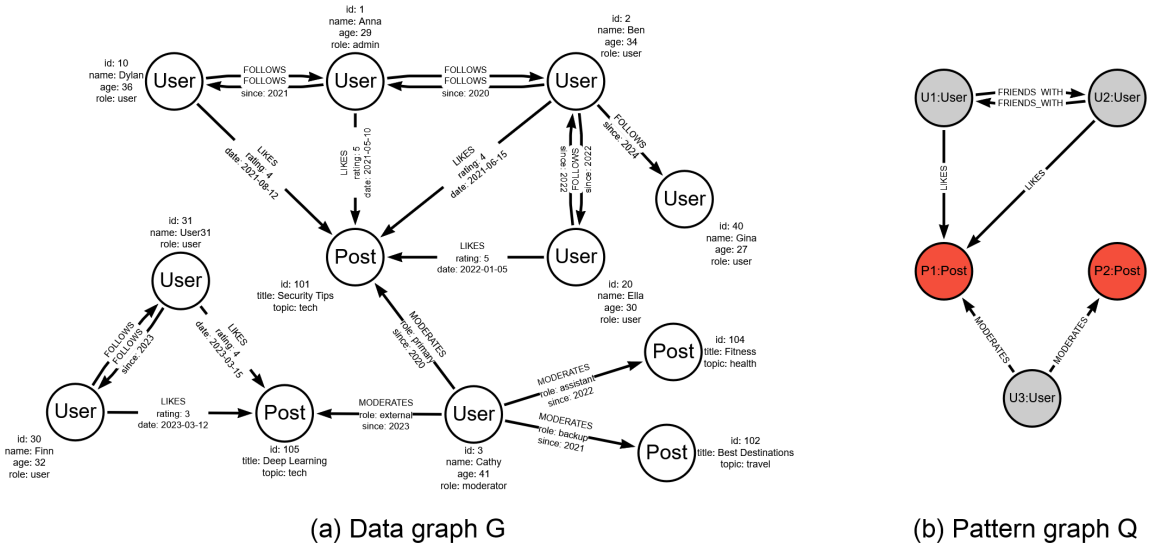


Figure 1: Example of a data graph and a pattern graph

knowledge graphs that can easily accommodate heterogeneous and dynamic data. Many query languages have been proposed for PGs, such as: *GQL* [8], *PGQL* [9] and *Cypher* [10]. The emergence of *Cypher* as an expressive and powerful language for PGs has led to the raise of a standardized graph query language: ISO-Graph Query Language (GQL) [11].

The choice between Property Graphs and RDF triplestores thus represents a fundamental trade-off between pragmatic flexibility and formal semantic rigor, with the Property Graph model often being favored for its agility in modeling complex networks. In addition, many works have studied mapping from RDFs to PGs, showing the importance of PGs.

A concrete example of a property graph can be seen in Figure 1(a) which illustrates a social network. This graph includes Users (e.g., Anna, Ben) and Posts, connected by relationships such as FOLLOWS (with properties like 'since'), LIKES, and MODERATES. This structure effectively captures complex social dynamics and interactions. Figure 1(b) is a graph representing a specific query, which looks for a team of 3 Users with some posts and interactions

Table 1: Most Popular RDF and Property Graph Database Systems

System	Graph Model	Storage Model	Persistence	Query Language	Mapping Tools	Distributed
RDF Graph Systems						
Virtuoso [12]	RDF	Hybrid (Native/Relational)	On-disk	SPARQL 1.1	ETL, R2RML	Yes
Apache Jena [13]	RDF	Hybrid (Native/Relational)	On-disk/In-memory	SPARQL 1.1	Jena API, RML	No
GraphDB [14]	RDF	Native	On-disk	SPARQL 1.1	OntoRefine, RDF4J	Enterprise
Blazegraph [15]	RDF	Native	On-disk/In-memory	SPARQL 1.1	Bulk Load API	Yes
Stardog [16]	RDF	Native	On-disk	SPARQL 1.1+	Virtual Graphs	Yes
Property Graph Systems						
Neo4j [17]	PG	Native	On-disk/In-memory	Cypher	Neo4j ETL	Enterprise
Memgraph [18]	PG	Native	In-memory	Cypher	Data Loaders	No
TigerGraph [19]	PG	Native	On-disk/In-memory	GSQL	REST Loaders	Yes
ArangoDB [20]	PG	Native	On-disk/In-memory	AQL	Arangoimport	Yes
Amazon Neptune [21]	PG/RDF	Native	On-disk	Gremlin/SPARQL	Bulk Loader	Yes

between them. By executing Q over G, each team in G that has all the requirements of Q is returned as query result.

Modern graph database systems can be broadly categorized into two main paradigms: RDF-based systems (triple stores) and PG systems. Table 1 provides the 5 most popular systems from each category based on key technical aspects.

For each system, Table 1 shows key technical aspects such as the graph model, storage model, persistence method, query language, available mapping tools, and whether the system supports distributed architecture. This allows for a quick, side-by-side comparison of the fundamental characteristics of these representative graph databases.

We note that only Neo4j and Memgraph support Cypher, which is currently the only widely adopted implementation of a graph query language. As a result, these two systems are among the most widely used. However, Neo4j outperforms Memgraph in several aspects, including built-in functions and APIs, data analytics capabilities, and the size of its user community. Notice also that both systems do not support distributed querying. Recently, Neo4j introduced *Infinigraph* as a distributed graph system. However, no formal specification or complete documentation has yet been offered.

2 Contributions

This thesis tackles fundamental challenges at the intersection of database modeling and efficient querying. While graph databases offer an intuitive model for interconnected data, their adoption is hampered by two primary obstacles: the inertia of existing relational data and the complexity of query evaluation on a single machine. Our work addresses these challenges through two key contributions:

- A formal and comprehensive mapping from the relational model to the property graph model.
- A novel framework for distributed evaluation of graph queries.

2.1 Motivations

The relational model has been the cornerstone of data management for decades, resulting in a vast ecosystem of legacy systems and expertise. Conversely, property graph databases have emerged as the superior paradigm for managing highly connected data, powering applications in social networks, recommendation engines, and knowledge graphs. However, a significant gap exists between existing relational databases and graphs creating a barrier to adoption.

The first motivation for this thesis is the need for a **mapping** from relational tuples to graphs. Existing mapping approaches are often ad-hoc, focusing solely on data translation while neglecting critical aspects like schema integrity, constraints (e.g., primary keys, foreign keys), and the semantic equivalence of queries. This leads to a loss of data integrity guarantees and unpredictable query results post-migration. A formal mapping is essential to ensure that the robust foundations of the relational model are not abandoned, but rather transformed

and preserved within the graph model, leveraging then the maturity of relational systems and the flexibility of graph systems.

The second motivation stems from the **inherent scalability requirements** of modern graph databases. As graph datasets grow beyond the capacity of a single machine, distributing the data becomes necessary. However, distributing graph queries introduces significant challenges in query optimization, data consistency, and fault tolerance. While distributed relational query processing is well-understood, graph queries, with their recursive and exploratory nature (e.g., variable-length path traversals), require novel distributed evaluation strategies. The gap between theoretical distributed algorithms and their practical integration within commercial graph systems like Neo4j is pronounced. This thesis aims to bridge this gap by providing a formal framework that ensures efficient and consistent query execution over partitioned graph data.

2.2 Contribution 1: Mapping from relations to graphs

Most enterprise data remains stored in relational databases. While simplistic mappings that convert tables to nodes and foreign keys to edges exist, they are insufficient for a production-ready migration. These naive approaches fail to capture the richness of the relational schema, leading to an impoverished graph model that lacks the semantic clarity and constraint enforcement of the original database.

This thesis contributes a **systematic and formal approach** for mapping a complete relational database including its data, schema, constraints, and queries into an equivalent property graph database.

This comprehensive mapping facilitates a smooth and reliable transition from relational to graph databases. It provides a solid foundation for benchmarking graph database performance using established benchmarks like TPC-H, by

enabling a semantically equivalent graph representation of the relational schema and workload. By preserving the full semantics of the source database, our work ensures that migrations are not just data transfers, but are trustworthy transformations of the entire data model.

Precisely, our mapping not only transforms data and queries but also satisfies all essential mapping properties including information preservation, semantic preservation, monotonicity, and query/update preservation along with formal proofs of consistency and efficiency. A proof of concept has been implemented between PostgreSQL and Neo4j.

2.3 Contribution 2: Distributed evaluation of graph queries

As graph datasets scale, a single-machine architecture becomes a bottleneck for both storage and query processing. This stems from the semantics of graph querying, which often rely on subgraph isomorphism a core operation used to match query patterns against the data graph. Subgraph isomorphism is an NP-complete problem, meaning its computational complexity grows exponentially with the size of the input. As a result, query execution becomes increasingly intractable as data volume and structural complexity grow, making distributed processing a necessity for scalable and efficient graph analytics.

Distributing the graph across a cluster of machines is necessary for scalability, but it introduces significant complexity for query execution. Challenges include minimizing expensive network communication during traversals, maintaining consistency across partitions, and optimizing complex queries like multi-hop pattern matching in a distributed setting.

This thesis introduces a **formal framework for querying distributed graph databases**. The framework addresses the core challenges of distribution by providing a rigorous formal model for representing graph partitions and

defining query execution plans. Using Neo4j and its Cypher query language as a concrete use case, we demonstrate the practical application of our theory.

Precisely, the framework is centered on three formal algorithms that handles user query fragmentation for distributed execution, assembling the partial results returned from each fragment and coordinates all these operations between the coordinator and the fragments, managing them in a parallel and multi-threaded manner.

By combining a theoretical foundation with a practical case study, this contribution paves the way for more scalable and reliable distributed graph databases. It provides a blueprint for system architects and a basis for future research into distributed graph query optimization.

At the time of writing this thesis, and publishing our results of distributed GPM, Neo4j reveals a new system called *InfiniGraph* aimed to perform distributed querying over large graphs. However, no complete documentation or access is provided yet in order to make comparison with our results. Thus, our work remains the first one that provides a strong formal algorithms to conduct GPM in a distributed context.

3 Outline of Dissertation

The thesis is structured as follows:

- Chapter 1: introduces the theoretical concepts and fundamental technologies essential for comprehending the research conducted in this dissertation.
- Chapter 2: proposes a complete direct mapping from relational databases to graph databases, covering data, schema, constraints, and queries.
- Chapter 3: presents a formal framework for distributed query evaluation over a data graph, along with an implementation that demonstrates the concept.
- Chapter 4: develops *R2G-ETL*, a tool for mapping any relational database including its schema, data, and queries to a graph database.

Each chapter is based on at least one scientific communication/publication.

In the final stage, we give a general conclusion to the thesis as whole.

Chapter I:
Graph databases: State of the Art

1 Basic Graph notations

To ensure clarity and precision throughout this thesis, this section introduces the fundamental graph concepts that underpin our study.

1.1 Property Graph Data

Definition 1 (Property graph)**.** *A* PG *is a directed multi-graph* $G=(V, E, L, A, \mathfrak{I})$ *where: 1)* V *is a finite set of nodes; 2)* $E \subseteq V \times V$ *is a finite set of edges in which* (v, v') *denotes an edge from node* v *to* v'*; 3)* L *is a function that assigns a label* $L(v)$ *(resp.* $L(e)$*) to each node* $v \in V$ *(resp. edge* $e \in E$*); 4) for each node* $v \in V$ *(resp. edge* $e \in E$*),* $A(v)$ *(resp.* $A(e)$*) is a tuple* $(A_1 = c_1, \ldots, A_n = c_n)$ *where:* A_i *is an attribute of* v *(resp.* e*),* c_i *is a constant value,* $n \geq 0$*, and* $A_i \neq A_j$ *if* $i \neq j$*; 5) and finally* $\mathfrak{I}$ *is a function that assigns a unique identifier to each node and edge of* G*.* □

Intuitively, the label of a node represents an entity (e.g. *Person*, *Movie*) while the label of an edge represents a relationship (e.g. a *friendship* between two persons, or a *direction* role between a person and a movie). Moreover, attributes specify properties of both entities (e.g. *age* of a person) and relationships (e.g. *starting year* of a friendship). Notice that there may be several edges between the same pair of nodes, which express different relationships. In addition, two nodes/edges with the same label may have different attribute sets. We use the term element to refer to both nodes and edges where it is clear in the context. As a common practice, we use the function $\mathfrak{I}$ to attribute a unique identification to each element in the graph data.

Definition 2 (Subgraphs)**.** *We say that* $G_s=(V_s, E_s, L_s, A_s, \mathfrak{I}_s)$ *is a* subgraph *of* $G=(V, E, L, A, \mathfrak{I})$*, denoted by* $G_s \subseteq G$*, if: a)* $V_s \subseteq V$*; b)* $E_s \subseteq E$*; and c) for each element* $x \in E_s \cup V_s$*:* $L_s(x)=L(x)$*,* $A_s(x)=A(x)$ *and* $\mathfrak{I}_s(x)=\mathfrak{I}(x)$*.* □

1.2 Graph Query

We give hereafter the syntax of graph queries (isomorphism semantics) from both academic and commercial point of view.

1.2.1 The formal definition

Definition 3 (Graph Pattern)**.** *A* ***Graph Pattern*** *is a directed multi-graph* $Q=(V_q, E_q, L_q, A_q, \vartheta)$ *where: 1)* (V_q, E_q, L_q) *are defined as for graph data; 2) for each element* $x \in V_q \cup E_q$*,* $A(x)$ *is a tuple* $(A_1\ op_1\ c_1, \ldots, A_n\ op_n\ c_n)$ *where:* A_i *is an attribute*

of the element x, c_i is a constant value, $n \geq 0$, and $op_i \in \{=, >, <, \neq\}$; 3) and finally ϑ is a function that assigns a unique variable to each node and edge of Q. □

Intuitively, Q is composed by nodes and edges connected together to form a pattern that one would like to look for on the graph data. Moreover, nodes and edges of Q may have attribute-based conditions that allow to extract complex insights from graph data. For each element x, its variable $\vartheta(x)$ specifies a unique identification of it. That is, $\vartheta(x_1) \neq \vartheta(x_2)$ for each $x_1, x_2 \in V_q \cup E_q$.

Definition 4 (Subpattern)**.** *We say that $Q^s = (V_q^s, E_q^s, L_q^s, A_q^s, \vartheta^s)$ is a* subpattern *of $Q = (V_q, E_q, L_q, A_q, \vartheta)$, denoted by $Q^s \subseteq Q$, if: a) $V_q^s \subseteq V_q$; b) $E_q^s \subseteq E_q$; and c) for each element $x \in E_q^s \cup V_q^s$: $L_q^s(x) = L_q(x)$, $A_q^s(x) = A_q(x)$ and $\vartheta_q^s(x) = \vartheta I_q(x)$.* □

1.2.2 The Neo4j definition

Cypher is a declarative language developed by Neo4j that enables users to interact with property graph databases efficiently, making it easier to express complex queries and explore graph-structured data.

Cypher queries. We consider a simple but useful class of Cypher queries [10] that are largely used in practice. A Cypher query C is specified formally as:

C ::= $\mathcal{M}$ $\mathcal{W}$ $\mathcal{R}$
$\mathcal{M}$ ::= **MATCH** ρ
$\mathcal{W}$ ::= **WHERE** ω | ε
$\mathcal{R}$ ::= **RETURN** *
ρ ::= ρ' | ρ',ρ
ρ' ::= (m) | (m)-[m]->ρ' | (m)<-[m]-ρ' | (m)-[m]-ρ'
m ::= v:l | v:l{σ} | v
σ ::= a:val | σ,σ
ω ::= ω AND ω | ω OR ω | NOT ω | (v.a op val)

A Cypher query C consists of three types of clauses: Match, Where, and Return. The Match clause specifies the patterns to search for in the data graph. It is composed of one or more path patterns. Each path pattern (defined by ρ') can be either a single vertex or a sequence of vertices and relationships. Each vertex (m) (resp. relationship $[m]$) must be defined by a variable v, a label l, and an optional set σ of attribute values assignment (e.g., $(v\{a : val\})$ specifies that, for an element referenced to by

the variable v, its attribute a must have the value val). For any path pattern, if a variable v is given no label, then v must have been defined in a previous path pattern. The Where clause is optional and allows adding attribute conditions to vertices and relationships retrieved by the Match clause. These conditions may be simple (defined over one element) or composed (defined over many elements) and allow to filter data for concise search. Once the elements are retrieved from the graph data and possibly refined, the Return clause produces a set of tuples, each containing a match of each element of the query.

Example 1. *Consider the following Cypher query:*

$$
\begin{aligned}
&\texttt{Match } (v1\!:\!Person) - [r1\!:\!Director] - > (v2\!:\!Movie)\texttt{,}\\
&(v3\!:\!Person) - [r2\!:\!Director] - > (v2)\texttt{,}\\
&\ (v1) - [r3\!:\!Friend] - (v3)\\
&\texttt{Where } v2.\texttt{type} = \text{``}\texttt{Action}\text{''} \texttt{ AND } r1.\texttt{year} = 2018\\
&\texttt{Return *}
\end{aligned}
$$

It is composed by three path patterns looking for two friends which have directed an action movie at 2018*. The result of this Cypher query is a set of tuples of the form:*

$$\langle v1{=}?, r1{=}?, v2{=}?, v3{=}?, r2{=}? \rangle$$

Each tuple is a mapping of each variable of the Cypher query into an identifier of the graph data. □

1.3 GQL Standard

The GQL standard [11], officially published as ISO/IEC 39075 on April 11, 2024, is the first new database language adopted by ISO since SQL in 1987. Developed by the same ISO/IEC committee responsible for Structured Query Language (*SQL*) [22], GQL addresses the growing need for a standardized query language for *property graph databases*.

Inspired by the success of SQL and closely modeled after *Cypher*, the query language originally developed by Neo4j, GQL offers a *declarative* approach to querying, updating, and managing graph data. Many of its core constructs, particularly in Graph Pattern Matching (*GPM*), are *derived directly from Cypher*, making the two languages *highly interoperable*.

GQL supports both schema-free and schema-defined graphs, a wide range of data types, and powerful pattern matching capabilities. It draws influence from other graph languages such as PGQL [9], GSQL [23], and G-Core [24], but its syntax and semantics remain closest to *Cypher*, establishing it as the natural evolution of Cypher into an ISO standard.

Current implementations and early adopters focus on use cases such as fraud detection, supply chain analysis, drug discovery, and knowledge graphs for AI, where native graph capabilities offer significant advantages over relational models.

2 Graph pattern matching

Graph query processing involves evaluating queries over graph-structured data, where the semantics of the query determine how the results are computed and interpreted. Two primary approaches to query evaluation are approximate answers and exact answers, each with distinct semantics. Approximate querying leverage techniques such as simulation semantics, including dual simulation, bounded simulation, and strong simulation, to provide flexible and efficient query results, particularly in large-scale or noisy graphs. Exact querying, on the other hand, rely on isomorphism semantics to ensure precise, one-to-one matches between the query and the data graph. Additionally, commercial graph database systems, such as those supporting the Cypher query language, implement specific execution semantics that may differ from theoretical models. These systems often incorporate optimizations like indexing, parallel processing, and caching to efficiently handle isomorphism-based queries. This section explores these concepts in detail, focusing on simulation semantics, isomorphism semantics, the execution of graph simulation in commercial systems, and the execution of Cypher queries in commercial systems.

2.1 Exact querying

Isomorphism semantics is the gold standard for exact graph query matching. It requires a one-to-one correspondence between the nodes and edges of the query graph and the data graph, ensuring that the structure and labels of the graphs are identical.

2.1.1 Subgraph Isomorphism

Definition 5 (Subgraph Isomorphism)**.** *The graph data* $G = (V, E, L, A, \mathfrak{I})$ *matches the graph pattern* $Q = (V_q, E_q, L_q, A_q, \vartheta)$ *via* subgraph isomorphism *if there exists a*

subgraph $G_s = (V_s, E_s, L_s, A_s, \mathfrak{I}_s)$ *of* G *that is* isomorphic *to* Q *and satisfies its attribute conditions. Formally, there exists a bijective function* h *from* E_q *to* E_s *such that:*

1. *for each pattern edge* $e_q = (u, u') \in E_q$*, there exists a data edge* $e_s = (v, v') \in E_s$ *with* $L_q(e_p) = L_s(e_s)$*,* $L_q(u) = L_s(v)$ *and* $L_q(u') = L_s(v')$*;*
2. *for each pattern edge* $e_q = (u, u')$ *with* $h(e_q) = e_s = (v, v')$*, attribute values on* e_s *satisfy all attribute conditions of* e_q*, and moreover, attribute values on* v *(resp.* v'*) satisfy all attribute conditions of* u *(resp.* u'*).* □

Therefore, for each pattern element $x_q \in V_q \cup E_q$, $h(x)$ maps it to a data element in $x_s \in V_s \cup E_s$. In other words, the variable $\vartheta(x_q)$ is mapped to the identifier $\mathfrak{I}(x_s)$. If G_s satisfies the above conditions, then a ***match*** of Q in G is given as a tuple $t = \langle x_1 = id_1, \cdots, x_n = id_n \rangle$ that contains all variables in Q (i.e. $x_1, \cdots, x_n$) with their associated identifiers in G_s (i.e. $id_1, \cdots, id_n$). The ***match result*** of Q in G, denoted by $Q(G)$, is the set $Q(G) = \{t_1, \cdots, t_m\}$ of all matches of Q in G where: $m \geq 0$ and each t_i is a match based on some subgraph of G.

Remark that the bijection is defined between edges of the query and those of the data subgraph. This semantics is called *edge isomorphism* and it is supported by all graph database systems such as Neo4j. *GPM* is mostly defined in the literature with a bijection between nodes instead of edges, which is called *node isomorphism*. This semantics can be supported within existing graph systems easily by forcing inequality conditions between all nodes of the query.

Note that the decision problem of subgraph isomorphism is *NP-complete* [25, 26].

2.1.2 Subgraph Homomorphism

Definition 6 (Subgraph Homomorphism)**.** *A* subgraph homomorphism *is a function* $\psi : V_q \to V$ *(not necessarily injective) satisfying:*

1. $\forall v \in V_q, L_q(v) = L(\psi(v))$
2. $\forall (u, v) \in E_q, (\psi(u), \psi(v)) \in E$ *and* $L_q(u, v) = L(\psi(u), \psi(v))$
3. *For each* $x \in V_q \cup E_q$*,* $A_q(x)$ *conditions are satisfied by* $\psi(x)$ *in* G

Example 2. *For the same* G*, consider pattern* Q' *with 3 nodes* u_1*,* u_2*,* u_3 *and the 2 edges* $u_1 \to u_2$*,* $u_2 \to u_3$ *: The mapping* $\psi(u_1) = v_1$*,* $\psi(u_2) = v_2$*,* $\psi(u_3) = v_2$ *is a valid homomorphism.* □

The complexity of subgraph homomorphism is *NP-complete* [27].

2.1.2.1 Difference with Subgraph Isomorphism

The key distinctions between subgraph homomorphism and isomorphism in property graphs are:

Criterion	**Subgraph Isomorphism**	**Subgraph Homomorphism**
Injectivity	Required (injective ϕ)	Not required
Edge Correspondence	Bijective	Only forward mapping
Non-adjacency	Preserved	Not preserved
Self-loops	Forbidden	Allowed

Table 2: Structural comparison of graph pattern matching semantics.

Example 3. *Consider a graph query Q composed of three nodes $\{u_1, u_2, u_3\}$ and three edges $\{(u_1, u_2), (u_2, u_3), (u_3, u_1)\}$. Let the data graph G consist of a single edge (v_1, v_2), with matching labels.*

Subgraph Isomorphism: *Not applicable - there is no injective mapping from the three query nodes to two distinct nodes in G.*

Subgraph Homomorphism: *A valid mapping exists: $\psi(u_1) = \psi(u_2) = \psi(u_3) = v_1$, provided that:*

1. *The label conditions are satisfied ($L(u_i) = L(v_1)$ for all i), and*
2. *There are no conflicting attribute constraints.*

This example illustrates how the relaxed constraints of homomorphism enable solutions that are not possible under isomorphism. □

2.1.3 Sequential algorithms for subgraph isomorphism

Subgraph isomorphism is a well-known NP-Complete problem [28] that has been extensively researched over the years. There are two main approaches to solving it: exact and approximate algorithms. Exact algorithms identify all possible subgraphs matching the query, but their computational complexity grows exponentially. In contrast, approximate algorithms trade accuracy for efficiency, offering faster execution and lower memory usage at the cost of less precise results. Previous surveys [29, 30, 31, 32] summarize various techniques developed for both exact and inexact subgraph isomorphism evaluation from the 1970s up to 2015.

Traditionally, subgraph isomorphism focuses solely on structural matching without considering semantic similarities, which are often encoded in vertex and edge labels and attributes. However, newer approaches integrate semantic information alongside structural patterns to enhance matching accuracy.

Ullmann [33] proposed a tree-based algorithm for graph isomorphism that systematically explores potential mappings between query vertices and data graph vertices using an initial mapping matrix. The approach incorporates a refinement step to reduce the search space by discarding invalid matches early. This pruning strategy relies on the principle that if a query vertex u is mapped to a data vertex v, their respective neighbors must also be compatible. Any mapping failing this condition is immediately eliminated. Subsequent research builds upon Ullmann's foundational work, primarily focusing on two key challenges: (1) optimizing query vertex ordering to minimize intermediate results, and (2) developing effective pruning techniques to eliminate redundant computations.

The VF2 algorithm [34] was one of the first approaches to address graph isomorphism using a state space representation. It systematically explores the search space by examining the neighbors of already matched query vertices. Building upon VF2, VF2Plus [35] enhances vertex ordering by prioritizing query vertices that are both less likely to find matches in the data graph and highly connected to previously ordered vertices. Further improvements were introduced in VF3 [36], which incorporates heuristics to reduce the search space and optimize state processing time. Later, VF3-Light [37] streamlined this approach by removing certain heuristics from VF3, resulting in faster execution times.

Several algorithmic approaches have been developed for subgraph isomorphism evaluation. Some methods, including QuickSI [38], RI [39], and VF2++ [40], employ data graph exploration techniques. Another category of algorithms utilizes precomputed tree or graph indices to store candidate sets, as seen in GraphQL [41], TurboIso [42], BoostIso [43], CFL-Match [44], TurboFlux [45], DAF [46], and VC [47]. Additionally, certain approaches like GADDI [48] and SPath [49] create specialized indices derived from the data graph structure to facilitate embedding enumeration. A distinct approach is presented in MQO-iso [50], which builds a Directed Acyclic Graph (DAG) representing inter-query relationships to optimize the search process.

Unlike Ullmann's approach which lacks query vertex ordering, QuickSI [38]

introduced a strategy prioritizing vertices with rare labels in the data graph to minimize unnecessary computations. The algorithm optimizes processing by first examining edges and vertices with fewer potential embeddings, thereby reducing the search space. Its methodology involves: (1) selecting a spanning tree that minimizes potential embeddings, (2) determining processing order by prioritizing edges with the smallest candidate sets for their endpoints, and (3) enumerating embeddings through Depth-First Search (DFS) traversal of the spanning tree.

Similar to QuickSI, both RI [39] and VF2++ [40] directly explore the data graph without precomputing candidate sets. However, they employ distinct vertex selection strategies: RI prioritizes vertices with higher degrees and greater connectivity to already ordered vertices, while VF2++ selects query vertices based on label rarity in the data graph (PG) combined with degree centrality in the query graph (Q).

To enhance pruning efficiency, GraphQL [41] introduced a neighborhood signature technique for each query vertex u, defined by the label set of its adjacent vertices. During matching, a data vertex v is eliminated if its neighborhood signature fails to subsume that of its corresponding query vertex. GraphQL's vertex selection strategy prioritizes query vertices that (1) connect to already matched vertices and (2) possess the smallest candidate sets.

Building upon this concept, SPath [49] developed an optimized neighborhood signature method to further reduce candidate set sizes. Additionally, SPath employs a path-at-a-time matching approach to improve search efficiency.

TurboISO [42] presents an alternative approach that optimizes subgraph isomorphism search through structural grouping and region-based processing. The algorithm begins by clustering query vertices sharing identical labels and neighborhood patterns into consolidated nodes, effectively minimizing redundant processing of unpromising matches. Its methodology involves partitioning the data graph into distinct Candidate Regions (CRs), each anchored at a data node matching the query tree's root node.

For efficient enumeration, TurboISO employs a strategic combination of permutation and combination operations within each CR. The algorithm processes query vertices sequentially - when a vertex mapping produces a valid partial solution, the system generates compatible combinations from related vertices in its group. Conversely, if a vertex fails to yield an acceptable match, TurboISO efficiently prunes all associated

vertices in the group from further consideration, significantly reducing the search space.

2.1.4 Parallel algorithms for subgraph isomorphism

To enhance the scalability of subgraph isomorphism algorithms, two main parallel processing paradigms have emerged: (1) relational join-based approaches and (2) exploration-based techniques.

Join-based approaches.

In relational approaches for subgraph isomorphism, edges from the data graph are modeled as database tables where each unique label pair constitutes a separate relation, with vertex labels serving as attributes and edge identifiers as attribute values. The matching process involves filtering these relations and performing successive join operations. Due to the computational expense of joins, researchers have developed several optimization strategies: (1) binary joins that iteratively combine pairs of edge candidate tables [51, 52, 53, 54] with optimized ordering to reduce intermediate results; (2) structural joins using larger components like star patterns (two-level trees) identified through localized graph exploration [55]; and (3) advanced techniques employing specialized joining units such as TwinTwig [56], Crystal [57], and Clique-based structures [58, 59], all designed to minimize intermediate data volume while ensuring complete subgraph matching.

Afrati et al. [60] introduced a distinct multiway join strategy that simultaneously processes all edge candidate sets, contrasting with sequential join methods. This approach leverages worst-case optimal join algorithms, particularly GenericJoin [61] and LeapFrog TrieJoin [62], which have inspired parallel implementations for subgraph isomorphism detection [63, 64]. Building on these foundations, Mhedhbi et al. [65] developed a hybrid technique integrating both traditional binary joins and the GenericJoin algorithm. The field has subsequently evolved to include systematic evaluations, as demonstrated by Lai et al. [59] who performed a comprehensive empirical study of join-based graph pattern matching *(GPM)* algorithms, ultimately proposing a tripartite classification system based on their underlying join methodologies.

Exploration-based approaches.

These graph exploration methods align naturally with modern TLAV (Think-Like-A-Vertex) paradigms for distributed graph processing. We highlight

four representative systems here as exceptions: QFrag [66], CECI [67], PSM [68], and BENU [69]. QFrag and CECI employ a data replication strategy, distributing graph fragments across multiple machines where each processor conducts independent subgraph isomorphism searches. In contrast, PSM implements parallel search within a single machine environment. BENU shares conceptual similarities as a parallel GPM system, though with distinct architectural implementations. These systems collectively demonstrate the spectrum of parallel processing approaches for graph pattern matching.

QFrag [66] employs a distributed approach that replicates the data graph across k worker nodes to parallelize TurboISO's subgraph isomorphism search while ensuring load balancing through task fragmentation. The system decomposes processing into two distinct phases: in the first super-step, each worker builds candidate regions *(CRs)* from assigned root vertices, treating each CR's embedding enumeration as an independent subtask and estimating processing times to identify computationally intensive outlier trees. Regular trees are processed locally during this phase, while outlier trees are dynamically partitioned into balanced subtasks and redistributed among workers for processing in the second super-step. This two-phase design combines data parallelism through graph replication with computational parallelism via adaptive task fragmentation, maintaining TurboISO's matching accuracy while achieving scalable performance through intelligent workload distribution and outlier handling.

CECI [67] utilizes a Breadth-First Search (BFS)-based spanning tree index derived from the query graph structure. The index maintains: (1) mappings between query nodes (u) and their corresponding data graph matches (v), (2) candidate sets for tree edges connecting these nodes, and (3) additional candidates for non-tree edges between neighbors. During parallel execution, each candidate match for the query tree's root serves as a pivot for an Embedding Cluster (EC), with the system dynamically balancing workload by: (a) estimating EC processing costs, (b) decomposing computationally intensive ECs into smaller units, and (c) distributing equal EC quantities across workers. For embedding enumeration, CECI employs set intersection operations between tree-edge and non-tree edge candidates stored in its index structure, ensuring comprehensive subgraph isomorphism detection while maintaining parallel efficiency.

SM [68] parallelizes backtracking-based subgraph isomorphism search with DFS traversal on a single machine by decomposing the candidate tree index into independent search regions, each processed by separate worker threads. To maintain load balance, the system dynamically monitors workload distribution and splits overloaded regions into equally-sized subtasks when necessary, ensuring efficient utilization of computational resources while preserving the algorithm's completeness. This shared-memory approach combines the systematic exploration of traditional backtracking methods with fine-grained parallelism through (1) region-based decomposition of the search space and (2) adaptive workload redistribution, demonstrating how careful parallelization can enhance performance without sacrificing result quality in memory-constrained environments.

Finally, BENU system [69] implements a parallelized backtracking algorithm that partitions the search tree into distinct regions for distributed processing across worker nodes. Each worker utilizes adjacency list representations of the data graph to efficiently compute potential embeddings via iterative set intersection operations. The framework incorporates an adaptive load-balancing mechanism that dynamically decomposes tasks when encountering root vertices exceeding a predefined degree threshold, ensuring equitable workload distribution while maintaining the method's systematic search properties. This threshold-based decomposition strategy effectively balances the trade-off between parallelization granularity and computational overhead in subgraph enumeration tasks.

2.2 Approximate querying

This section explores Graph Simulation Semantics (GSS), which includes graph simulation and some of its extensions: dual simulation, strong simulation and bounded simulation. While traditional GPM requires exact structural matches, GSS-based models relax these constraints to capture more flexible similarities, even when results deviate from the query's original structure. This adaptability is crucial for addressing key challenges in graph query processing, particularly in applications requiring approximate matches, such as plagiarism detection.

2.2.1 Graph simulation

Graph Simulation (GS) [70] is a flexible matching model that permits a *one-to-many* mapping between query vertices and data vertices. Unlike exact structural matching,

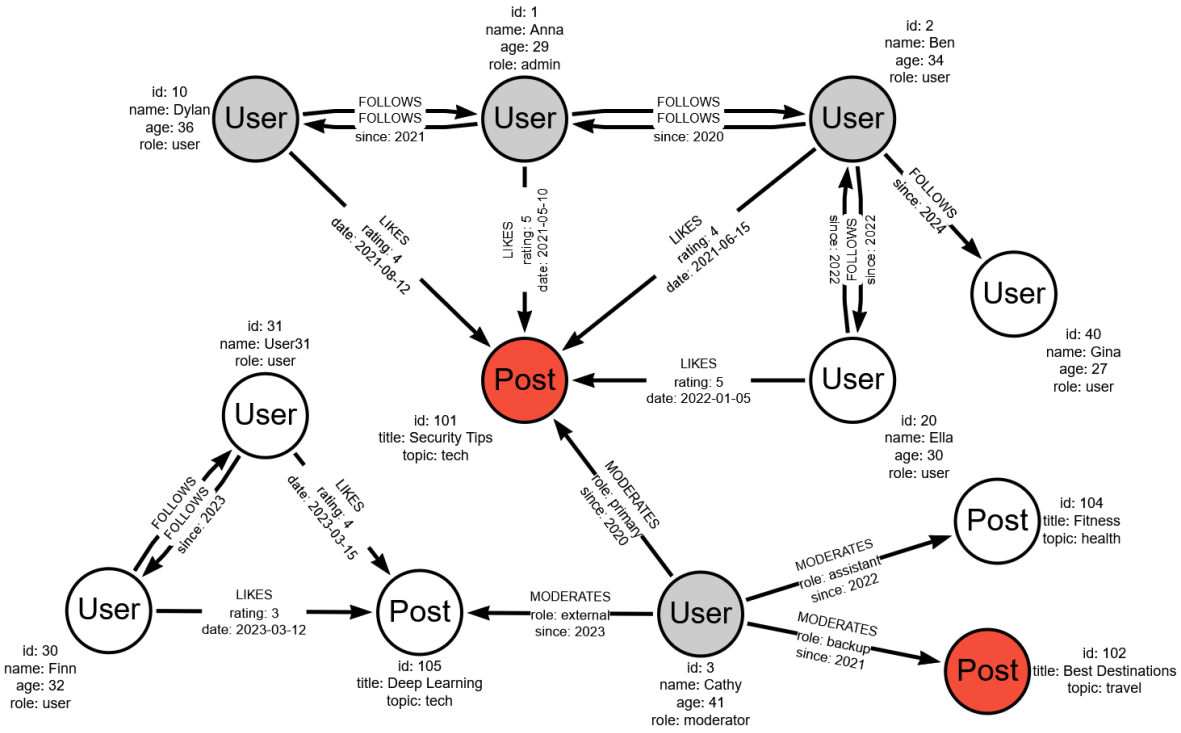


Figure 2: Graph simulation results (colored in gray and red).

GS independently maps each edge in the pattern graph to corresponding edges in the data graph, producing a bijective match.

Formally, given a data graph $G(V, E, f)$ and a query graph $Q(V_q, E_q, f_q)$. A vertex $v \in V$ *matches* a vertex $u \in V_q$ via graph simulation if:

1. $\forall (u, v) \in R$, $f(u) = f_q(v)$ i.e. their labels are identical.
2. $\forall (u, u') \in E_q$, $\exists (v, v') \in E$ such that $(u', v') \in R$

We say that Q matches G via GS if a *total match relation* $R \subseteq V_q \times V$ exists where:

$$\forall u \in V_q, \exists v \in V \text{ such that } (u, v) \in R \tag{1}$$

GS can be computed efficiently in *polynomial time* (quadratic time) using optimized algorithms such as: The HHK algorithm [71] or Alternative approaches described in [72].

Example 4. *Given the data graph G in Figure 1(a) and the pattern graph Q in Figure 1(b), the application of graph simulation returns a match set, as shown in Figure 2. The nodes colored in the data graph (gray for the User label and red for the Post label) represent the final results.* □

2.2.2 Dual simulation

Dual Simulation (DS) [73] generalizes graph simulation by enforcing constraints on both incoming and outgoing edges. Given a data graph $G = (V, E, f)$ and a pattern graph $Q = (V_q, E_q, f_q)$. A vertex $v \in V$ is said to *dual-simulate* a vertex $u \in V_q$ if and only if:

1. $\forall (u, v) \in R$, $f_q(u) = f(v)$, i.e. u and v have the same label.
2. $\forall u \in V_q$, $\exists v \in V$ such that $(u, v) \in R$ and:
   (a) $\forall (u, u') \in E_q$, $\exists (v, v') \in E$ such that $(u', v') \in R$ (child relationship),
   (b) $\forall (u'', u) \in E_q$, $\exists (v'', v) \in E$ such that $(u'', v'') \in R$ (parent relationship).

Furthermore, the maximum dual match set is defined as follows.

Definition 7 (Maximum Dual Match Set)**.** *Given a data graph $G = (V, E, f)$ and a query graph $Q = (V_q, E_q, f_q)$, let $R \subseteq V_q \times V$ be a dual match relation under dual simulation constraints. The* ***maximum dual match set*** *R_d is the unique relation satisfying:*

1. *R_d is a dual match relation*
2. *For any dual match relation R, $R \subseteq R_d$*

This yields the maximal set of vertex pairs preserving both parent and child relationships in G for Q.

Example 5. *Given the data graph G in Figure 1(a) and the pattern graph Q in Figure 1(b), the application of dual simulation returns a match set, as shown in Figure 3. The nodes colored in the data graph (gray for the User label and red for the Post label) represent the final results.* □

2.2.3 Strong simulation

Strong simulation (SS) [73] extends dual simulation by introducing **data locality constraints**, ensuring that pattern matching occurs only within a restricted neighborhood of the data graph. Specifically, for a given vertex v in the data graph G, matches are confined to a **ball** b centered at v with radius d_q, where d_q represents the diameter of the pattern graph Q. The subgraph of G within this ball is denoted as $\hat{G}[v, d_q]$.

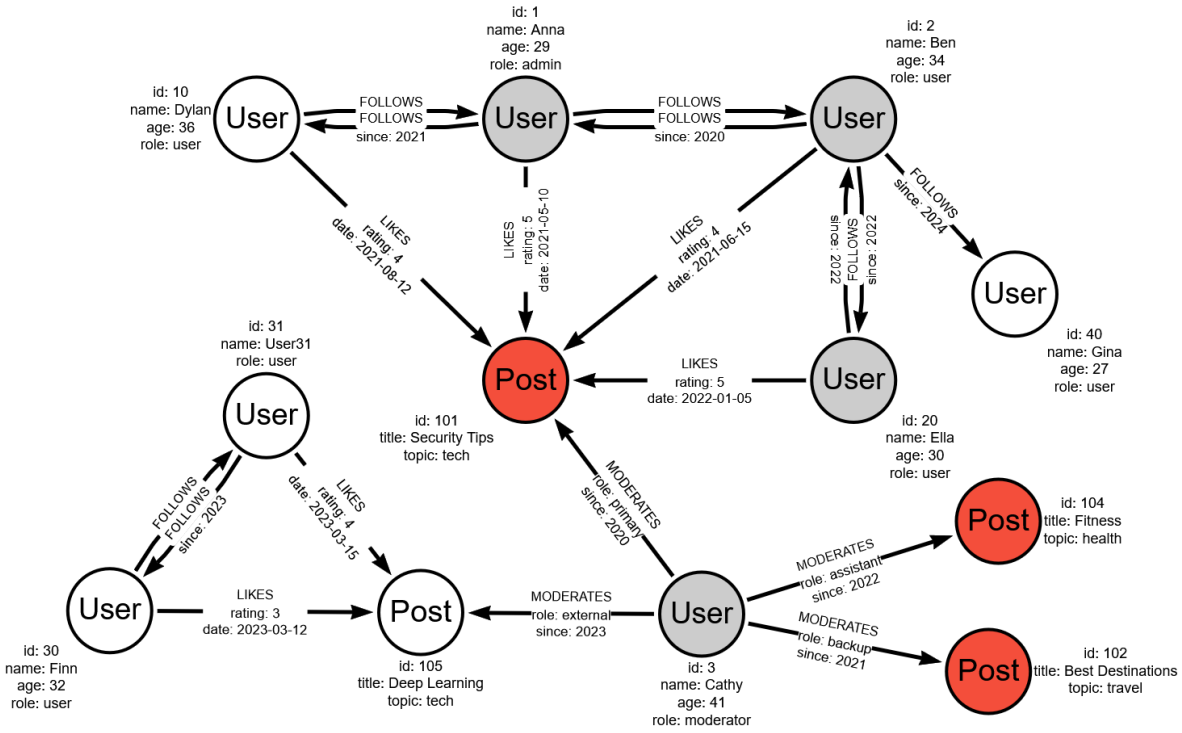


Figure 3: Dual simulation results (colored in gray and red).

A graph G is said to match Q under strong simulation if:

1. Q matches $\hat{G}[v, d_q]$ via dual simulation, producing a maximal match set $R_b^{d_q}$ within the ball. i.e. Local Dual Simulation.

2. The central vertex v must participate in at least one match pair in $R_b^{d_q}$. i.e. Center Vertex Inclusion.

Additionally, for each vertex v in G, the largest connected subgraph derived from the dual simulation results within its respective ball while ensuring v is included is termed a **maximum perfect subgraph (Max-PG)** of G relative to Q. The authors demonstrated that strong simulation not only enhances topological matching accuracy but also maintains computational efficiency, as their proposed algorithm operates in *cubic time*, matching the complexity of traditional graph simulation.

Example 6. *Given the data graph G in Figure 1(a) and the pattern graph Q in Figure 1(b), the application of strong simulation returns a match set, as shown in Figure 4. The nodes colored in the data graph (gray for the User label and red for the Post label) represent the final results.* □

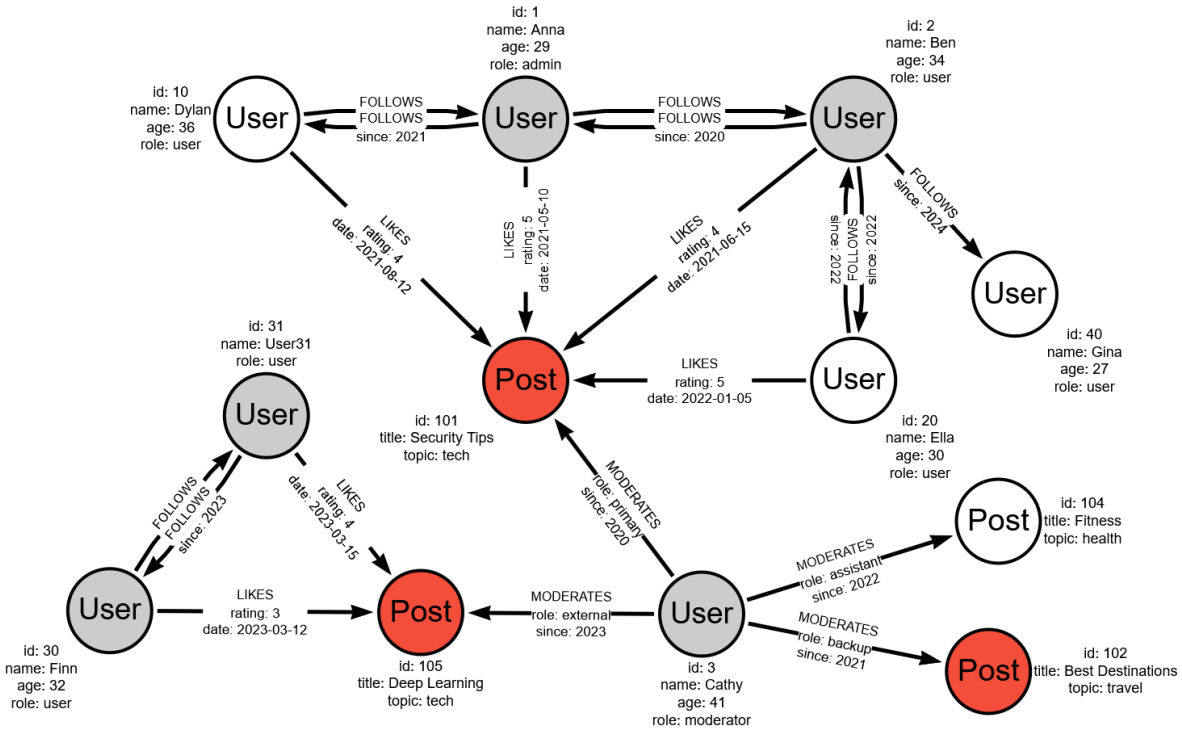


Figure 4: Strong simulation results (colored in gray and red).

2.2.4 Bounded simulation

Bounded Simulation (BS) [74] extends the concept of graph simulation, addressing limitations in traditional subgraph isomorphism, which struggles to effectively model similarity in modern applications such as social network community detection. In this framework, a **bounded path** p within a graph G is defined as a sequence of vertices $v_1, v_2, \ldots, v_n \in V$ where every pair of consecutive vertices (v_i, v_{i+1}) is connected by an edge in E.

The data graph G is represented as a triplet (V, E, f_A), where f_A replaces the conventional labeling function. Here, $f_A(u)$ assigns each node $u \in V$ a tuple of attributes $(A_1 = a_1, \ldots, A_n = a_n)$. On the other hand, the **pattern graph** Q is defined as (V_q, E_q, f_v, f_e), incorporating two key functions:

- $f_v(u)$: A set of attribute-based predicates that must hold for a matching node.
- $f_e(u, u')$: A constraint on edge mappings, specifying the maximum allowable path length (denoted as k) between matched nodes in G (where $f_e(u, u') = \{k, *\}$ and $*$ indicates no length restriction).

A data graph G **matches** a pattern graph Q under bounded simulation if there exists a binary relation $R \subseteq V_q \times V$ satisfying:

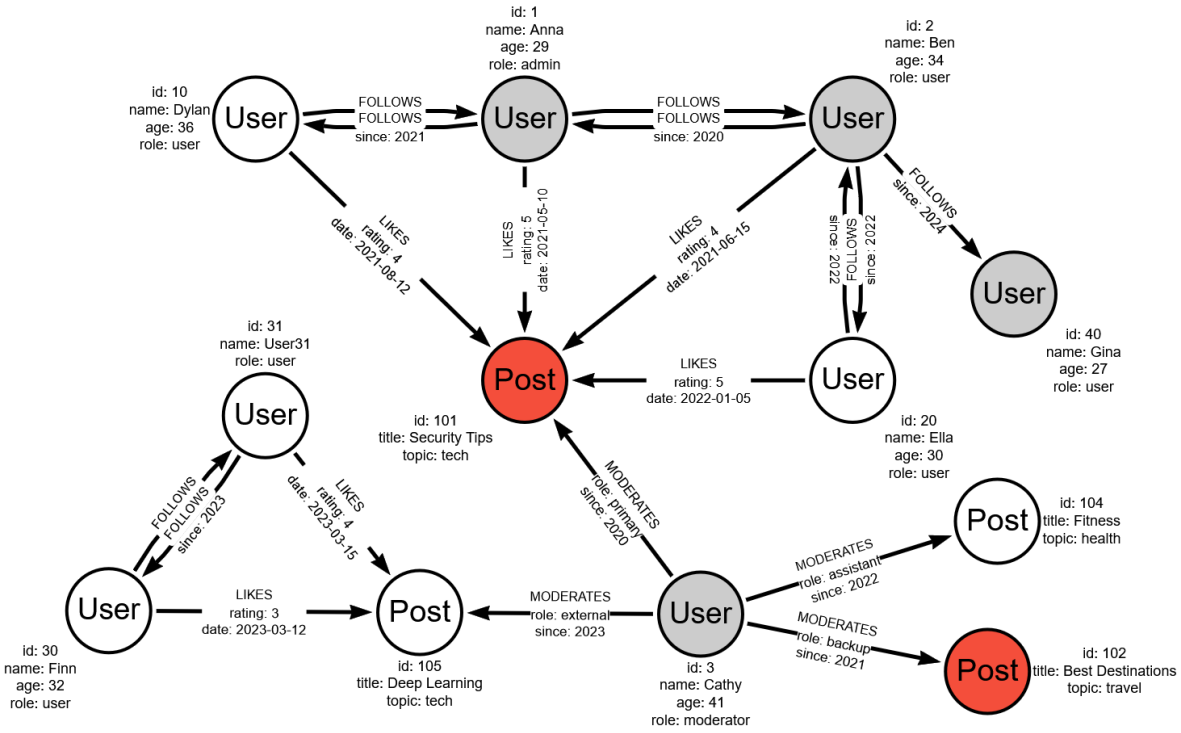


Figure 5: Bounded simulation results (colored in gray and red).

1. Every node in V_q must map to at least one node in V.
2. For each $(u, v) \in R$:
   - The node v's attributes must satisfy u's predicate $f_v(u)$.
   - For every edge $(u, u') \in E_q$, there must exist a corresponding bounded path p (of length $\leq k$ if $f_e(u, u') = k$) from v to some v' in G, where $(u', v') \in R$.

Notably, classical graph simulation is a restricted case of bounded simulation where edges must map directly (without intermediate paths). The authors of [74] introduced a *cubic time* algorithm to compute bounded simulation and later developed incremental methods to handle dynamically updated graphs efficiently.

Example 7. *Given the data graph G in Figure 1(a) and the pattern graph Q in Figure 1(b), the application of Bounded simulation (with k=2) returns a match set, as shown in Figure 5. The nodes colored in the data graph (gray for the User label and red for the Post label) represent the final results.* □

2.3 Real-life applications of graph pattern matching

GPM have been widely applied across various domains, including document processing (handwritten text and symbol recognition [75, 76, 77, 78]), biometric

identification (fingerprint, retina, and facial recognition [79, 80, 81]), computer vision (image retrieval and video object tracking [82, 83, 84]), and social network analysis (community detection, expert finding, and fraud detection [85, 86, 87]). Recent advancements have extended GPM to cybersecurity for intrusion and anomaly detection [88, 89, 90]. These techniques continue to evolve, demonstrating their versatility in solving complex real-world problems through structural pattern recognition.

Moreover, GPM serves as a fundamental operation in popular commercial graph database systems, such as Neo4j [91] and RDF query engines [92].

2.4 Graph pattern matching in commercial systems

Commercial graph database systems, such as Neo4j, Amazon Neptune, and TigerGraph, implement the Cypher query language to interact with graph data. The execution semantics of Cypher queries in these systems often involve a combination of theoretical models and practical optimizations to balance performance and accuracy.

- **Edge-Isomorphism vs. Node-Isomorphism**:
  - **Edge-Isomorphism**: In many commercial systems, edge-isomorphism is the default semantics for query execution. This means that the system ensures a one-to-one correspondence between the edges of the query graph and the data graph, while allowing flexibility in node mappings. This approach is efficient and aligns with the way many graph algorithms are implemented [93].
  - **Node-Isomorphism**: Node-isomorphism, on the other hand, requires a strict one-to-one correspondence between nodes in addition to edges. While this provides stricter matching, it is computationally more expensive and is less commonly used in commercial systems [94].
  - **Trade-offs**: The choice between edge-isomorphism and node-isomorphism depends on the application requirements. Edge-isomorphism is often preferred for its efficiency, while node-isomorphism is reserved for scenarios requiring exact node-level matching [95].
- **Optimization techniques**:
  - **Indexing**: Commercial systems often use indexing techniques to accelerate

query execution, such as label-based indexes, property indexes, range indexes and full-text search indexes [96], some systems such as Neo4j support advanced optimization techniques like vector-based embeddings [97].

- **Parallel Processing**: Commercial systems support federated database querying such as the Fabric feature of Neo4j [98] and parallel runtime of Cypher queries.
- **Caching**: Frequently accessed subgraphs or query results may be cached to reduce redundant computations [99].

3 Mapping between Database models

In this section, we categorize prior work on mappings between different database models into three main directions: *from Relations to RDF stores*, *from RDF stores to graphs* and *from relations to graphs*.

3.1 From Relations to RDF stores

Several studies have explored the mapping of relational databases (RDs) to RDF graphs.

Squeda et al. [100] studied the mapping of RDs to RDF graph data and relational schema to OWL ontology's. Moreover, SQL queries are translated into SPARQL queries. They were the first to define a set of mapping properties : information preservation, query preservation, semantic preservation and monotonicity preservation. They proved first that their mapping is information preserving and query preserving, while when it comes to the two remaining properties, preserving semantics makes the mapping not monotonicity preserving, so they show that the two properties semantic preservation and monotonicity are orthogonal.

Das et al. [101] proposed a framework for direct mapping of relational data to RDF, focusing on preserving the structure and semantics of the original database. They introduced a set of rules for automatic transformation, ensuring that the resulting RDF graph maintains the integrity of the relational schema.

Similarly, Bizer et al.[102] developed D2RQ, a mapping language and tool that enables the translation of relational data into RDF while supporting querying via SPARQL. Their approach emphasizes query preservation and scalability for large datasets.

Pham et al. [103] proposed RDB2RDF, a method that transforms all tables in a relational database into an RDF ontology while preserving the relationships between primary and foreign keys. Their approach is fully automated and allows for the reverse transformation of RDF ontology back to relational tables. Unlike other methods, RDB2RDF ensures that the transformation process maintains the semantic constraints of Resource Description Framework Schema (RDFS), such as class and property hierarchies, and supports complex queries.

3.2 From RDF stores to Graphs

Several approaches have been proposed to address the interoperability between RDF and PGs.

Angles et al. [104] studied mapping RDF databases to property graphs by considering both data and schema. They proved that their mapping ensures both information and semantic preservation properties.

Rabbani et al. [105] propose S3PG, a novel technique for transforming RDF graphs into property graphs using standardized schemas, specifically Shapes Constraint Language (*SHACL*) for RDF and PG-Schema for property graphs. S3PG present an approach capable of transforming large knowledge graphs to property graphs while fully preserving information and semantics. Unlike existing methods, which exhibit lossy transformations and lack monotonicity, S3PG achieves 100% accuracy in query answers and supports incremental updates without requiring full recomputation.

G2GML [106] is a language for mapping RDF graphs to PGs, but unlike S3PG, it requires manual specification of mappings and does not provide an automatic transformation algorithm.

Haihong et al. [107] proposed an approach to transform RDF to PGs, focusing on challenges like ensuring node uniqueness and supporting multi-label graphs. However, their method is specific to the Hugegraph system [108] and does not guarantee information preservation.

rdf2neo [109] is a tool for populating Neo4j databases from RDF graphs, but it lacks schema-based transformation and does not ensure completeness.

Zhang et al. [110] introduced a bidirectional mapping method between RDF and PGs, but it does not rely on schema-based approaches to guarantee completeness and

monotonicity.

3.3 From Relations to Graphs

De Virgilio et al. [111, 112] proposed a method for mapping relational databases (RDs) to property graph (PG) data by considering schema information in both the source and target models. Additionally, they developed an approach to translate any SQL query into an equivalent set of graph traversal operations[1], ensuring that the semantics of the original query are preserved over the resulting graph data.

Stoica et al.[113, 114] investigated the transformation of relational databases (RDs) into Graph Databases (GDs), along with the translation of relational queries (formalized as an extension of relational algebra) into equivalent G-core queries.

O.Orel et al. [115] explored the mapping of relational data into property graph, presenting both the conceptual framework and detailed conversion algorithms. Finally, they conducted an experimental study to compare their data models and some other methods.

The Neo4j system offers a dedicated tool called Neo4j-ETL [116], which enables users to import relational data into Neo4j and represent it as property graph. This provides a practical solution for users seeking to migrate relational data into a graph database environment.

S. Li et al. [117] propose an extension of Neo4j-ETL that includes mapping of SQL queries to Cypher queries, enhancing the original tool's functionality.

I. M. Putrama et al. [118] present an automated approach for converting relational databases (RDs) to graph databases (GDs) using relational metadata. Their work includes algorithms for constructing graph databases and introduces a methodology to validate the resulting graph through probabilistic data structures that measure completeness and consistency.

3.3.0.1 Limits of existing approaches (Case of Relational to PGs)

While mapping between RDs and RDF or from RDF to PGs are well studied, the mapping from RDs to PGs is still in its infancy and requires more attention. We show hereafter drawbacks of existing approaches.

[1]Graph traversal queries do not meet neither the syntax of ISO-GQL standard nor Cypher

Table 3: Comparative table of previous mapping solutions.

Data type	Works	Mapping		Mapping properties				
		Schema	Instance	IP	QP	UP	MP	SP
Relations → Graphs	De vergillio et al. [111, 112]	✓	✓	✗	✗	✗	✗	✗
	Stoica et al. [113, 114]	✓	✓	✓	✓	✗	?	?
	O.Orel et al. [115]	✗	✓	✗	✗	✗	✗	✗
	S.Li et al. [117]	✗	✓	✗	✗	✗	✗	✗
	Neo4j-ETL. [116]	✓	✓	✗	✗	✗	✗	✗
	I. M. Putrama et al. [118]	✗	✓	✗	✗	✗	✗	✗

* For the same work, properties with **?** mean that they are controversial.

Table 3 summarizes the most important features of related works when it comes to mapping between RDs and PGs.

De Virgilio et al. [111, 112] approach has several drawbacks. First, the mapping obscures the relational schema, making the resulting graph schema difficult to interpret. Second, it does not account for typed data, limiting its applicability. From a practical standpoint, the graph query language used in their work is not widely adopted, and their query translation heavily depends on its syntax and semantics, making it challenging to adapt to current systems. Furthermore, their method employs an aggregation process that merges different relational tuples into the same graph vertex to optimize traversal operations. While this improves performance, it compromises information preservation and may distort analytical results when applied to the generated graph.

Stoica et al.[113, 114] approach has several shortcomings. First, the choice of source and target languages limits its practical applicability. Additionally, the semantic preservation of the mapping is unclear due to the lack of a formal definition of graph data consistency. Furthermore, the representation of attributes, primary keys, and foreign keys in the resulting graph is overly verbose, leading to reduced readability and increased complexity when querying the data.

O.Orel et al. [115] work has significant shortcomings: it neither addresses schema considerations nor examines mapping properties, resulting in an incomplete approach to relational-to-graph data transformation.

Neo4j-ETL [116] approach has several critical drawbacks:

- **Loss of Relational Structure**: The mapping does not preserve the original relational structure (both instance and schema), as some tuples (or relations in the

schema) are converted into edges for storage optimization, following the method in [112], As noted in [113], this transformation may distort the results of certain analytical tasks, such as graph density calculations.

- **Lack of Query Mapping**: Neo4j-ETL does not support the translation of relational queries into equivalent graph queries, limiting its utility for applications requiring query interoperability.

S. Li et al. [117] approach suffers from several limitations. First, it inherits all the limits of the original Neo4j-ETL system. Also, the authors fail to provide a detailed algorithm for their query mapper, which prevents meaningful comparison with alternative solutions.

I. M. Putrama et al. [118] proposed approach has several notable limitations. First, it focuses exclusively on mapping relational instances while completely omitting the relational schema. The authors neither provide a schema graph definition nor discuss fundamental mapping properties. Additionally, the work lacks any mechanism for query mapping between relational and graph paradigms. The conversion method follows Neo4j-ETL's approach of representing many-to-many relations as edges, which inherits the same limitations. Finally, the absence of formal definitions for the proposed concepts and transformations raises concerns about the rigor and reproducibility of the approach.

In summary, current mapping approaches exhibit one or more of the following shortcomings: (a) they do not satisfy (all) the fundamental properties of mapping; (b) they do not consider mapping of complete relational databases (i.e. data, schema and constraints); (c) mapping of read and write queries is not (well) studied; (d) they rely on theoretical query languages which makes the results hard to apply; and (e) they obfuscate the relational schema. Therefore, no previous work fully satisfies all the requirements for a valid and comprehensive mapping.

4 Distributed Frameworks for Graph Data

Graph data processing has become increasingly important due to the raise of applications in social networks, bio-informatics, recommendation systems, and knowledge graphs. Distributed frameworks are essential for handling large-scale graph data efficiently. These frameworks can be broadly categorized into theoretical solutions (which provide algorithmic foundations) and commercial solutions (which offer scalable,

production-ready implementations).

4.1 Case of simulation semantics

Graph simulation.

Ma et al. [119] propose distributed algorithms for graph simulation on vertex-labeled graphs, focusing on simple queries (without Where clauses). Their approach uses edge-cut partitioning and distributed assembly but is limited to data distribution rather than true queries distributed processing (i.e., this approach is for execution of sequential graph algorithms in a distributed manner).

Wen et al. [120] introduce DRONE, a distributed graph engine based on a native graph model with vertex-cut partitioning and distributed assembly. However, it lacks support for complex pattern conditions (e.g., Where clauses) and multi-environment execution.

Fan et al. [121] present GRAPE, a parallel graph system (later integrated into GraphScope [122]) that parallelizes sequential algorithms with minimal modifications. It supports only simple queries, employs edge-cut partitioning and distributed assembly, but does not address multi-environment execution. A key limitation is data duplication across workers, particularly for small-diameter graphs or complex queries.

4.2 Case of isomorphism semantics

RDF.

Peng et al. [123] propose a partial evaluation and centralized/distributed assembly framework for distributed SPARQL querying over RDF graphs partitioned using the edge-Cut strategy. Similarly, Song et al. [124] and Peng et al. [125] present distributed SPARQL query frameworks based on Edge-Cut partitioning and centralized assembly, handling patterns with conditions.

PG.

Trigonakis et al. [126] propose aDFS, a distributed graph-querying system for pattern-matching queries on large-scale property graphs using Property Graph Query Language (*PGQL*) [9]. The system employs edge-cut partitioning, distributed assembly, and considers where conditions in queries, with a formal workflow-based approach. However, it lacks support for multi-environment execution.

Hao et al. [127] present a distributed, Cypher-compatible pattern-matching system over property graphs, aligning academic optimizations with industrial needs. Similar

to [126], it uses edge-cut partitioning and distributed assembly but omits formal algorithms and multi-environment support.
Min Wu et al. [128] introduce Nebula, a distributed open-source graph database for massive-scale data, supporting property graphs and multiple query languages (OpenCypher/NebulaGraph Query Language (*nGQL*)). It relies on edge-cut partitioning and distributed assembly, but its query capabilities are limited to short queries with basic WHERE conditions, lacking algorithmic details and multi-environment execution.

4.2.0.1 Limits of existing approaches

Table 4 compares between the most important approaches in the case of graph isomorphism.

Pattern Matching	Works	Graph model	Query language	Partitioning strategy	Assembly	*WHERE* Conditions	Formalism	System-independent
Graph Isomorphism	[123]	RDF	SPARQL	Edge-cut	Centralized & Distributed	Yes	Yes	Yes
	[124, 125]	RDF	SPARQL	Edge-cut	Centralized	Yes	Yes	Yes
	[126]	PG	PGQL	Edge-cut	Distributed	Yes	No	No
	[127]	PG	Cypher	Edge-cut	Distributed	Yes	No	No
	[128]	PG	Cypher/nGQL	Edge-cut	Distributed	Yes	No (open-source)	No

Table 4: Comparative table of previous distributed querying solutions.

We notice that the graph simulation semantics is not supported by commercial graph database systems (e.g., Neo4j, MemGraph, Oracle), which makes the dedicated approaches [119, 120, 122, 129] not really applicable in practice. Approaches based on subgraph isomorphism are more interesting as this semantics is supported by existing graph database systems. They use different strategies for data partitioning (i.e., *edge-cut* or *vertex-cut*) and data assembly (i.e., *centralized* or *distributed*). After a deep analysis of these approaches, we noticed three major limits: 1) some distributed *GPM* solutions are presented as a black box with no formalization that allows their application in another context or their comparison with another approach; 2) some solutions are presented as a complete system that must be completely implemented to leverage the advantage of the distribution, while there are actually many advanced graph systems that one would like to use with no additional fees; 3) some solutions are based on an SQL-like query language dedicated to graph data (e.g., PGQL), while the Cypher query language is now considered as the most closely language to the ISO GQL standard. We hope advancing research in this area by proposing a distributed GPM framework that is based on a solid formalism, considers the syntax of the ISO-GQL, and seamless integrates into real life systems without migrating costs.

Chapter II:
Mapping Relations to Graphs

1 Introduction

For many decades, relational databases have been extensively studied by researchers and widely used by practitioners. The relational model gained its popularity [130] due to a straightforward design, a high data accuracy, a minimal redundancy, and the presence of a standard query language (SQL). The advent of Big Data has been marked by an explosion of Web data in terms of volume and interconnections. For instance, social networks (e.g. Facebook, Youtube) contain billions of entities [131] (e.g. Persons, Videos) which are interconnected with each other. Faced with this kind of data, the relational model has reached its limits in the sense that relational tables do not provide a clear view of the relationships that exist between the data. Furthermore, querying this data requires SQL queries that are often complex and time-consuming [132]. This has led researchers to adopt a new model for Web data. Indeed, the graph database model has received more attention during the last decade. Actually, graph database systems (e.g. Neo4j [133], JanusGraph [134], ArangoDB [20] and TigerGraph [135]), are widely used in many real-life applications, e.g. social network analysis, fraud detection, recommendation engines. Graph database model allows natural presentation of complex data where interconnections can be visualized more easily and clearly than in the relational model. In addition, several graph algorithms can be used to achieve efficient analytical tasks over graph database [4]. A real-life example has been discussed in [113]: investigative journalists have recently found, via the graph database model, surprising social relationships between executives of companies within the Offshore Leaks financial social network, linking company officers and their companies registered in the Bahamas. They first mapped big relational database into graph database and then executed some analytical operations over this latter. This case-study suggested to investigate mapping of existing relational database into graph database. As this thematic is still new, only a few works [111, 112, 113, 114] have considered it. The existing mapping models suffer from one or more of the following limitations: (a) they do not satisfy (all) the fundamental properties of mapping; (b) they do not consider mapping of complete relational databases (i.e. data, schema and constraints); (c) mapping of read and write queries is not (well) studied; (d) they rely on theoretical query languages which makes the results hard to apply; (e) they obfuscate the relational schema. For instance, our study of the Neo4j mapping tool, called *Neo4j-ETL* [116], showed that it does not preserve the same structure as the relational database, and in addition, it takes a long time to produce the graph database.

2 Preliminaries

In this section, we define the several concepts that will be used throughout this chapter.

Let consider $\mathcal{R}$ as an infinite set of relation names, $\mathcal{A}$ as an infinite set of attribute names with a special attribute *tid*, $\mathcal{T}$ as a finite set of attribute types (e.g. *Varchar*, *Date*, *Integer*, *Float*, *Boolean*, *Char*, *Object*), and $\mathcal{D}$ as a countably infinite domain of data values with a special value NULL.

2.1 Relational Databases

The next definition extends the one provided in [136] by considering advanced constraints.

A *relational database* is composed of a *relational schema* and a *relational instance*. We define these two notions below. However, it is essential to first introduce a fundamental concept within relational schemas, namely the notion of *keys*.

Keys are fundamental elements in relational databases that serve to uniquely identify tuples (rows) in relations. There exists a hierarchy of keys (*super keys*, *candidate keys*, *primary keys*, *composite keys* and *alternate keys*) as is mentioned in [137, 138, 139, 140, 141, 142]. We summarize hereafter these concepts:

- A *super key* is the set of one or more attributes that uniquely identifies a tuple in a relation r. A *super key* may contain additional attributes that are not strictly necessary for unique identification.
- A *candidate key* is a minimal *super key*, meaning it is a subset of a *super key* with no redundant attributes. It is a set of attributes that uniquely identifies a tuple within a relation r, and no proper subset of it can uniquely identify a tuple.
- Among *candidate keys* of a relation r, one must be selected as the *primary key*. Unlike *primary keys*, which cannot contain NULL values, the attributes of a *candidate key* can include NULL values.
- If *primary key* in a relation r consists of multiple attributes (i.e., $n > 1$), it is referred to as *composite key* [139, 141, 142].
- Any remaining *candidate keys* that are not chosen as *primary key* are designated as *alternate keys*.

- A *foreign Key* is the set of one (resp. Many) attributes of a relation s that refers to the *primary* (resp. *composite*) key of another relation r.

It is very important to understand the difference and equivalence between all these keys when designing a database. For the sake of clarity, we consider only *composite keys* in our mapping while the remaining keys can be mapped in a similar way. In addition, *composite keys* are the most used in practice by commercial database systems which makes easy integration of our mapping within a real system.

Our formal definition of *relational schema* given below considers the notion of *composite key*, and consequently, covers also the notion of *primary key* as it is just a special case of the former. Notice that the two notions of *composite key* and *primary key* are used interchangeably both in industry and some literature. For the sake of clarity, we use *primary key* to refer to both notions.

A ***relational schema*** is a tuple $S = (R, A, T, \Sigma, N, C, U, D)$ where:

- R is a finite set of relation names, where $R \subseteq \mathcal{R}$.
- A is a function that assigns a finite set of attributes for each relation $r \in R$ such that $A(r) \subseteq \mathcal{A} \backslash \{tid\}$;
- T is a function that assigns a type for each attribute of a relation, i.e. for each $r \in R$ and each $a \in A(r) \setminus \{tid\}$, $T(r, a) \in \mathcal{T}$;
- Σ is a finite set of *primary* keys and *foreign* keys defined over R and A. A *primary key* over a relation $r \in R$ is an expression of the form $r[a_1, \cdots, a_n]$ where $a_{1 \leq i \leq n} \in A(r)$.[1] A *foreign key* over two relations $r \in R$ and $s \in R$ is an expression of the form $r[a_1, \cdots, a_n] \rightarrow s[b_1, \cdots, b_n]$ where $a_{1 \leq i \leq n} \in A(r)$ and $s[b_1, \cdots, b_n] \in \Sigma$;
- N is a function defined over R such that: for each $r \in R$, $N(r) \subseteq A(r)$ specifies attributes of r whose values cannot be NULL;
- C is a partial function defined over R such that, for each $r \in R$, $C(r)$ is a boolean expression that is defined over some attributes of r and must be satisfied by the values of these attributes;
- U is a function defined over R such that: for each $r \in R$, $U(r) \subseteq A(r)$ specifies attributes of r whose values are *unique*;

[1] This definition covers *primary* key (with $n = 1$) and *composite* key (with $n > 1$) in a relation r.

- D is a partial function defined over R and A such that: for each $r \in R$ and each $a \in A(r)$, $D(r, a) \in \mathcal{D}$ is the *default* value of attribute a;

Intuitively, R, A and T specify the database model, i.e. relations composing the database, attributes of these relations, and types of these attributes. The functions Σ, N, C, and U specify the database constraints, i.e. conditions that must be satisfied by attributes of some relations. Precisely, we consider the following well-used constraints: *integrity* constraints (Σ), *not null* constraints (N), *check* constraints (C), and *unique* constraints (U). The function C specifies a condition that must be satisfied by some attributes of a relation r. For instance, let's consider a relation r with the name Product, and attributes a_1 and a_2 specifying the purchase price and sale price of the product, respectively. It is clear that the sale price must be greater than the purchase price, and both prices must be different from 0. That is a *check* constraint, It can be defined over the relation r with: $C(r) = a_1 \neq 0 \wedge a_2 \neq 0 \wedge a_2 > a_1$.

In addition to these constraints, the function D specifies default values for some attributes in the sense that if these attributes are not given any value during data insertion, so they take their values from D. Notice that D does not specify constraints neither from theoretical nor industrial point of view. However, its integration in our schema graph allows dealing with missing values which cannot be replaced by NULL. In other words, graph database systems (such as Neo4j) remove attributes containing NULL values which may change semantics/structure of the data, hence, default values allow keeping these missing data in the graph for more consistency.

An ***instance*** I of S is presented by assigning to each relation $r \in R$ a finite set $I(r) = \{t_1, \cdots, t_n\}$ of *tuples*. Each *tuple* t_i is represented by a function $A(r) \cup \{tid\} \rightarrow \mathcal{D}$ assigning a value to each attribute $a \in A(r) \cup \{tid\}$. We use $t_i(a)$ to refer to the value of attribute a in tuple t_i. Particularly, $t_i(tid)$ is an integer that refers to the identifier of tuple t_i. Indeed, for any tuples $t_i, t_j \in I(r)$, $t_i(tid) \neq t_j(tid)$ if $i \neq j$.

For any instance I of a relational schema $S = (R, A, T, \Sigma, N, C, U, D)$, we show hereafter when I satisfies constraints of S.

1. I satisfies *typing* constraints if: for any relation $r \in R$ and any attribute $a \in A(r)$, ***type***$(t(a))$=$T(r, a)$.

2. I satisfies a primary key $r[a_1, \cdots, a_n]$ in Σ if: 1) for each tuple $t \in I(r)$, $t(a_{1 \leq i \leq n}) \neq$ NULL; and 2) for any $t' \in I(r)$, if $t(a_{1 \leq i \leq n}) = t'(a_{1 \leq i \leq n})$ then $t = t'$ must hold.

Table 5: Comparative table of SQL clauses and Cypher clauses.

Type of data manipulation	SQL Clauses	Cypher Clauses
Querying data	**SELECT / FROM :** is used to select one/or more relations, to optionally join them together, and to return attribute values of some of these relations.	**MATCH / RETURN :** is used to retrieve all sub-graphs of some property graph that have some pattern and to return attribute values of some vertices and edges belonging to these sub-graphs.
Updating data	**INSERT INTO :** is used to add new tuples to a relation instance. **UPDATE / SET :** is used to modify attribute values of tuples. **DELETE :** is used to delete tuples from a relational instance.	**CREATE :** is used to add vertices and edges to a property graph. **MATCH / SET :** is used to modify attribute values of vertices and edges of a property graph. **DELETE :** is used to delete vertices and edges from a property graph. **DETACH DELETE :** is used to remove a vertex from a graph database, along with all its incoming and out-coming edges.
Filtering data	**WHERE :** is used to specify conditions over tuples that have to be returned, updated or deleted.	**WHERE :** is used to specify conditions over vertices and edges that have to be returned, updated or deleted.

3. I satisfies a foreign key $r[a_1, \cdots, a_n] \rightarrow s[b_1, \cdots, b_n]$ in Σ if: 1) I satisfies $s[b_1, \cdots, b_n]$; and 2) for each tuple $t \in I(r)$, either $t(a_{1 \leq i \leq n}) = \text{NULL}$ or there exists a tuple $t^{'} \in I(s)$ where $t(a_{1 \leq i \leq n}) = t^{'}(b_{1 \leq i \leq n})$.

4. I satisfies all integrity constraints in Σ, denoted by $I \vDash \Sigma$, if it satisfies all primary keys and foreign keys in Σ.

5. I satisfies *not null* constraints if: for each tuple $t \in I(r)$ and each attribute $a \in N(r)$, $t(a) \neq \text{NULL}$.

6. I satisfies *check* constraints if: for each tuple $t \in I(r)$, the values assigned by t to attributes of r satisfy the boolean expression given by $C(r)$.

7. I satisfies *unique* constraints if: for any relation $r \in R$ and any attribute $a \in U(r)$, there exists no pair of tuples $t_1, t_2 \in I(r)$ with $t_1 \neq t_2$ and $t_1(a) = t_2(a)$.

We say that I is a ***consistent instance*** of S if it satisfies all constraints of S, denoted by $I \models S_R$.

Finally, a *relational database* is defined with $D_R = (S_R, I_R)$ where S_R is a relational schema and I_R is an instance of S_R.

Intuitively, the label of a vertex represents an entity (e.g. Person, Product) while the label of an edge represents a relationship between two entities (e.g. an edge between entities Person and Product may have the label Invented to specify who invented what).

In real-life graph databases, vertices have attributes that specify their properties (e.g. attributes name and age for a vertex Person). We use two functions to formalize attributes: A^a is a function that assigns attributes to vertices of the property graph; while A^c is a function that assigns content to each of these attributes. Notice that edges may have attributes in real-life applications; however, our mapping approach yields edges without attributes.

For any edge $e \in E_S$, we denote by $e.s$ (resp. $e.d$) the starting (resp. ending) vertex of e. A property graph is a multi-graph as, for the same pair of vertices, there may be several edges between them that express different relationships.

We defined *property graph* as a general structure that will be used later to define two different notions: *schema* and *instance*. Indeed, each attribute of a vertex has a *content* which can be either a type (case of *schema*) or a value (case of *instance*). Thus, the function A^c returns values that belong either to $\mathcal{T}$ (i.e. types) or $\mathcal{V}$ (i.e. values).

2.2 SQL and Cypher

This chapter studies the mapping of data, constraints and queries from the relational model to the graph model. We consider the SQL language [143] to model relational queries and the Cypher query language [10] to model graph queries. Notice that each of these languages is the most used in its category. We consider a very practical class of SQL queries and define its corresponding class of Cypher queries. It is necessary to understand the relations between basic SQL queries and basic Cypher queries before studying these languages' expressive power.

Cypher was first designed and developed as part of the Neo4j graph database system, but other commercial products use it. It is the most used language for querying data modeled as a graph. The Cypher query language has been inspired by SQL, hence, each SQL feature (modeled over relations) has an equivalent feature (modeled over graphs) in Cypher. Table 5 compares the most used SQL clauses and their equivalent Cypher clauses.

The following example clarifies some equivalence between SQL clauses and Cypher clauses.

Example 8. *Hereafter are examples of some SQL and Cypher queries grouped according to the type of data manipulation they allow.*

Querying data:

SQL:

```
SELECT t.name, c.Title FROM Teachers AS t, Courses AS c
WHERE t.id = c.id AND c.Title = 'Neo4j';
```

Cypher:

```
MATCH (t:Teachers)-[r:Has]->(c:Course)  WHERE c.Title='Neo4j
RETURN t.name, c.title;
```

Inserting data:

SQL:

```
INSERT INTO Teachers (name, age)  VALUES ('John', 36);
```

Cypher:

```
CREATE(t:Teachers{name:'John',age: 36});
```

Updating data:

SQL:

```
UPDATE Teachers SET age = 55  WHERE  name = 'John';
```

Cypher:

```
MATCH (t:Teachers { name: 'John' })
SET t.age = 55;
```

Deleting data:

SQL:

```
DELETE FROM Teachers WHERE name = 'John';
```

Cypher:

```
MATCH (t:Teachers { name: 'John' }) DELETE t;
```

□

3 Direct Mapping (DM)

It is worth noting that the definition of *direct mapping* as well as its properties are a little bit standards as many previous works have used the same definitions. That is

why we report hereafter the same definitions given in [136].

We first give the definition of *direct mapping* from relational databases to property graphs, and then we recall its fundamental properties [100, 113, 114].

Given a relational database D_R=(S_R, I_R) with possibly $S_R = \emptyset$. A Direct Mapping (DM) consists in translating D_R into a pair of property graphs (S_G, I_G), with possibly $S_G = \emptyset$, that we call a graph database. The mapping is called *direct* since no user interaction is necessary. Assume that $\mathcal{D}_R$ is an infinite set of relational databases and $\mathcal{D}_G$ is an infinite set of graph databases. That is:

Definition 8. *A* direct mapping *is a total function* $DM : \mathcal{D}_R \rightarrow \mathcal{D}_G$. □

In a straightforward sense, for each $D_R \in \mathcal{D}_R$, $DM(D_R)$ generates a graph database $D_G \in \mathcal{D}_G$ that aims to represent the source relational database (i.e. instance and optional schema) in terms of graphs.

As underlined in [100], a direct mapping must preserve some properties: Information Preservation (IP), Semantic Preservation (SP), Query Preservation (QP) and Monotonicity Preservation (MP). The *information preservation* and the *semantic preservation* ensure that the direct mapping does not lose neither information nor semantic of the relational database being translated. The *query preservation* ensures that the mapping does not impede the querying capabilities as any relational query can be translated into a graph query. Finally, the *monotonicity* ensures that any updates on the relational database D_R do not require re-computing $DM(D_R)$ from scratch. We give hereafter formal definitions of these mapping properties.

Definition 9 (Information preservation)**.** *A direct mapping* DM *is* information preserving *if there is a computable inverse mapping* $DM^{-1} : \mathcal{D}_G \rightarrow \mathcal{D}_R$ *satisfying* $DM^{-1}(DM(D_R)) = D_R$ *for any* $D_R \in \mathcal{D}_R$. □

A direct mapping DM is *information preserving* if its inverse process gives the original relational database, i.e., each information in D_R is correctly mapped in terms of a graph.

Definition 10 (Semantic preservation)**.** *A direct mapping* DM *is semantic preserving if for any relational database* $D_R = (S_R, I_R)$*,* $I_R \models S_R$ *iff:* $DM(D_R)$ *produces a consistent graph database.* □

A direct mapping *DM* is *semantics preserving* if any consistent (resp. inconsistent) relational database is mapped into a consistent (resp. inconsistent) graph database. Notice that no previous work has provided a formal definition of *graph consistency*. We shall later give our own definition to ensure *semantic preservation*.

To discuss the *query preservation* property, we must first understand the execution semantics of both SQL and Cypher. Any query of these languages returns result modeled as table where columns represent entities requested by the query (i.e. relational attributes, vertices, edges and graph attributes), while each row of the table result assigns values to these entities. In addition, there may be rows containing the same result if this one can be computed in different manners.

Suppose we have I_R, which represents a relational instance and Q_s be an SQL query defined over I_R. We denote by $[Q_s]_{I_R}$ the result obtained by evaluating Q_s over I_R. Similarly, $[Q_c]_{I_G}$ is the result obtained by evaluating a Cypher query Q_c over a graph instance I_G. Moreover, we denote by $[Q_s]^*_{I_R}$ (resp. $[Q_c]^*_{I_G}$) a refined version of $[Q_s]_{I_R}$ (resp. $[Q_c]_{I_G}$) by omitting repeated rows. A direct mapping *DM* is *query preserving* if any query Q_s over the relational database D_R can be translated into an equivalent query Q_c over the graph database D_G that results from the mapping of D_R. Equivalence between two table results has been studied first in [114] between relational queries and RDF queries. Given the result of the SQL query, $[Q_s]^*_{I_R}$, and that of the Cypher query $[Q_c]^*_{I_G}$, the goal is to show that each row in the former can be mapped to a row in the latter and vice versa. Therefore, we modify the definition of *query preservation* as follows:

Definition 11 (Query preservation)**.** *A direct mapping* DM *is* query preserving *if for any relational database* D_R=(S_R,I_R) *and any SQL query* Q_s *over* I_R*, there exists a Cypher query* Q_c *such that: each row in* $[Q_s]^*_{I_R}$ *can be mapped into a row in* $[Q_c]^*_{I_G \in DM(D_R)}$ *and vice versa.* □

This property ensures that, when mapping data from relational context to graph context, the expressive power given to the users by SQL is preserved by Cypher.

We discuss next the last mapping property, *monotonicity preservation*. Given a relational schema S_R and two instances I_R^1 and I_R^2 of S_R. Obviously, we say that I_R^1 is contained in I_R^2, written $I_R^1 \subseteq I_R^2$, if all tuples of the former are contained in the latter. As defined in [100], a direct mapping DM is said *monotone* if the instance graph resulted from the mapping of I_R^1 is contained in that resulted from the mapping of I_R^2.

Definition 12 (Monotonicity)**.** *A direct mapping* DM *is* monotone *if for every relational schema S_R and any relational instances I_R^1 and I_R^2 of S_R with $I_R^1 \subseteq I_R^2$: $I_G^1 \subseteq I_G^2$ with $I_G^1 \in DM(S_R, I_R^1)$ and $I_G^2 \in DM(S_R, I_R^2)$.* □

This property allows incrementally maintaining the property graphs obtained by some mapping process when the original relational data is updated. In other words, if I_R^1 has already been mapped to I_G^1, then for any relational update U_R on I_R^1 that yields to I_R^2, one can found the equivalent update U_G on I_G^1 that yields to I_G^2, without mapping I_R^2 into I_G^2 from scratch.

4 Complete Mapping (*CM*)

Notice that the formalisation of complete mapping is different to the one provided in [136] as this chapter considers more advanced constraints. Precisely, the definitions of schema graph, instance graph, schema mapping and instance mapping are thoroughly revised.

We discuss in this section a Complete Mapping (CM) that converts a complete relational database (schema and instance) into a complete graph database (schema and instance). We call our mapping *Complete* since some proposed mappings (e.g [115]) deal only with data and not schema.

Definition 13 (*Complete Mapping*)**.** *A complete mapping is a total function $CM : \mathcal{D}_R \to \mathcal{D}_G$ such that : for each relational database $D_R = (S_R, I_R)$, $CM(DR)$ generates a graph database $D_G = (S_G, I_G)$.* □

To complete the definition of our *CM* process, we need first to provide a definition of *graph database*. Next, we must detail the two steps of our mapping: *schema mapping* and *instance mapping*.

4.1 Graph Databases

Notice that the definition of graph database [136] has undergone significant updates to accommodate the newly added integrity constraints.

Contrary to relational database, there exists yet no standard for schema graphs, and no formal definition has been proposed to capture all features of relational schema. Hence, we propose a definition of schema graph as an extension of the property graph. Our definition allows us to express all relational constraints defined above (i.e. integrity constraints, along with not null, unique and check constraints).

Definition 14 (*Schema Graph*)**.** *A* schema graph S_G *is given by* $(V_S, E_S, L_S, A^a_S, A^c_S, Pk, Fk, N_S, C_S, U_S, D_S)$ *where:*

1. $(V_S, E_S, L_S, A^a_S, A^c_S)$ *is a property graph such that for each* $v \in V_S$ *and each attribute* $a \in A^a_S(v)$, $A^c_S(v, a) \in \mathcal{T}$. *That is,* A^c_S *assigns types to vertex attributes;*
2. Pk *is a partial function, specifying* primary keys *over vertices, such that* $Pk(v) \subseteq A^a_S(v)$ *for any vertex* $v \in V_S$*;*
3. Fk *is a function, specifying* foreign keys *over edges, such that for each edge* $e \in E_S$*,* $Fk(e, s) \subseteq A^a_S(e.s)$ *(resp.* $Fk(e, d) \subseteq A^a_S(e.d)$*);*
4. N_S *is a partial function, specifying attributes whose values cannot be* NULL*, defined by* $N_S(v) \subseteq A^a_S(v)$ *for any vertex* $v \in V_S$*;*
5. C_S *is a partial function, specifying* check *constraints over vertices, such that: for any vertex* $v \in V_S$*,* $C_S(v)$ *is a boolean expression defined over* $A^a_S(v)$*;*
6. U_S *is a partial function, specifying attributes whose values are* unique*, defined by* $U_S(v) \subseteq A^a_S(v)$ *for any vertex* $v \in V_S$*;*
7. D_S *is a partial function, specifying* default values *of some attributes, defined by* $D_S(v, a) \in \mathcal{V}$ *for any vertex* $v \in V_S$ *and any attribute* $a \in A^a_S(v)$*;* □

Similarly to relational schema, a schema graph S_G is composed of two parts: (1)

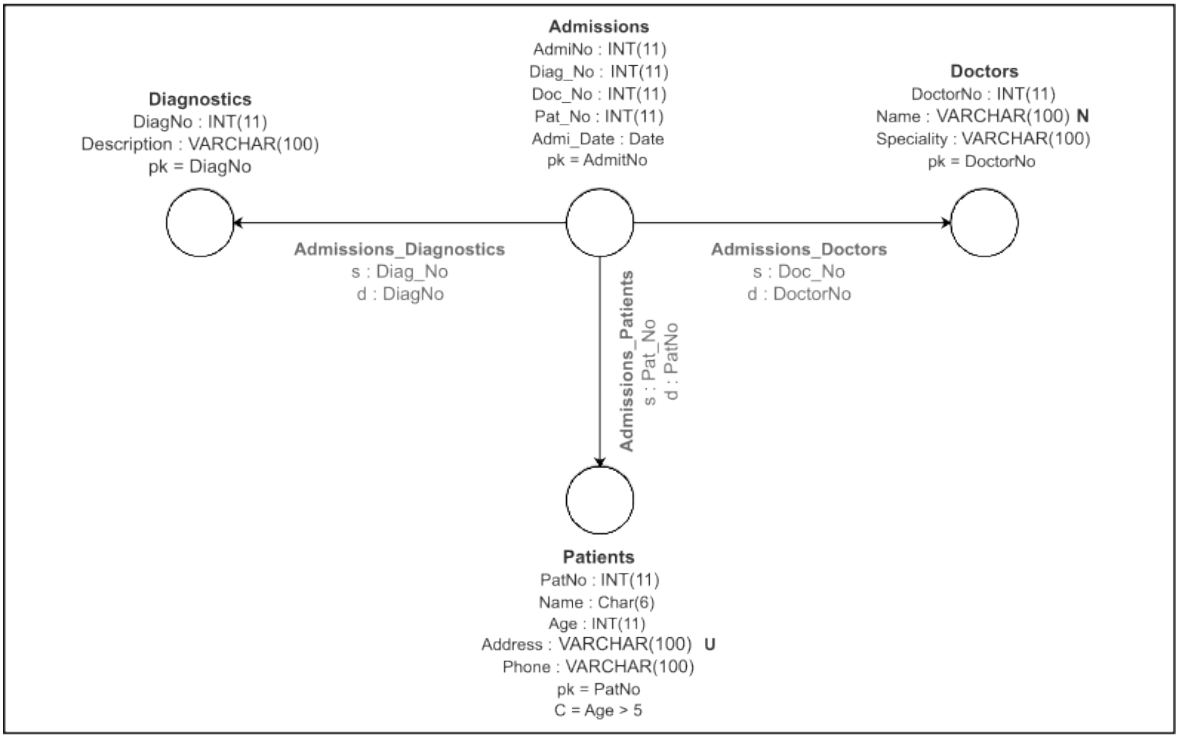


Figure 6: Example of schema graph.

the database model that is characterized by the property graph $(V_S, E_S, L_S, A^a_S, A^c_S)$; and (2) the constraints model that is given by $(P_k, F_k, N_S, C_S, U_S, D_S)$. The database model of S_G specifies vertices and edges that compose the graph database, possible attributes these vertices may have, as well as, types of values that can be assigned to these attributes. The constraints model of S_G allows to express all constraints of its relational counterpart.

We next define the notion of *instance graph* that will be used to represent relational instances in terms of graphs.

Definition 15 (*Instance Graph*)**.** *Given a schema graph* $S_G = (V_S, E_S, L_S, A^a_S, A^c_S, Pk, Fk, N_S, C_S, U_S, D_S)$, *an* instance graph I_G *of* S_G *is given by a property graph* $I_G = (V_I, E_I, L_I, A^a_I, A^c_I)$ *that follows the database model of* S_G *as follows:*

1. *for each vertex* $v_i \in V_I$ *and each attribute* $a \in A^a_I(v_i)$: $A^c_I(v_i) \in \mathcal{V}$;
2. *for each vertex* $v_i \in V_I$, *there exists a vertex* $v_s \in V_S$ *such that: a)* $L_I(v_i) = L_S(v_s)$; *b)* $A^a_I(v_i) \subseteq A^a_S(v_s)$; *and c) for each attribute* $a \in A^a_I(v_i)$, ***type***$(A^c_I(v_i))$=$A^c_S(v_s)$. *We say that* v_i *corresponds to* v_s, *denoted by* $v_i \sim v_s$.

3. *for each edge* $e_i = (v_i, w_i)$ *in* E_I*, there exists an edge* $e_s = (v_s, w_s)$ *in* E_S *such that: a)* $L_I(e_i) = L_S(e_s)$*; b)* $v_i \sim v_s$*; and c)* $w_i \sim w_s$*. We say that* e_i *corresponds to* e_s*, denoted by* $e_i \sim e_s$*.* □

Recall that the database model of a schema graph S_G assigns attributes to vertices (via the function A_S^a), and moreover, specifies the content of these attributes as types (via the function A_S^c). An instance graph I_G satisfies the database model of S_G by assigning a value to each vertex attribute a provided that the type of this value respects the type required by S_G for a (via the function A_S^c).

I_G is an instance of S_G if each vertex (resp. edge) in the former corresponds to a vertex (resp. edge) in the latter. A vertex $v_i \in V_I$ corresponds to a vertex $v_s \in V_S$ if they have the same label, and for any attribute a attached to v_i with a value c, there exists an attribute a that is attached to v_s with the type of c as content. This means that v_i is not supposed to have all attributes attached to v_s. This is helpful when modeling data with NULL values. Moreover, an edge $e_i \in E_I$ corresponds to an edge $e_s \in E_S$ if they have the same label, and the starting (resp. ending) vertex of e_i corresponds to the starting (resp. ending) vertex of e_s.

Example 9. *Figure 6 depicts an example of a schema graph defined with four vertices that represent entities (i.e.* Diagnostics*,* Admissions*,* Doctors*, and* Patients*) and three edges that represent relationships between these entities (i.e.* Admissions_Diagnostics*,* Admissions_Doctors*, and* Admissions_Patients*). Attributes of this schema graph are described below:*

- *A* Diagnostics *vertex is characterized by a* Diagnostic Number *(*`DiagNo`*) and a* `Description`*.*
- *An* Admissions *vertex is characterized by an* Admission Number *(*`AdmiNo`*), a* Diagnostic Number *(*`Diag_No`*), a* Doctor Number *(*`Doc_No`*), a* Patient Number *(*`Pat_No`*), and an* Admission Date *(*`Admi_Date`*).*
- *A* Doctors *vertex is characterized by a* Doctor Number *(*`DoctorNo`*), a* `Name`*, and a* `Speciality`*.*
- *A* Patients *vertex is characterized by a* Patient Number *(*`PatNo`*), a* `Name`*, an* `Age`*, and an* `Address`*.*

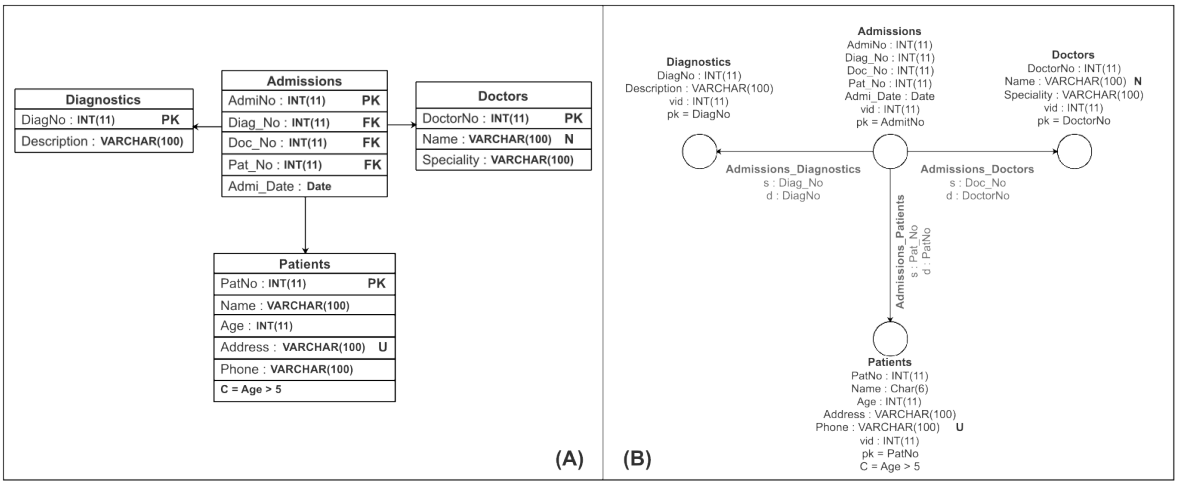


Figure 7: Example of schema mapping.

- *Each edge in our approach will help specifying a join between two entities, and its attributes will associate each foreign key with a primary key. For instance, an* Admissions_Diagnostics *edge specifies a join between* Admissions *and* Diagnostics *entities, hence it has two attributes,* `s` *and* `d`*, clarifying respectively the foreign key* `Diag_No` *of* Admissions *and the primary key* `DiagNo` *of* Diagnostics*.*

A special attribute pk is added to each vertex of the schema graph in order to model the semantics of primary key*. In addition, the special attribute C refers to a constraint defined over attribute values of the vertex (i.e. entity).*

It can be observed that each vertex (resp. edge) is naturally represented with its label (e.g. the vertex Admissions *and the edge* Admissions_Doctors*) and a list of typed attributes (e.g.* `AdmiNo:INT(11)` *refer to Integer type with size* 11*). For each vertex, the value of the function Pk is given (e.g.* Pk=`AdmiNo` *on vertex* Admissions*). Moreover, for each edge e, the values* $Fk(s)$ *and* $Fk(d)$ *correspond to the values of the functions* $Fk(e, s)$ *and* $Fk(e, d)$ *respectively (e.g.* $Fk(s)$=`Doc_No` *and* $Fk(d)$=`DoctorNo` *on the edge* Admissions_Doctors*).* □

Finally, a *graph database* is defined with $D_G = (S_G, I_G)$ where S_G is a schema graph and I_G is an instance of S_G. Remark that this definition considers only the database model of S_G. The constraints model of S_G will be used later to define the notion of *consistent instance graph*.

Now that the notions of *relational database* and *graph database* are well defined, we next show how to establish a mapping from the former to the latter.

4.2 Schema and Constraints Mapping (*SM*)

Given a relational schema $S_R = (R, A, T, \Sigma, N, C, U, D)$, we propose a Schema Mapping (SM) process that produces a schema graph $S_G = (V_S, E_S, L_S, A_S^a, A_S^c, Pk, Fk, N_S, C_S, U_S, D_S)$ as follows:

1. For each relation name $r \in R$, there exists a vertex $v_r \in V_S$ with: $L_S(v_r) = r$;
2. For each attribute $a \in A(r)$ with $T(a) = t$, we have: $a \in A_S^a(v_r)$ and $A_S^c(v_r, a) = t$;
3. For each *primary key* $r[a_1, \cdots, a_n] \in \Sigma$, we have: $Pk(v_r) = \{a_1, \cdots, a_n\}$;
4. For each *foreign key* $r[a_1, \cdots, a_n] \rightarrow s[b_1, \cdots, b_n] \in \Sigma$, we have an edge $e = (v_r, v_s) \in E_S$ such that: (a) $L_S(e) = s\_r$; (b) $Fk(e, s) = \{a_1, \cdots, a_n\}$; and (c) $Fk(e, d) = \{b_1, \cdots, b_n\}$;
5. For each relation $r \in R$, we have: $N_S(v_r) = N(r)$;
6. For each relation $r \in R$, we have: $C_S(v_r)$ is a rewriting of $C(r)$ in Cypher syntax[1];
7. For each relation $r \in R$, we have: $U_S(v_r) = U(r)$;
8. For each relation $r \in R$ and each attribute $a \in A(r)$ playing a role in D, we have: $D_S(v_r, a) = D(r, a)$;
9. Particularly, for each vertex $v_r \in V_S$, we have: $vid \in A_S^a(v_r)$ and $A_S^c(v_r, vid) =$ Integer. This special attribute is used only for storage concerns (i.e., equivalent of *tid* in a relational context).

Example 10. *Figure 7 depicts (A) a relational schema and (B) its corresponding schema graph. We use special notations over the relational schema to express constraints. Precisely: (1) the notation Pk (resp. Fk) behind an attribute specifies whether this one plays a role in a primary (resp. foreign) key; (2) N (resp. U) specifies that the attribute has a* not null *(resp.* unique*) constraint; and (3) a boolean expression at the end of a relation table (e.g.* Age≥ 5 *on relation* Patients*) refers to a* check *constraint.*

*One can check that applying our schema mapping process (*SM*) over this relational schema produces the schema graph at the right side of the figure. It is evident that our*

[1] $C_S(v_r)$ and $C(r)$ are equivalent boolean expressions except the fact that $C(r)$ is expressed over attributes of r according to SQL syntax, while $C_S(v_r)$ expresses the same semantics over attributes of v_r using the Cypher syntax.

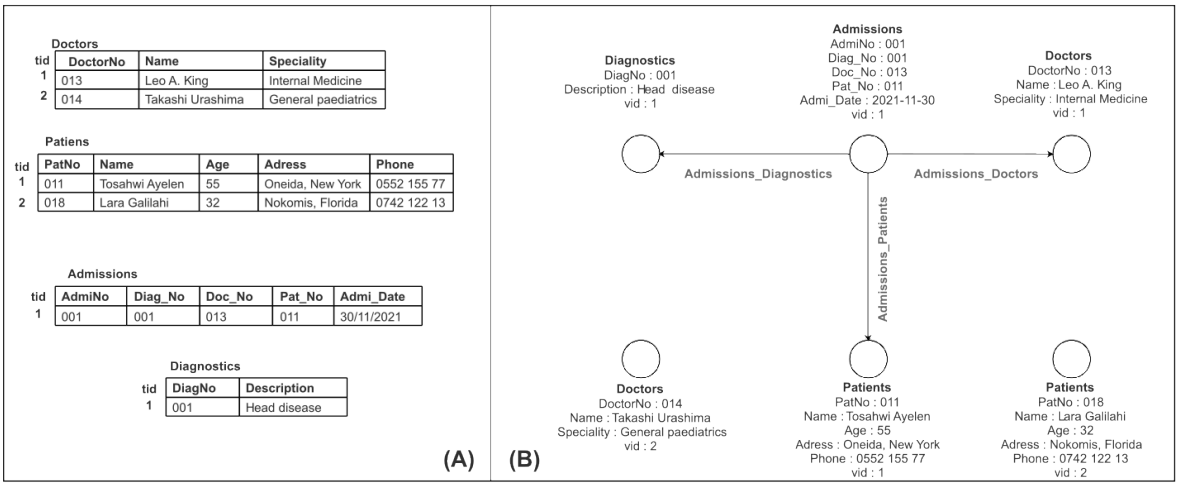


Figure 8: Example of instance mapping.

schema mapping rules are respected: (1) each relation is represented by a vertex that includes the relation's label, its primary key(s), and a list of its typed attributes; (2) each foreign key that links two relations (e.g. relations Admissions *and* Patients *in part side (A)) is mapped to an edge between the vertices of these two relations (e.g. edge* Admissions_Patients *in part side (B)).* □

4.3 Instance Mapping (*IM*)

Given a relational database $D_R = (S_R, I_R)$, we introduce the Instance Mapping (IM) process which converts the relational instance I_R into an equivalent instance graph $I_G = (V_I, E_I, L_I, A_I^a, A_I^c)$ as follows :

1. For each tuple $t \in I(r)$, there exists a corresponding vertex $v_t \in V_I$ with $L_I(v_t) = r$. The vertex representing the tuple t is denoted by v_t;
2. For each tuple $t \in I(r)$ and each attribute a with $t(a) = c$ and $a \neq tid$, we have: $a \in A_I^a(v_t)$, $A_I^c(v_t, a) = c$. As a special case, the attribute tid of t is expressed over v_t as follows: $vid \in A_I^a(v_t)$ and $A_I^c(v_t, vid) = t(tid)$;
3. for each *foreign key* $r[a_1, \cdots, a_n] \rightarrow s[b_1, \cdots, b_n]$ defined in Σ and any tuples $t \in I(r)$ and $t' \in I(s)$, if $t(a_{1 \leq i \leq n}) = t'(b_{1 \leq i \leq n})$ then: there is an edge $e = (v_t, v_{t'}) \in E_I$ with $L_I(e) = r\_s$.

Example 11. *An example of our* IM *process is given in Figure 8 where part (A) is the relational instance and part (B) is its corresponding instance graph.* □

Given the above, we complete the definition of our *complete mapping* process (*CM*). We notice that this process is query language independent in the sense that, once the graph database is generated from the mapping, any query language (e.g. Cypher [10], Gremlin [144], PGQL [9], G-CORE [24]) can be used to query it.

5 Properties of *CM*

In this section, we discuss the properties of our complete mapping, which includes the newly introduced properties (*monotonicity* and Update Preservation (*UP*)) in addition to those from [136].

We first show that our *CM* process preserves the five mapping properties: (1) *information preservation*; (2) *query preservation*; (3) *semantics preservation*; and (4) *monotonicity*. In addition, we introduce a new mapping property, called *update preservation*, that aims to preserve the semantics of SQL update operations during the mapping process. We give a formal definition of this new property and we show that our *CM* process preserves it

To our knowledge, this is the first paper that considers data mapping under update semantics.

5.1 Information Preservation

The following theorem demonstrates that our *CM* process does not lose any part of the information in the relational instance being translated :

Theorem 1. *The direct mapping* CM *is information preserving.* □

Proof. Theorem 1 can be proved by demonstrating the existence of a computable inverse mapping $CM^{-1} : D_G \rightarrow D_R$ that reproduces the initial relational database from the generated graph database.That is, for any relation database D_R, we have $CM^{-1}(CM(D_R)) = D_R$. As our mapping CM is comprised of two distinct steps (schema and instance mappings), then CM^{-1} requires the definition of SM^{-1} and IM^{-1} processes. For the sake of clarity, the definitions of CM^{-1} are reported to Appendix 1.

□

5.2 Query Preservation

When mapping SQL queries into Cypher queries, particular attention must be paid to the *edge-isomorphism* semantics of the Cypher language. Given a Cypher query Q_c, the edge-isomorphism semantics [145] requires that any pair of edges e_1 and e_2 that belong to the same pattern (i.e. pattern of Match or Where statements), must be mapped into different data edges. However, no isomorphism semantics is applied for vertices which means that two different vertices in Q_c may be mapped to the same data vertex. Given this requirement of Cypher language, a naive mapping of an SQL query Q_s into its equivalent query Q_c may result in some edge-isomorphism semantics that has no sense in the relational part, i.e. loss of semantics. We explain this point in the next example.

Example 12. *Consider the relational instance of Figure 8 (A) and let define the SQL query Qs (Figure 9) over it. It is easy to see that the result of this query will be given by a pair containing the same patient having an identifier equal to* 011*. One can generate many Cypher queries that are equivalent to Q_s, among others: Q_c^1,Q_c^2,Q_c^3 and Q_c^4 as mentioned in Figure 9.*

The Cypher queries Q_c^1 and Q_c^2 do not return any result due to the presence of edge-isomorphism semantics. Precisely, the relationships $r1$ and $r2$ make part of the same Match statement which yields to the application of edge-isomorphism between them. Since the two vertices $v1$ and $v2$ refer to the same patient, then the only match result of Q_c^1 should map $r1$ and $r2$ to the same data edge, which is forbidden as $r1$ and $r2$ are supposed to be different. Similarly, for the relationships $r1$ and $r2$ of Q_c^2. However, remark that the relationships $r1$ and $r2$ of Q_c^3 belong to different Match *statements, which means that no edge-isomorphism will be applied between them. That is, Q_c^3 will return the same result as Q_s, i.e., a pair containing the same patient identified by* 011*. However, Q_c^3 suffers from cartesian products: it will first look for all possible matches of $(v1, v2, v3)$ and then apply the two* Match *statements over them, which will be time-consuming. The Cypher query Q_c^4 is equivalent to Q_c^3 but it will run much faster, without computing cartesian products. Notice that our approach will produce Q_c^4 for Q_s.* □

Q_s :
```
SELECT a1, a2 FROM Patients AS a1, Patients AS a2, Admissions AS a3
WHERE a1.PatNo= a3.Pat_No AND a2.PatNo= a3.Pat_No AND a1.PatNo= 011
AND a2.PatNo= 011;
```

Q_c^1 :
```
MATCH (v1:Patients)-[r1:Admissions_Patients]-(v3:Admissions)
-[r2:Admissions_Patients]-(v2:Patients)
WHERE v1.PatNo = 011 AND v2.PatNo = 011
RETURN v1, v2;
```

Q_c^2 :
```
MATCH (v1:Patients)-[r1:Admissions_Patients]-(v3:Admissions),
(v2:Patients)-[r2:Admissions_Patients]-(v3)
WHERE v1.PatNo = 011 AND v2.PatNo = 011
RETURN v1, v2;
```

Q_c^3 :
```
MATCH (v1:Patients), (v2:Patients), (v3:Admissions)
MATCH (v1)-[r1:Admissions_Patients]-(v3)
MATCH (v2)-[r2:Admissions_Patients]-(v3)
WHERE v1.PatNo = 011 AND v2.PatNo = 011
RETURN v1, v2;
```

Q_c^4 :
```
MATCH (v1:Patients)-[:Admissions_Patients]-(v3:Admissions)
MATCH (v2:Patients)-[:Admissions_Patients]-(v3)
WHERE v1.PatNo = 011 AND v2.PatNo = 011
RETURN v1, v2;
```

Figure 9: Example of mapping SQL queries to Cypher queries

This tells us that a naive mapping of an SQL query may lose the semantics of the original query or yield inefficienct computation. This suggests converting multiple *Join operations*, in some SQL query Q_s, into multiple MATCH clauses in the resulting Cypher query Q_c in order to avoid edge-isomorphism between edges and cartesian products between vertices. This principle is applied to generate Q_c^4 of the previous example.

Theorem 2. *The direct mapping* CM *is query preserving.* □

Theorem 2 can be proved by providing a correct algorithm that computes a Cypher query for any SQL query. Precisely, given a relational instance I_R, its equivalent graph instance I_G, and an SQL query Q_s over I_R. The algorithm must produce a Cypher query Q_c that returns the same entities as Q_s, i.e., there exists a mapping from each row in $[Q_s]_{I_R}$ into a row in $[Q_c]_{I_G}$, and vice versa. For the sake of clarity, we provide a query mapping algorithm that deals only with simple SQL queries. However, one can easily extend our algorithm to deal with composed versions (e.g. queries with IN clause). We prove Theorem 2 in two parts: (I) and (II).

(I) Our algorithm, referred to as $S2C$, is summarized in Algorithm 1. In a nutshell, for any SQL query Q_s and any relational schema S_R, $S2C$ produces an equivalent Cypher query Q_c as follows:

(1) **Renaming Relations**: $S2C$ first checks if any relation in Q_s has an alias, and if not, it assigns an alias to this relation using AS clause (line 1). These aliases will make easy the creation of Cypher variables in Q_c.

(2) **Extracting Clauses**: $S2C$ parses Q_s and extracts its different clauses (line 2).

(3) **Expressing relations**: Recall that each relation ri ($1 \leq i \leq k$) in the relational schema S_R is assigned an alias ai (line 1). That is, a Match statement is created in Q_c by creating a variable ai with label ri for each relation ri (line 4).

(4) **Expressing joins**: In order to avoid edge-isomorphism semantics in Q_c, each join operation in Q_s, between two relation alias ai and aj, is expressed via a separated Match statement (line 5). This statement expresses a relationship between the Cypher variables ai and aj and its label is a concatenation of the relation names ri and rj.

(5) **Expressing conditions**: Recall that the Where clause of Q_s may express both join operations and conditions over attributes. As join operations are already treated (line 5), a Where statement is created in Q_c in order to express the remaining conditions in WC defined over attributes of S_R (line 6). Precisely, any condition defined in WC over an attribute a of a relation alias ai is translated into a condition in Q_c defined over attribute a of the Cypher variable ai. Consider for instance the queries of Example 12. Remark that the condition a1.PatNo=11 in Q_s is expressed in Q_c^4 with v1.PatNo=11 as the Cypher variable $v1$ is created for the relation alias $a1$.

(6) **Expressing return**: A Return statement is created in Q_c (line 7) such that: it returns a Cypher variable ai only if its alias ai is concerned by the Select clause of Q_s.

(7) **Optimizing the Cypher query**: According to the Cypher documentation, a Match statement that looks only for separated vertices without any relationship between them may take a lot of time. Remark that this kind of statement is created at line 4 of the algorithm. To make our approach efficient, the algorithm avoids this case by integrating the definition of some Cypher variables to the Match statement created at line 5. Consider for instance the queries of Example 12. The Cypher query Q_c^3 is generated according to steps 1–5 of the algorithm (i.e. lines 1–7), while applying the final step (i.e. lines 1–8) yields to the optimized version Q_c^4.

Finally, the algorithm returns the generated Cypher query at line 9.

A running example of our query mapping algorithm is given in Appendix 2 (Figures 26 and 27).

(II) One can easily check the correctness of our algorithm $S2C$ based on the following points. (1) Any relation in the SQL query (Q_s) is represented with a Cypher variable in Q_c having the name of the relation as a label. (2) Any join operation in Q_s, between two relation alias ai and aj, is expressed in Q_c with a relationship connecting the Cypher variables ai and aj (corresponding to alias ai and aj respectively). The relationship is labeled ri_rj and is expressed as a separated Match statement in order to avoid the semantic of edge-isomorphism. (3) Any attribute condition in Q_s over a relation alias ai is expressed by an equivalent attribute condition over the Cypher variable ai. (4) For each relation alias ai selected as result by Q_s, its corresponding Cypher variable ai is returned by Q_c. This ensures that each entity returned by Q_s has an equivalent entity returned by Q_c. (5) No edge-isomorphism semantics is applied by the generated Cypher query to avoid any semantics loss. Given the above, our algorithm $S2C$ is correct and produces an equivalent Cypher query Q_c for any SQL query Q_s.

Parts (I) and (II) complete the proof of Theorem 2.

5.3 Semantic Preservation

We show that our CM process is semantic preserving by studying (in)consistency of relational databases and graph databases. Recall that a direct mapping is considered semantic preserving if it translates any consistent (resp. inconsistent) relational database into a corresponding consistent (resp. inconsistent) graph database. While the consistency of relational databases is well-known, no definition is given for graph databases except some ones that do not cover all relational constraints (e.g. [3]).

Algorithm 1 S2C

Input: A simple SQL query Q_s, A relational schema S_R
Output: Its equivalent Cypher query Q_c.

1: if not applied, assign an alias ai to each relation name ri in Q_s (via as clause);
2: Extract the Select clause (SC), the From clause (FC) and the Where clause (WC) from Q_s;
3: Let $\{r1, ..., rk\}$ be the set of relation names in Q_s and $\{a1, ..., ak\}$ be the set of alias assigned to them respectively;
4: Create a statement Match (a1:r1),...,(ak:rk) in Q_c where each ai ($\leq i \leq k$) is a Cypher variable referring to the alias ai;
5: For each join in WC between relation alias ai and aj, create a statement Match (ai)-[:ri_rj]-(aj) in Q_c expressing this join via a relationship labeled "*ri_rj*";
6: Generate a Where statement in Q_c by considering all conditions in WC expressed over relation attributes;
7: Generate a Return statement in Q_C by returning Cypher variables corresponding to all relation alias selected by SC;
8: Optimize Q_c;
9: Return Q_c;

We next provide a consistency definition for graph databases that considers all previously defined relational constraints.

Definition 16 (*Graph consistency*)**.** *Given a graph database* $D_G = (S_G, I_G)$ *with* $S_G = (V_S, E_S, L_S, A^a_S, A^c_S, Pk, Fk, N_S, C_S, U_S, D_S)$. *An instance graph* $I_G = (V_I, E_I, L_I, A^a_I, A^c_I)$ *is said to be* consistent *w.r.t* S_G *if:*

1. *For each vertex* $v_s \in V_S$ *with* $Pk(v_s)$=$\{a_1, ..., a_n\}$ *and each vertex* $v \in V_I$ *that corresponds to* v_s*: there exists no attribute* $a_{1\leq i\leq n}$ *with* $A^c_I(v, a_i) =$ NULL. *Moreover, for each* $v' \in V_I$*, if* $A^c_I(v, a_i) = A^c_I(v', a_i)$ *for each* $a_{1\leq i\leq n}$ *then* $v = v'$ *must hold.*

2. *For each edge* $e_s \in E_s$ *with* $Fk(e_s, s)$=$\{a_1, ..., a_n\}$ *and* $Fk(e_s, d)$=$\{b_1, ..., b_n\}$*, if* $e_i = (v, v') \in E_I$ *is an edge that corresponds to* e_s *then we have:* $A^c_I(v, a_i) = A^c_I(v', b_i)$ *for each* $1 \leq i \leq n$.

3. *For each vertex* $v_s \in V_s$ *and each attribute* $a \in N_S(v_s)$*: if* $v_i \in V_I$ *corresponds to* v_s *then* $A^c_I(v, a) \neq$ NULL.

4. *For each vertex* $v_s \in V_s$ *and each vertex* $v \in V_I$ *that corresponds to* v_s*, the boolean formulas* $C_S(v_s)$ *yields to* true *by replacing each attribute* a *in* $C_S(v_s)$ *with its value* $A^c_I(v, a)$. □

Intuitively, I_G is *consistent* w.r.t S_G if it satisfies all constraints of S_G.

Theorem 3. *The direct mapping* CM *is semantic preserving.* □

Proof. Recall that the definition of graph schema has been done to express any relational constraint in terms of a graph. In other words, each constraint in S_R has its equivalent constraint in S_G and vice versa. Since the semantics constraint is preserved between the two models, then each relational instance that is consistent (resp. inconsistent) w.r.t S_R will produce a consistent instance graph (resp. inconsistent) w.r.t S_G. Proving that is straightforward and can be done by contradiction based on the mapping rules of the *IM* process (Section 4.3). Given a relational database $D_R = (S_R, I_R)$ and let $D_G = (S_G, I_G)$ be its equivalent graph database generated by *CM*. We assume that *CM* is not semantic preserving. This means that either (A) I_R is consistent w.r.t S_R and I_G is inconsistent w.r.t S_G; or (B) I_R is inconsistent w.r.t I_R while I_G is consistent w.r.t S_G. We give only proof of case (A) since that of case (B) can be done similarly.

We suppose that I_R is consistent w.r.t S_R while I_G is inconsistent w.r.t S_G. Definition 16 states four conditions that must be satisfied by I_G in order to be consistent. That is, I_G is inconsistent if it violates at least one of these four conditions. Consider condition (1) (i.e. *primary key* constraint). I_G violates this condition if there exists a vertex $v \in V_I$ that corresponds to a vertex $v_s \in V_s$ where: $Pk(v_s) = \{a_1, ..., a_n\}$ while there exists an attribute $a_{1 \leq i \leq n}$ with $A_I^c(v, a_i) = \text{NULL}$ (*). According to the mapping rules of the *IM* process, the vertex v corresponds to a specific tuple t in I_R, and the values of attributes $a_1, ..., a_n$ correspond to the primary key of tuple t. Deduction (*) implies that the tuple t assigns a NULL value to some attribute $a_{1 \leq i \leq n}$ which makes I_R inconsistent according to the definition of relational consistency (Section 2). This leads to a contradiction as we supposed that I_R is consistent. Similarly, one can easily check that a contradiction is obtained each time we suppose that I_R violates one of the remaining conditions of Definition 16. This means that I_R cannot be inconsistent while supposing that I_R is consistent. Therefore, a consistent I_R implies a consistent I_G.

By doing proof of part (B) in a similar way, we can deduce that if I_R is consistent (resp. inconsistent), then its corresponding instance graph I_G is consistent (resp. inconsistent) as well. This completes the proof of Theorem 3.

□

5.4 Update Preservation

To the best of our knowledge, no previous work has studied the preservation of update semantics. Along the same principle as query preservation, a mapping process is *update preserving* if any update done on the relational instance can be translated into an update on the equivalent graph instance such that the two update operations yield the same effect.

Definition 17 (*Impact of Update*)**.** *Given a relational database $D_R = (S_R, I_R)$ and an SQL update U_S expressed over I_R. The* impact *of U_S over I_R, denoted by $[U_S]_{I_R}$, is given by:*

- *a set of tuples of I_R in case of* Delete *operation;*
- *a set of new tuples in case of* Insert *operation;*
- *a set of triplets (t, a, v) in case of* Set *operation where: t is a tuple in I_R, a is an attribute in t and v is a new value assigned to a in t;*

Similarly, the impact *of a Cypher update U_C over an instance graph I_G, written $[U_C]_{I_G}$, is formalized as $[U_S]_{I_R}$ by treating tuples as vertices and tuple attributes as vertex attributes.* □

As we are studying the mapping of updates from the relational model into the graph model, we introduce hereafter the notion of *impact equivalence* between updates belonging to these models.

Definition 18 (*Impact Equivalence*)**.** *Given a relational instance I_R and its equivalent instance graph I_G (i.e. produced by CM), and let U_S (resp. U_C) be an SQL update (resp. a Cypher update) expressed over I_R (resp. I_G). We say that $[U_S]_{I_R}$ is* equivalent *to $[U_C]_{I_G}$, written $[U_S]_{I_R} \equiv [U_C]_{I_G}$, if:*

1. *a tuple t is deleted from I_R by U_S if and only if U_C applies a* ***detach delete*** *in I_G over the vertex v_t that corresponds to t.*
2. *a tuple t is inserted in I_R by U_S if and only if U_C inserts a vertex v_t in I_G that corresponds to t. Moreover, t refers to (resp. referred by) the tuple s if and only if U_C inserts a new edge from v_t to v_s (resp. from v_s to v_t) where v_s corresponds*

to s in I_G.

3. *U_S assigns a new value v to the attribute a of a tuple t in I_R if and only if U_C assigns the value v to the attribute a of vertex v_t in I_G where v_t corresponds to t. Moreover:*

   (a) *t referred to (resp. referred by) the tuple o via the old value of a if and only if U_C deletes the edge from v_t to v_o (resp. from v_o to v_t).*

   (b) *t refers to (resp. referred by) the tuple n via the new value of a if and only if U_C inserts a new edge from v_t to v_n (resp. from v_n to v_t).*

Recall that when a tuple t is mapped to a vertex v, via some mapping process, we say that v_t corresponds *to t.* □

Recall that our *IM* process inserts a vertex v_t in I_G for each tuple t in I_R. Moreover, if t refers to (referred by) a tuple s via a foreign key (resp. a primary key), then *IM* inserts an edge from v_t to v_s (resp. from v_s to v_t). That is why when t is deleted from I_R then its corresponding vertex v_t is deleted from I_G as well as all edges related to it (i.e. edge representing primary and foreign keys related to t). The **detach delete** operation allows to delete a vertex with all edges related to it. When the value of an attribute a of t is updated, then the new value is assigned to the vertex v_t in I_G. Moreover, if t was referring to (resp. was referred by) a tuple o via the old value of a, and now is referring to (resp. is referred by) a tuple n via the new value of a, then an old edge is deleted between v_t and v_o (case 3-a) and a new one is inserted between v_t and v_n (case 3-b).

We are now ready to provide our definition of *update preservation*.

Definition 19 (Update preservation)**.** *A direct mapping* DM *is* update preserving *if for any relational database $D_R = (S_R, I_R)$ and any SQL update U_S over I_R, there exists a Cypher update U_C such that: $[U_S]_{I_R} \equiv [U_C]_{I_G \in DM(S_R, I_R)}$.* □

Next, we state the main result of this subsection.

Theorem 4. *The complete mapping* CM *is* update preserving. □

Algorithm 2 S2C^u

Input: A simple SQL update query U_s and a relational schema S_R
Output: Its equivalent Cypher update query U_c.

if not applied, assign an alias ai to each relation name ri in U_s (via as clause);
switch *type of update* **do**
  case *Insert*:
    Let r be the name of the relation concerned by the insertion and a its alias;
    let $\{p_1, \cdots, p_k\}$ be the attributes of r and $\{v_1, \cdots, v_k\}$ values assigned to them via the insertion;
    Replace v_i with $D(r, p_i)$ for each $v_{1 \leq i \leq k}$ = NULL;
    U_c := CREATE (a:r $\{p_1 : v_1, \cdots, p_k : v_k\}$);
    for $r[p_1, \cdots, p_m] \rightarrow s[b_1, \cdots, b_m]$ in S_R $(m \leq k)$ **do**
      //aa is a Cypher variable referring to the relation s
      Add the next statement to U_c:
      Match (aa:s$\{b_1 : v_1, \cdots, b_m : v_m\}$) Create (a)-[:r_s]→(aa);
    end for
    for each $s[p_1, \cdots, p_m] \rightarrow r[b_1, \cdots, b_m]$ in S_R $(m \leq k)$ **do**
      //aa is a Cypher variable referring to the relation s
      Add the next statement to U_c:
      Match (aa:s$\{b_1 : v_1, \cdots, b_m : v_m\}$) Create (aa)-[:s_r]→(a);
    end for
  case *Delete*:
    Let r be the name of relation concerned by the deletion and a its alias;
    Let *cnd* be the deletion condition expressed over attributes of r;
    U_c := Match (a:r) Where *cnd* Detach delete a;
  case *Set*:
    Let r be the name of the relation concerned by the *Set* operation and a its alias;
    Let *cnd* be the condition of the *Set* operation expressed over attributes of r;
    Let $\{p_1, \cdots, p_k\}$ be the attributes concerned by the *Set* operation and $\{v_1, \cdots, v_k\}$ their new values;
    U_c := Match (a:r) Where *cnd*;
    //Deleting some old edges
    for each $r[a_1, \cdots, a_m] \rightarrow s[b_1, \cdots, b_m]$ in S_R with $p_{1 \leq i \leq k} \cap a_{1 \leq i \leq m} \neq \emptyset$ **do**
      //aa is a Cypher variable referring to the relation s
      Add the next statement to U_c:
      Optional Match (a)-[old_edges:r_s]→(aa:s) Where a.a$_1$=aa.b$_1$ and ... a.a$_m$=aa.b$_m$ Delete old_edges;
    end for
    for each $s[b_1, \cdots, b_m] \rightarrow r[a_1, \cdots, a_m]$ in S_R with $p_{1 \leq i \leq k} \cap a_{1 \leq i \leq m} \neq \emptyset$ **do**
      //aa is a Cypher variable referring to the relation s
      Add the next statement to U_c:
      Optional Match (aa:s)-[old_edges:s_r]→(a) Where aa.b$_1$=a.a$_1$ and ... aa.b$_m$=a.a$_m$ Delete old_edges;
    end for
    //Applying the update on attribute values
    Add the next statement to U_c: Set a.p_1=v_1,...,a.p_k=v_k;
    //Creating some new edges
    for each $r[a_1, \cdots, a_m] \rightarrow s[b_1, \cdots, b_m]$ in S_R with $p_{1 \leq i \leq k} \cap a_{1 \leq i \leq m} \neq \emptyset$ **do**
      //aa is a Cypher variable referring to the relation s
      Add the next statement to U_c:
      Match (aa:s) Where a.a$_1$=aa.b$_1$ and ... a.a$_m$=aa.b$_m$ Create (a)-[:r_s]→(aa);
    end for
    for each $s[b_1, \cdots, b_m] \rightarrow r[a_1, \cdots, a_m]$ in S_R with $p_{1 \leq i \leq k} \cap a_{1 \leq i \leq m} \neq \emptyset$ **do**
      //aa is a Cypher variable referring to the relation s
      Add the next statement to U_c:
      Match (aa:s) Where a.a$_1$=aa.b$_1$ and ... a.a$_m$=aa.b$_m$ Create (aa)-[:s_r]→(a);
    end for
Return U_c;

Intuitively, for any update over I_R via SQL, one can express an equivalent update over I_G via Cypher. Proving Theorem 4 can be done by providing an algorithm that, for any SQL update U_s, produces an equivalent Cypher update U_c such that U_s over any relational instance I_R has the same impact as U_c over the instance graph I_G that corresponds to I_R.

Our algorithm, referred to as S2C^u, is summarized in Algorithm 2. In essence, S2C^u inputs an SQL update U_s and a relational schema S_R, and generates an equivalent Cypher update U_c as follows:

(1) **Renaming Relations**: S2C^u first checks if any relation in U_s has an alias, and if not, it assigns an alias to this relation using AS clause (line 1). These aliases will make easy the creation of Cypher variables in U_c.

(2) Case of **Insert** (lines 3–13): S2C^u first parses U_s and extracts the name of relation concerned by the insertion, i.e. r, as well as the attribute values that will compose the new tuple of r (lines 4–5). If some attribute has not given any value then the default value, if defined, is extracted from S_R and applied (line 6). Next, a statement is assigned to U_c in order to create a vertex a (labeled r) which has the same attribute values of the inserted tuple (line 7). If the newly inserted tuple of the relation r refers to a tuple of another relation s (i.e. via a foreign key), then: a **Match** statement is added to U_c in order to get the vertex of the data graph that corresponds to the tuple of s (expressed via the Cypher variable aa); and a **create** statement is added to U_c in order to add an edge between vertices a and aa (lines 8–10). Intuitively, the foreign key between the newly inserted tuple of r and some tuple of s is expressed by U_c via an edge between the vertex a (representing the tuple of r) and the vertex aa (representing the tuple of s). The same principle is applied in the case where the tuple of a relation s refers to the newly inserted tuple of r (lines 11–13).

(3) Case of **Delete** (lines 14–17): A relational delete operation is applied over a relation r, and is based on some condition *cnd*. Then, once these information are extracted from U_s (lines 15–16), a **Match** statement is added to U_c in order to get the vertex of the data graph that corresponds to the tuple deleted by U_s, and next, a **Detach delete** statement is added to U_c in order to delete this vertex along all its edges (line 17).

(4) Case of **Set** (lines 18–35): A relational *Set* operation aims to assign new values (i.e. $v_1, ..., v_k$) to some attributes (i.e. $p_1, ..., p_k$) of a tuple of some relation r. Thus, the equivalent Cypher update starts first by retrieving the vertex (identified by the

variable a) that corresponds to the tuple concerned by the update (lines 19–22). Given a foreign key $r[a_1, ..., a_m] \rightarrow s[b_1, ..., b_m]$ and a tuple t_r of r that refers to a tuple t_s of s via this foreign key. Recall that our mapping process expresses this relation via an edge from the vertex v_r (that corresponds to t_r) to the vertex v_s (that corresponds to t_s). It is clear that any change on attributes $a_{1 \leq i \leq m}$ of the tuple t_r makes this latter refer to another tuple $t_s^{'}$ of s rather than t_s. That is why the equivalent Cypher query U_c looks first for the old edge between v_r and v_s and deletes it (lines 23–25); then it creates a new edge between v_r and $v_s^{'}$, i.e. the vertex that corresponds to $t_s^{'}$ (lines 30–32). The same principle is applied for a foreign key $s[b_1, ..., b_m] \rightarrow r[a_1, ..., a_m]$ (lines 26–28 for deleting old edges, and lines 33-35 for creating new edges).

Finally, the algorithm S2C^u returns the generated Cypher update U_c (line 36).

The correctness of algorithm S2C^u can be checked obviously based on the following points. (1) For each tuple inserted by U_s, a vertex is created by U_c that contains all information of the newly inserted tuple (i.e. name of relation and attribute values). (2) For each tuple deleted by U_s, its corresponding vertex is found by U_c and is then deleted too with all its related edges. (3) Each update done by U_s on attribute values of some tuple t is followed by an update done by U_c over the same attributes of the vertex v_t that corresponds to t. (4) For each tuple inserted (resp. deleted) by U_s, U_c creates (resp. deletes) some edges in order to maintain integrity constraints related to this tuple (i.e. primary and foreign keys). (5) These integrity constraints are also maintained by U_c in cases where U_s changes values of some attributes that play a role in primary/foreign keys. (6) The default attribute property, expressed by S_R, is preserved by algorithm S2C^u.

Given the above, for any SQL update U_s over a relational instance I_R, algorithm S2C^u generates a Cypher update U_c that makes the same impact of U_s (in terms of data and constraints) over the corresponding instance graph I_G.

5.5 Monotonicity

We next state the last property of our mapping process.

Theorem 5. *The direct mapping* CM *is* monotonic. □

Table 6: Details of experiment datasets.

Name	Domain	Type	Count of relations	Count of rows	Size	Null values
CORA	Education	Real	3	57,353	4.5 MB	Yes
PKDD'99	Financial	Real	8	1,090,086	78.8 MB	Yes
IMDb	Entertainment	Real	7	5,647,694	477.1 MB	Yes
Sakila	Retail	Synthetic	16	47,273	6.4 MB	Yes
TPCH	Retail	Synthetic	8	6,885,051	2 GB	Yes
TPCDs	Retail	Synthetic	24	21,005,545	4.8 GB	Yes

Table 7: Metrics of data mapping: Neo4j-ETL vs Our approach.

Datasets	Neo4j-ETL		Our Approach	
	Vertices number	Edges number	Vertices number	Edges number
Cora	51924	54645	57353	60074
Sakila	40811	115307	47273	121769

Proof. The proof of Theorem 5 is straightforward and can be done by contradiction as follows. Given a relational schema S_R, two instances I_R^1 and I_R^2 of S_R and their corresponding instance graphs I_G^1 and I_G^2 produced respectively via $CM(S_R, I_R^1)$ and $CM(S_R, I_R^2)$. Supposing that the CM process is not *monotone* implies that $I_R^1 \subseteq I_R^2$ but $I_G^1 \not\subseteq I_G^2$. The statement $I_G^1 \not\subseteq I_G^2$ means that there exists at least one vertex/edge that belongs to I_G^1 but not to I_G^2. Recall that for each tuple t_1 in I_R^1, our mapping process CM creates a vertex v_1 in I_G^1 that contains all information of t_1. Moreover, for any pair of tuples t_1 and t_2 in I_R^1 such that t_1 refers to t_2 via a foreign key, our CM process creates edges between the vertices v_1 and v_2 that corresponds to t_1 and t_2 respectively. That is, if a vertex v_1 belongs to I_G^1 but not to I_G^2 then there exists a tuple t_1 that belongs to I_R^1 but not to I_R^2, which means that $I_R^1 \not\subseteq I_R^2$ (contradiction). Moreover, if an edge between the vertices v_1 and v_2 exists in I_G^1 but not in I_G^2, then this means that either: (a) one of the vertices v_1 and v_2 does not exist in I_G^2 which yields to a contradiction as explained above; or (b) the values assigned to attributes of v_1 in I_G^2 do not allow to refer to v_2, i.e. the tuple t_1 does not refer to the tuple t_2 in I_R^2, which implies that $I_R^1 \not\subseteq I_R^2$ (contradiction).

Given the above, $I_R^1 \subseteq I_R^2$ implies $I_G^1 \subseteq I_G^2$.

□

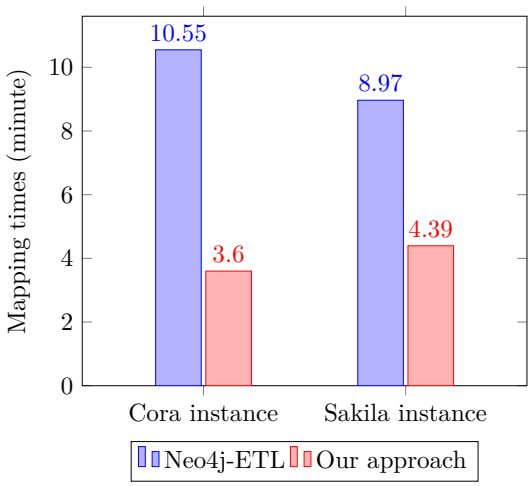
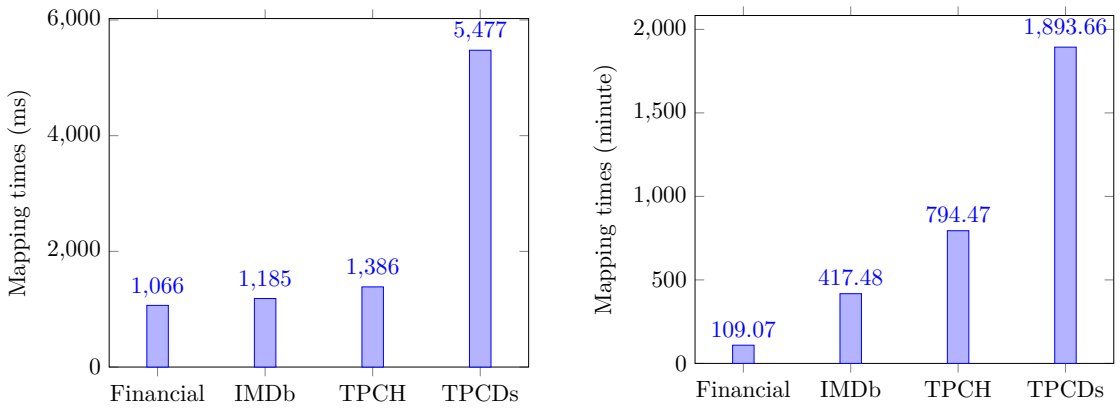


Figure 10: Instance-mapping based comparison.

6 Experimental Evaluation

Using real-life and synthetic datasets[1], we next conduct extensive experimental study in order to verify the effectiveness, efficiency and correctness of our approach.

The *effectiveness* of our approach refers to its ability to be seamlessly applied across a variety of real-world database scenarios, highlighting its adaptability and practical utility. The *efficiency* corresponds to the approach's capability to transform relational data and queries into their graph equivalents within a reasonable time frame while scaling effectively across datasets of different sizes. Finally, the *correctness* of our approach is measured by the accuracy of the mapping between any relational entity and its corresponding graph entity.



(a) SM (b) IM

Figure 11: Efficiency of complete mapping.

[1]https://relational.fit.cvut.cz

6.1 Experimental setting

A) Datasets.

We used three real-life datasets (CORA, PKDD'99, IMDb) and three synthetic datasets (Sakila, TPCH, TPCDs) which are described as follows:

1. CORA: it represents a paper's citation network consisting of scientific publications, description of their content, and citations between them.
2. PKDD'99: is a financial dataset containing information on successful and unsuccessful loans, their information, their transactions and the corresponding clients.
3. IMDb: is an international dataset containing information on movies and their corresponding directors and actors.
4. Sakila: is a synthetic movies database that was initially developed by MySQL former members in order to be used for examples in books, tutorials, articles, samples, and so forth. The Sakila sample database also serves to highlight features of MySQL such as Views, Stored Procedures and Triggers.
5. TPCH: is the benchmark published by the Transaction Processing Performance Council (TPC) for decision support.
6. TPCDs: is a benchmark for decision support that emulates various important aspects of a decision support system, such as data maintenance and queries.

Table 6 gives more details about our 06 datasets.

B) Implementation.

Algorithms. We implemented our algorithms SM, IM, SM^{-1}, IM^{-1}, $S2C$ and $S2C^u$ all in Java. Moreover, another algorithm, named *ConsChecker*, is implemented in order to check consistency of graph instances.

Setup setting. All the experiments were performed on a Windows 11 machine with Intel Core(TM) i5-1135G7 2.40GHz CPU, 8GB of memory and 500 GB of SSD storage. In addition, our relational datasets (resp. graph databases) were handled with MySQL 8.0.30 (resp. Neo4j community edition 5.2.0).

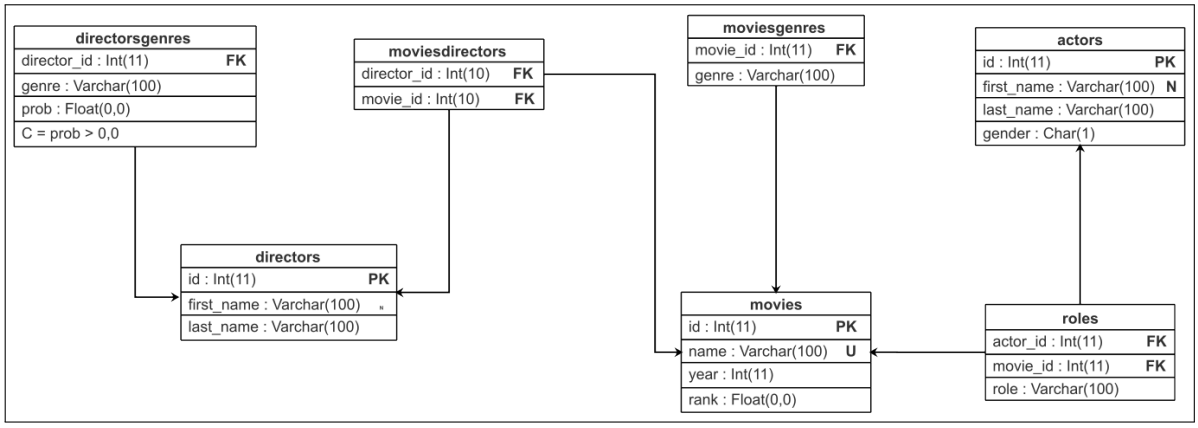


Figure 12: Relational schema of IMDb.

6.2 Experimental results.

We report hereafter our findings.

EXP-1 : Efficiency of our CM process.

We considered the Cora and Sakila datasets and we measured the mapping times required respectively by our approach and that of the Neo4j-ETL. The results are reported in Figure 10. It tells us that our approach is consistently faster than the Neo4j-ETL approach for both the Cora and Sakila instances. This is likely due to the fact that our approach uses a direct mapping from MySQL to Neo4j, whereas the Neo4j-ETL approach involves firstly generating CSV files (node.csv, relationship.csv) from MySQL and then importing them into Neo4j as is mentioned in Neo4j-ETL architecture [146]. This extra step in the Neo4j-ETL approach likely adds additional overhead, resulting in longer mapping times.

In addition, our approach outperforms that of [114] and [118]. Precisely, [114] (resp. [118]) took 10.55 (resp. 8.97) minutes to map the Cora (resp. Sakila) dataset into a graph database while our approach took only 3.6 (resp. 4.39) minutes for the same dataset.

Table 7 gives details about the graph databases generated by our approach and Neo4j-ETL, using Cora and Sakila datasets. Clearly, our approach generates more vertices and edges which is due to the fact that, in case of a relational tuple t_1 that connects two other tuples t_2 and t_3 (i.e. case of many-to-many relation), our approach generates a vertex for t_1 (let's be v_1) and two edges connecting v_1 with the two vertices that correspond to t_2 and t_3 (let's be v_2 and v_3 respectively). However, the Neo4j-ETL

tool represents t_1 on the data graph by adding only a simple edge connecting v_2 and v_3 without creating any vertex for the tuple t_1. That is why the number of vertices generated by our approach (resp. Neo4j-ETL) is the same as (resp. lesser than) the number of rows of the relational instance.

Our approach has multiple motivations. Firstly, our mapping principle aims to maintain the original relational structure unchanged. Secondly, using a simple edge instead of a vertex to represent a row may lead to less accurate or meaningful results for certain analysis tasks.

Next, we measured the time of our schema mapping and instance mapping processes, using the datasets PKDD'99, IMDb, TPCH and TPCDs. The results are reported in Figure 11.

1. Efficiency of SM process:
   Notice that the relational schemas of PKDD'99, IMDb, TPCH and TPCDs contain respectively 8, 7, 8 and 24 relations. Figure 11 (a) tells us that the mapping time of our SM process is proportional to the size of the relational schema. In addition, the mapping time remains very negligible.

2. Efficiency of IM process:
   Figure 11 (b) shows a consistent result, that is, the larger the size of the relational instance, the longer it takes for our process IM to map it into a graph instance.

Case of IMDb dataset: Figure 12 depicts the relational schema of the IMDb dataset while its corresponding schema graph, generated by our SM process, is given in Figure 13. Remark that each relation in Figure 12 is represented with a vertex in Figure 13. Each foreign key in Figure 12 (e.g. (roles) $\rightarrow$ (actors)) is represented by a directed edge in Figure 13 (e.g. :role_actors). A special attribute pk is introduced in our schema graph to specify the primary key of each vertex. Attributes s and d on each edge specify respectively starting and ending relations of the foreign key represented by this edge.

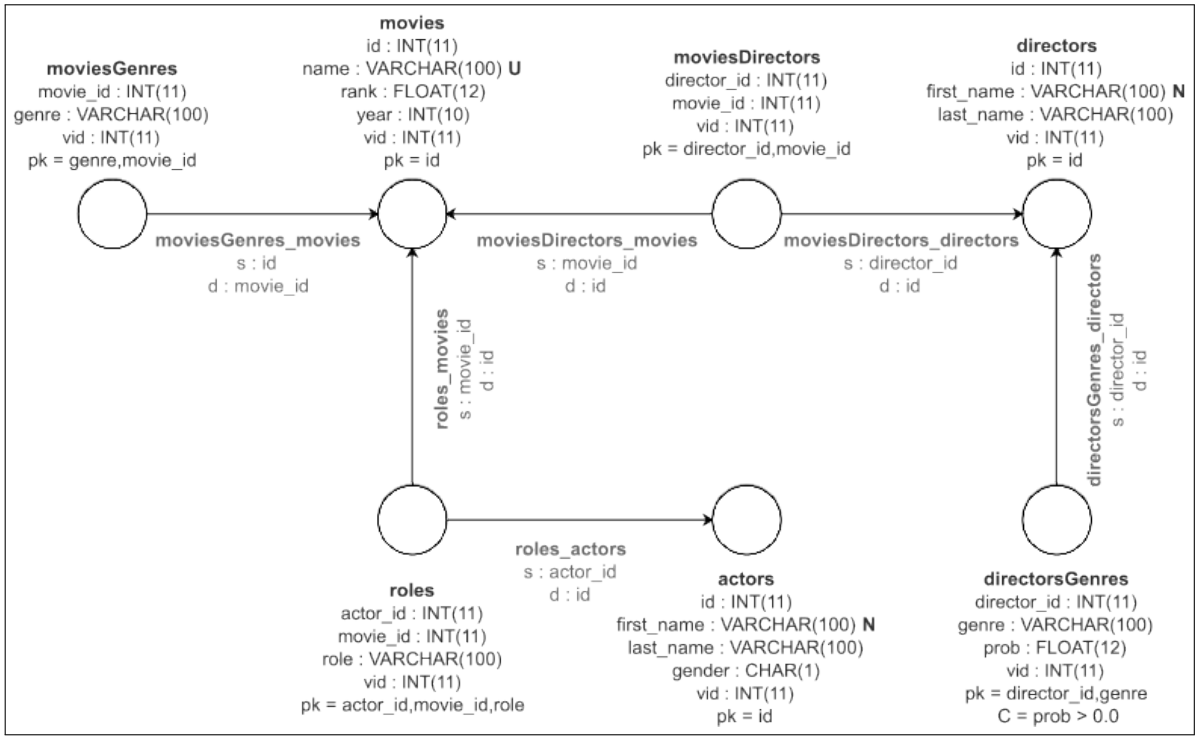


Figure 13: Schema graph of IMDb.

EXP-2 : Consistency checking.

Recall that each relational instance that is consistent (resp. inconsistent) w.r.t its relational schema, must be mapped to a consistent (resp. inconsistent) graph instance. The consistency is checked by considering the satisfaction of all constraints of the relational/graph schema (i.e. *not null*, *unique* and *check* constraints). To check the consistency of relational/graph instances, we implemented an algorithm, referred to as **ConsChecker** (not shown here), that takes a schema graph S_G and produces a Cypher query CC that aims to check whether a given graph instance respects the constraints described by S_G. The evaluation of CC over the underlying graph instance I_G must return true only if I_G is consistent.

Example 13. *Based on the schema graph illustrated in Figure 7 (B), which incorporates three integrity constraints:*

1. *A* not null *constraint on the* `Name` *attribute for the* `Doctors` *vertex.*
2. *A* unique *constraint on the* `Phone` *attribute for the* `Patients` *vertex.*
3. *A constraint specifying* `Age` > 5 *for the* `Patients` *vertex.*

```
MATCH (d:Doctors) WHERE d.DoctorNo IS NULL RETURN false
UNION
MATCH (d:Doctors) RETURN count(DISTINCT d.DoctorNo) <> count(d.DoctorNo)
UNION
MATCH (d:Doctors) WHERE d.Name IS NULL RETURN false
UNION
MATCH (p:Patients) WHERE p.PatNo IS NULL RETURN false
UNION
MATCH (p:Patients) RETURN count(DISTINCT p.PatNo) <> count(p.PatNo)
UNION
MATCH (p:Patients) WHERE p.Age <= 5 RETURN false
UNION
MATCH (p:Patients) RETURN count(DISTINCT p.Phone) <> count(p.Phone)
UNION
MATCH (d:Diagnostics) WHERE d.DiagNo IS NULL RETURN false
UNION
MATCH (d:Diagnostics) RETURN count(DISTINCT d.DiagNo) <> count(d.DiagNo)
UNION
MATCH (a:Admissions) WHERE a.AdmiNo IS NULL RETURN false
UNION
MATCH (a:Admissions) RETURN count(DISTINCT a.AdmiNo) <> count(a.AdmiNo)
```

Figure 14: Example of ConsChecker running.

Our ***ConsChecker*** *algorithm generates the Cypher query CC of Figure 14 in order to check these constraints. This query is designed to be evaluated over any graph instance in order to check whether it satisfies the above mentioned integrity constraints. Precisely, CC contains two sub-queries for each primary key to check both the* not null *and* uniqueness *constraints; and one sub-query for each of the remaining integrity constraints.*

The algorithm ***ConsChecker*** *is intuitive and creates a query CC which is the union of all sub-queries created to check satisfaction of constraints. If CC returns false then at least one sub-query has returned false, which means that at least one constraint is not satisfied. For instance, the first two sub-queries of CC in Figure 14 check the satisfaction of the primary key defined over attribute* `DoctorNo`*. The first sub-query checks whether there is some attribute* `DoctorNo` *with* `NULL` *value, which is forbidden for a primary key. The second sub-query checks whether there are two vertices that have the same value for attribute* `DoctorNo`*. If one of these sub-queries returns false then the primary key over attribute* `DoctorNo` *is not satisfied by the graph database. The remaining constraints are handled in a similar manner.* □

For each graph dataset generated by our IM process, we used the algorithm **ConsChecker** to produce a *consistency checking Cypher query* CC, and we evaluated CC over the generated graph instance. The result was true for all generated graph instances since their corresponding relational instances were all consistent. In the second experiment, we made some changes to the relational instance to become inconsistent, we mapped it into a graph instance, and in this case, **ConsChecker** was able to detect that the generated graph instance was inconsistent.

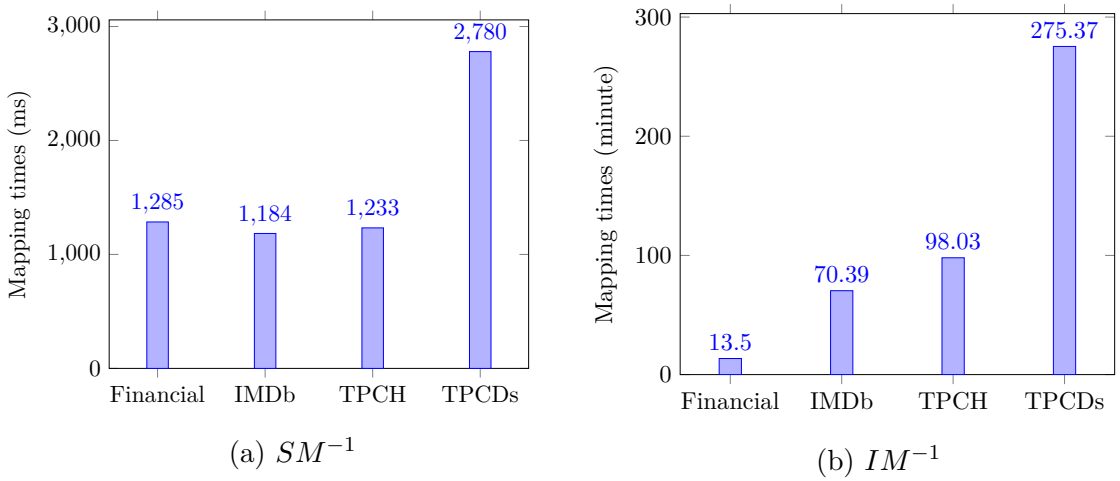
EXP-3 : Correctness of our CM process.

We used algorithms SM^{-1} and IM^{-1} (See Appendix 1) to check the correctness of our approach. Precisely, for each dataset with a relational schema S_R and a relational instance I_R, we first generated their corresponding schema graph S_G and instance graph I_G. Next, we applied an inverse schema mapping over S_G (via SM^{-1}) and an inverse instance mapping over I_G (via IM^{-1}). We found that, for any datatset, $SM^{-1}(S_G)$ = S_R and $IM^{-1}(I_G)$=I_R. The results of the inverse mapping processes are given in Figure 15. We observed the following:

1. The time of SM^{-1} (resp. IM^{-1}) is proportional to the size of the schema graph (resp. instance graph);
2. The time required by IM is greater than that required by IM^{-1}. Generally speaking, this is due to the fact that inserting vertices and edges on a graph database (i.e. going from I_R to I_G) takes more time than inserting tuples in a relational database (i.e. going from I_G to I_R). More importantly, inserting edges on the graph database (by IM) is a time-consuming process as it consists in finding vertices corresponding to each edge and then creating a relationship between them. On the other hand, converting vertices and edges on tuples (by IM^{-1}) is a less-consuming time process since it consists in adding textual lines in relational tables without any extra actions.

EXP-4 : Efficiency of our query mapper.

We defined five SQL queries (Q_1, Q_2, Q_3, Q_4 and Q_5) that correspond to five classes of queries (*Search*, *Join*, *Insert*, *Update* and *Delete* respectively). We mapped each SQL query into an equivalent Cypher query and we reported the required time in Figure 16. The SQL queries and their corresponding Cypher queries are given in Appendix 2. We give hereafter our remarks:



(a) SM^{-1} (b) IM^{-1}

Figure 15: Efficiency of inverse mappings.

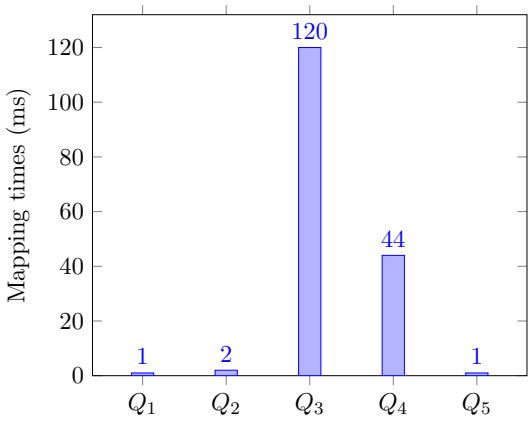


Figure 16: Efficiency of query mapper.

1. the query mapping time is negligible;
2. an *Insert-Into* query takes more time to be mapped into an equivalent Cypher query (case of Q_3). This is due to the fact that the corresponding Cypher query creates not only a vertex that corresponds to the inserted relational tuple, but also edges to handle foreign keys related to this tuple (lines 8–13 of Algorithm 2);
3. an *Update* query takes more time to be mapped into an equivalent Cypher query (case of Q_4) as this latter must remove some old edges and creates other ones to maintain the semantics of foreign keys;
4. a *Delete* query (case of Q_5) requires a minimal mapping time as its corresponding Cypher query does not manage edges;
5. read queries (case of Q_1 and Q_2) take less time than update queries, which means that Algorithm 1 is much faster than Algorithm 2;

6. the mapping time varies w.r.t various factors related to the input SQL query (e.g. number of conditions in the WHERE clause, and number of foreign keys).

7 Conclusion

In this chapter, we proposed a complete mapping (*CM*) process for translating relational databases into equivalent graph databases, addressing data, schema, constraints, and queries. We introduced a novel *schema graph* definition that uniquely captures relational structures and constraints through a graph representation, considering critical integrity constraints (unique, not null, and check constraints). We also defined *graph consistency* to verify whether an *instance graph* satisfies the schema graph's constraints. Our mapping process preserves fundamental properties including *information*, *query*, *semantic*, and *monotonicity* preservation. Additionally, we investigated query mapping between SQL and Cypher languages and introduced an *update preservation* property for maintaining update semantics. Empirical evaluations using real-life and synthetic data demonstrated the effectiveness, efficiency, and scalability of our approach.

We notice hereafter some use-cases where our mapping approach can be very useful. First of all, by mapping relational data into graph data, this allows us to perform complex analytical tasks over the graph data (using graph and path algorithms) which cannot be applied directly over the relational form. Moreover, some hidden connections can be revealed via the graph model (e.g., use of a visualization tool like Neo4j Bloom). In terms of efficiency, mapping an SQL query that is time-consuming (due to various join operations) into a Cypher query may reduce computation time as the graph model is more flexible than the join-based relational model, and thus the resulted Cypher query can be executed faster than its equivalent SQL query. In terms of data integration, our approach allows the use of data from heterogeneous sources; these data can be mapped into a single model, i.e., a graph, and then manipulated quickly and expressively. In terms of security, it is clear that Access Control Policies can be defined with more expressivity over a graph instead of tables. This will allow for fine-grained control over all entities of the data, which cannot be easily ensured within the relational model.

Chapter III: Querying Distributed Graph Data with Cypher

1 Introduction

Graph pattern matching (*GPM*), which is quite crucial in graph databases, involves the intricate process of extracting insights (modeled as graph patterns) from big graph databases. This challenge is of particular interest in different areas such as data mapping [147, 148], access control [149], finance, cybersecurity, social networks, and chemistry. Formally, *GPM* requires the identification of all subgraph isomorphisms between the graph data and the graph query, and has been shown to be an NP-Complete problem [150]. That is, computing *GPM* within large-scale graph data is time-consuming and even infeasible on a single machine. To deal with this cost/infeasibility, some works have studied the extraction of only the most relevant results of *GPM* so its computation can be done efficiently in a single machine [151, 152]. On the other hand, computing *GPM* over distributed graph data (i.e. across different machines) has received a lot of attention during the last decade both from researchers and practitioners. Some studies [129, 153, 154] tackled distributed *GPM* with limited settings (very simple graph data and queries) and under the semantics of *graph simulation*. This semantics [155] aims to reduce the cost of *GPM* from NP-Complete to PTIME by relaxing the semantics of the isomorphism and returning approximate results. However, real-world scenarios often require a rich representation of graph data (e.g. with attributes and labels), complex graph queries (e.g. with conditions, subqueries), and exact answers. Moreover, all existing graph database systems (e.g. *Neo4j*, *Memgraph*) support *GPM* under isomorphism semantics only. Other studies [127, 128] propose practical solutions for distributed *GPM*, however, they lack formalization to integrate these solutions in different contexts or compare them with other approaches. A comparison made by the Neo4j group [1] shows that the *Cypher* query language [10] is very close to the new ISO standard of graph query language (denoted by ISO GQL [11]). Cypher is currently supported by many graph database systems (e.g. *Neo4j*, *MemGraph*, *RedisGraph*, *Amazon Neptune*).

The most challenge with Cypher relies on the semantics of its implementation, i.e. all graph systems use isomorphism semantics to execute Cypher queries, which is an NP-Complete problem, and consequently makes Cypher-based querying intractable. To reduce this time, some systems (e.g. *Neo4j*) allow the use of federated graph databases which is a fragmentation of the initial large graph data into independent subgraphs.

[1] https://neo4j.com/blog/cypher-and-gql/cypher-path-gql (Accessed:2025-06-23)

The answering of Cypher queries in this case remains simple as it requires to execute the query on each fragment and to merge all results by a simple union. The task becomes more intriguing in the case of distributed data where the fragments (e.g. subgraphs) may contain common nodes and edges between them. That is, a query result may exist across one or many machines. Despite the wide use and high expressivity of Cypher, it is surprising that no work has studied the answering of Cypher queries over distributed graph data. Moreover, no graph database system allows such an answering. Furthermore, a formalization of this distributed answering is necessary to ensure the interoperability of the solution and to allow future comparisons/extensions.

Motivated by these challenges, we propose a formalization of distributed *GPM* that can be easily implemented into existing graph database systems. As a use case, an implementation has been tested on AuraDB, a cloud-based solution of Neo4j that uses the Cypher language. A real-life graph data has been distributed across many AuraDB instances, each one playing the role of a separated machine. Our proposed algorithms ensure the distributed answering of complex Cypher queries over these AuraDB instances. These algorithms apply to other *Cypher*-like query languages.

2 Preliminaries

2.1 Data Graph partitioning and assembly

2.1.1 Data Graph partitioning

We now recall the definition of distributed graph data, graph data partitioning, and assembly techniques followed by our problem statement.

Given a graph data G, its *distributed version* G^d is given by $\{P_1, \cdots, P_k\}$ where: $k \geq 2$ and each *partition* $P_{1 \leq i \leq k}$ is a subgraph of G that is hosed at a separate machine. According to the partitioning strategy, some partitions of G^d may have common nodes called *boundary nodes*. Moreover, some edges of G, called *crossing edges*, may have endpoints belonging to different partitions (i.e. machines). The graph data partitioning is NP-Complete in general [156] and it is not the focus of this work. Nevertheless, we explain briefly is main strategies.

Edge-cut Partitioning. This approach aims to partition the node set V of G into disjoint subsets $\{V_1, \cdots, V_k\}$. For each subset V_i, its corresponding edge set E_i consists of all edges in G that have both endpoints in V_i. In other words, (V_i, E_i) defines the subgraph of the partition P_i, where $V_i \cap V_j = \emptyset$ for all $i \neq j$, and $\bigcup_i V_i = V$. As a result,

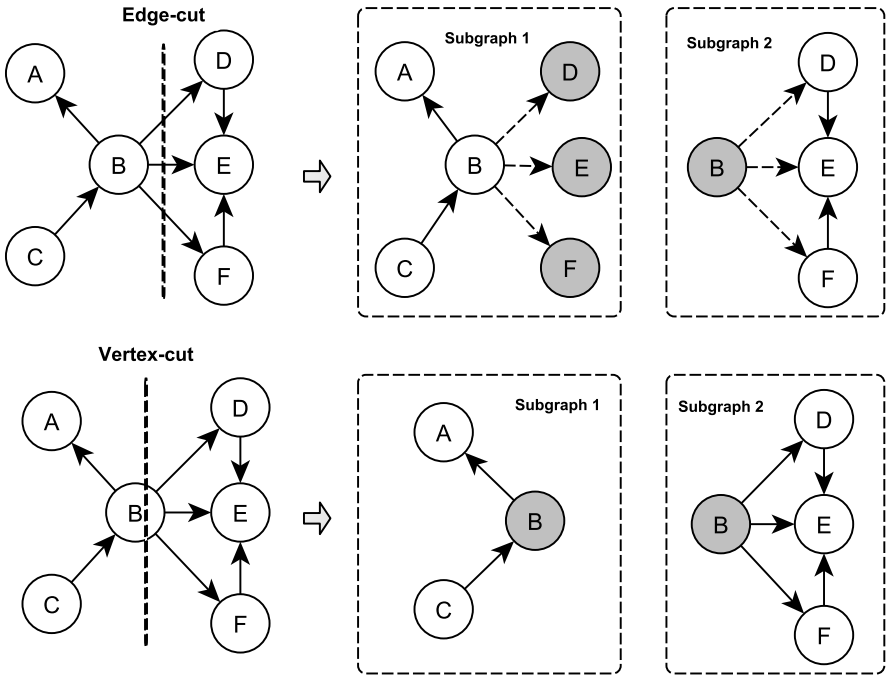


Figure 17: Vertex partitioning vs Edge partitioning

the obtained distributed graph $G^d = \{P_1, \cdots, P_k\}$ may have *crossing edges*, i.e., those connecting vertices in different partitions.

Vertex-cut Partitioning. This approach aims to partition the edge set E of G into disjoint subsets $\{E_1, \cdots, E_k\}$. For each subset E_i, its corresponding vertex set V_i consists of all vertices of G that are endpoints of edges in E_i. In other words, (V_i, E_i) defines the subgraph of the partition P_i, where $E_i \cap E_j = \emptyset$ for all $i \neq j$, and $\bigcup_i E_i = E$. As a result, the obtained distributed graph $G^d = \{P_1, \cdots, P_k\}$ may have *boundary nodes*, i.e., those that are endpoints of edges belonging to different partitions.

Notice that there exist also ***Hybrid partitioning*** [157] that combines **vertex-cut** and **edge-cut** strategies to optimize graph partitioning for power-law graphs. Unlike pure **vertex-cut** (which replicates vertices to evenly distribute edges) or **edge-cut** (which minimizes edge-cuts but may imbalance load), **Hybrid partitioning** distinguishes vertices by degree:

- **High-degree vertices** are partitioned via *vertex-cut*, distributing their edges across partitions to balance computational load.
- **Low-degree vertices** are partitioned via *edge-cut*, keeping all incident edges in one partition to minimize communication.

This approach exploits the power-law structure of real-world graphs (where few vertices have high degree and most have low degree) to simultaneously reduce communication overhead and maintain load balance. **Hybrid partitioning** aims to outperform pure **edge-cut** or **vertex-cut** by strategically trading off replication for better parallelism and lower edge-cuts.

Example 14. *Figure 17 shows the application of* edge-cut *and* vertex-cut *strategies over a graph data composed by* 6 *nodes* ($A-F$) *and* 7 *edges. The top of this figure shows an* edge-cut *partitioning where the node set is partitioned into two subsets* $\{A, B, C\}$ *and* $\{D, E, F\}$*, which yields to two partitions having* 3 crossing edges ($B \to D$, $B \to E$, $B \to F$)*. The bottom of this figure shows a* vertex-cut *partitioning where the edge set is partitioned into two subsets* $\{C \to B, B \to A\}$ *and* $\{B \to D, B \to E, B \to F, D \to E, F \to E\}$*, which yields to two partitions sharing the boundary node* B*.* □

2.1.2 Data Assembly

The process of partial evaluation and assembly in distributed graph query processing typically involves two key steps: first, computing local partial matches within each fragment; and second, *assembling* these partial matches to derive the final crossing matches. Two main strategies are commonly used for the assembly phase: centralized assembly and distributed assembly.

Centralized Assembly. This is the simpler of the two strategies. In this approach, all local partial matches generated by each graph fragment are sent to a single designated machine (the coordinator), where they are combined to compute the final crossing matches.

Distributed Assembly. Also known as *parallel assembly*, this strategy aims to distribute the workload of joining partial matches across multiple nodes. Instead of a single coordinator, partial matches are exchanged and combined across multiple fragments in parallel, typically using a communication model such as Bulk Synchronous Parallel (BSP) [158].

2.2 From Cypher queries to Graph patterns

It is common to translate user-friendly queries from the ASCII syntax into a formal structure in order to make easy parsing and handling of these queries [159]. Our solution relie on formal queries (i.e. graph patterns) to make an abstraction on the commercial graph query language used (e.g., Cypher). Hence, we explain hereafter the

translation of Cypher queries into graph patterns.

Given a Cypher query C with an optional Where clause. When the graph data G is distributed, some matches of C in G may arise in different machines instead of a single one. That is, it is natural to extract matches from all machines, merge them in a (de)centralized manner, and finally filter them according to the conditions of the Where clause. In order words, the checking of the Where clause can be postponed into a later stage until matches of all patterns are retrieved from the different machines. For this reason, it is sufficient to map Cypher queries into graph patterns without paying attention to the Where clauses which have no representation in our formal definition of graph patterns.

For any Cypher query C, we denote by C^* its simplified version without the Where clause. Then, C^* can be easily translated into a graph pattern $Q_c = (V_c, E_c, L_c, A_c, \vartheta)$ as follows:

1. For each vertex $(v{:}l\{\sigma\})$ defined in C^* with label l and an optional σ, create a pattern node n in V_c with: $L_c(n){=}l$ and $\vartheta(n){=}v$. Moreover, if σ is not empty then add $A_c(n) = \sigma$;

2. For each relationship $[r : l]$ defined from vertex $v1$ into vertex $v2$ in C^* with an optional σ, create an edge $e = (n1, n2)$ in E_c from node $n1$ into node $n2$ such that: node $n1$ (resp. $n2$) corresponds to vertex $v1$ (resp. $v2$); $L_c(e) = l$ and $\vartheta(e) = r$. Moreover, if σ is not empty then add $A_c(e) = \sigma$;

Therefore, the graph pattern Q_c encapsulates all paths of C with their vertices, relationships and attribute-based conditions defined over them. Q_c is used to show formally how our framework extracts the matches of all patterns of C in a distributed context. Once these matches are retrieved, the initial Where clause of C is used to filter them. Notice that an advanced definition of graph patterns is given in [160] which considers conditions over nodes and edges of the graph pattern. However, this definition is not necessary for this work as the Where clause is checked in a later stage.

3 Our Distributed GPM Framework

We first explain the problem of *partial matches* and our solution to deal with it. Next, we introduce our algorithms to answer graph queries over distributed data.

3.1 Partial Matches & Query Fragmentation

Given a distributed graph data $G^d = (P_1, \cdots, P_k)$ and a graph pattern Q. Recall that partitions of G^d may contain common nodes, called *boundary nodes*, which makes some matches of Q belong to different machines. These matches must be extracted in efficient manner without increasing network overhead. Precisely, given two partitions P_i and P_j of G^d and a boundary node v between them. Partial matching across P_i and P_j aims to find partial matches, t_i from P_i and t_j from P_j, so that the cartesian product of t_i and t_j on the boundary node v gives a complete match of Q in G. A match t of Q is said to be *partial* if some of its variables have no mappings. For instance, the match $t = \langle v1 = id1, v2 = \mathsf{null}, v3 = id3 \rangle$ is partial as the variable $v2$ has no mapping. As outlined in [153], the goal of distributed *GPM* is to extract complete matches from each partition, partial matches across all partitions, and to merge them in order to get all matches of Q in G. Complete matches are obtained simply by executing Q over each partition $P_i \in G^d$. However, extracting partial matches requires a non-trivial process. To this end, we propose the notion of Boundary Subpattern (*BSP*) as follows:

Definition 20. *Given a graph pattern Q and a set $\mathcal{BL}$ of boundary node labels. A* boundary subpattern *of Q w.r.t $\mathcal{BL}$ is any* subpattern $Q^s \subseteq Q$ *such that:*

1. *for any pair of pattern nodes (u, u') in Q^s, u and u' can be connected via a simple edge or an undirected path that contains no* boundary pattern node *(i.e., a pattern node of Q^s with label in $\mathcal{BL}$);*
2. *there exists no* BSP Q^l *with $Q^s \subseteq Q^l$.* □

Informally, a *BSP* Q^s formalizes a *possible* structure of partial matches. Contrary to complete matches, partial matches should be extracted locally over each partition and do not require computation across different machines. This is ensured by condition (1) which specifies that matches of all edges of Q^s can be extracted locally. In other words, if the pattern nodes u and u' of Q^s are connected only through a boundary pattern node w, then some matches of Q^s may be computed across different partitions connected via w. That is, condition (1) ensures that Q^s looks for partial matches with one possible structure. Condition (2) ensures that Q^s is as large as possible and allows to extract maximum partial matches from the same partition. As a special case, if $\mathcal{BL}$ is empty (i.e., partitions are all disjoint) then Q is itself the only *BSP*.

Example 15. *Consider a graph pattern Q represented simply by its edges with:*

Algorithm 3 Algorithm for discovering *BSPs*

Algorithm: **BSP_Discover**$(Q, \mathcal{BL})$
Input: A graph pattern $Q = (V_q, E_q, L_q, A_q, \vartheta)$ and a set of boundary node labels.
Output: A set S_Q of all *BSPs* in Q.

```
1: S_Q := ∅;
2: for each unmarked edge e from E_q do
3:     Initialize a subpattern Q^s with edge e;
4:     Mark e;
5:     for each pattern node u in Q^s with L_q(u) ∉ BL do
6:         for each unmarked edge e' in Q starting/ending at u do
7:             Mark e'; Extend Q^s with e';
8:         end for
9:     end for
10:    S_Q := S_Q ∪ {Q^s};
11: end for
12: Return S_Q;
```

$$B \leftarrow A \rightarrow \boldsymbol{C} \leftarrow D \rightarrow E$$

If C is a boundary pattern node then two BSPs *are given: $B \leftarrow A \rightarrow C$ and $C \leftarrow D \rightarrow E$. Notice that the subpattern $B \leftarrow A$ is not a* BSP *as it is contained in a larger one.* □

Notice that there may be different *BSPs* at the same graph pattern. Thus, our goal is to extract all of them in order to get all possible partial matches. This is done via our algorithm BSP_Discover, given in Algorithm 3. The main idea of the algorithm is to take an arbitrary edge e from Q (line 2) and to extract the largest *BSP* based on it (lines 3–11). The *BSP* is initialized to be a simple subpattern Q^s containing only the edge e (line 3). Next, this Q^s is iteratively extended by adding edges from Q that satisfy condition (1) of Definition 20. Precisely, for each node u in Q^s which is not a boundary pattern node w.r.t $\mathcal{BL}$, all its adjacent edges are added to Q^s (lines 5-10). This simple condition aims to avoid reachability between two nodes of Q^s via only a boundary pattern node. In other words, if some pattern node w from Q is connected to u only via a boundary pattern node, then w is not added to Q^s. Once this extension is finished, i.e., no other edges to add into Q^s, Q^s becomes the largest *BSP* that contains the initial edge e and it is added to the set S_Q of *BSPs* (line 11). Remark that each edge added to Q^s is marked (lines 4,7), which yields to *BSPs* that do not share any edge. The whole process (lines 2–12) is repeated to find all remaining *BSPs* in Q in the same manner. Finally, the algorithm returns the set S_Q of all possible *BSPs* in Q (line 13).

Example 16. *Consider the graph pattern Q of Figure 18 (a) where B and C correspond to labels of boundary nodes. Suppose that algorithm* BSP_Discover *initializes first Q^s to contain the edge $E \rightarrow B$. Then, it will extend Q^s iteratively by adding the edge $D \rightarrow E$*

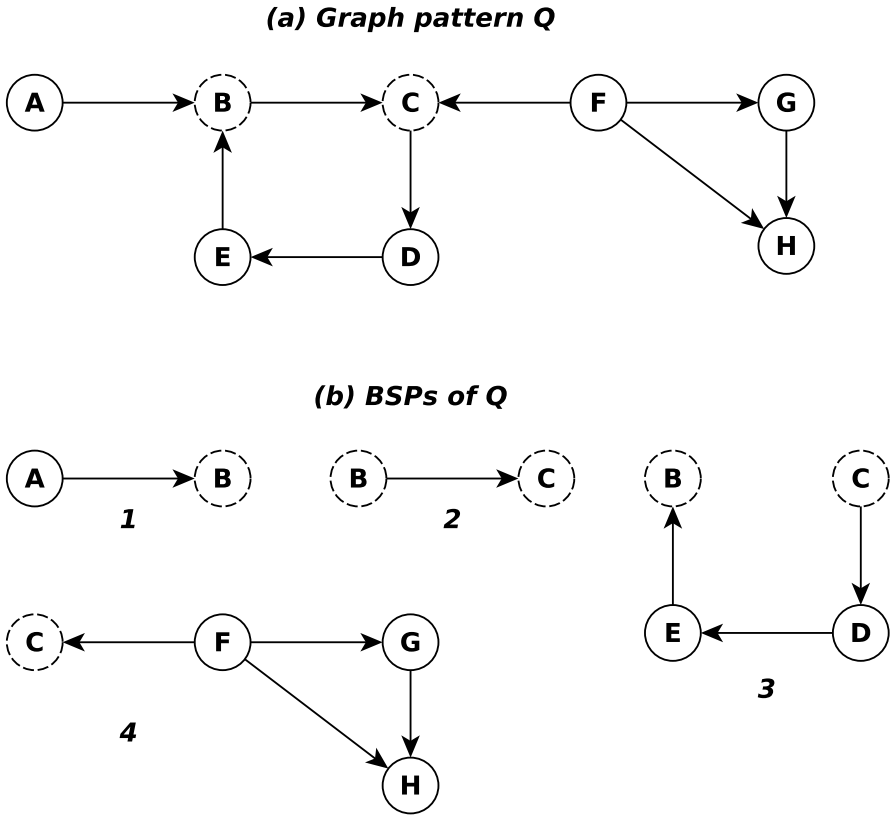


Figure 18: Example of *BSPs* discovering.

and next the edge $C \to D$. *The extension of* Q^s *terminates by giving the* BSP*:* $C \to D \to E \to B$. *Figure 18 (b) depicts the* 4 *possible* BSPs *of* Q. □

Notice that, for a graph pattern Q, the *BSP* set S_Q represents all possible structures of partial matches of Q that can be retrieved at graph partitions. The goal is to merge all these partial matches in order to form complete matches.

Property 1. *Given a graph data* G *and a graph pattern* Q *with edge set* E_Q. *Algorithm* BSP_Discover *returns a set of* BSPs S_Q *with* $|S_Q| \leq |E_Q|$. □

Based on the edge marking of algorithm BSP_Discover (lines 4,7), each edge in Q is added into one and only one *BSP*, then the number of *BSPs* of any graph pattern Q is bounded by its number of edges[1].

3.2 Computation Model

Given a distributed graph data $G^d = (P_1, \cdots, P_k)$ where each partition P_i is placed at a separate machine M_i (i.e., the number of partitions is the same as that of machines).

[1] BSP_Discover runs in $O(m)$ time with m query edges.

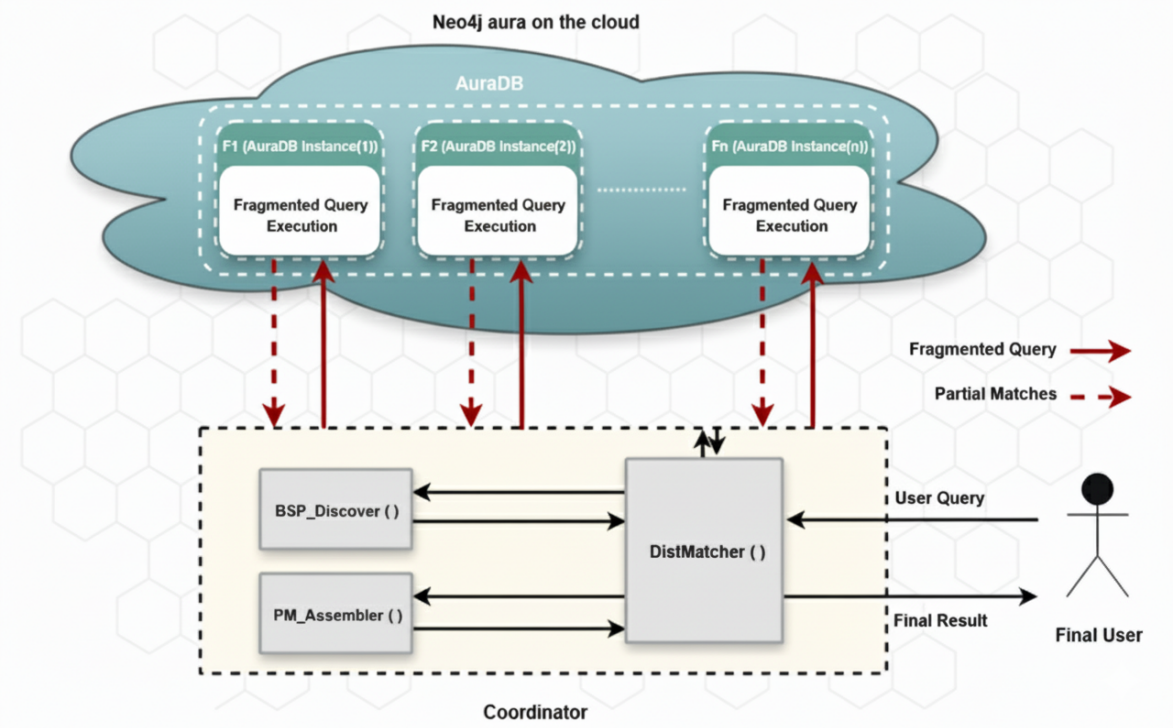


Figure 19: Architecture of our distributed *GPM* framework

A *coordinator* is a special machine responsible for sending tasks to different machines, receiving and handling their responses. Firstly, the user initiates a graph query C following a commercial query language (e.g. Cypher). The coordinator translates C into a graph pattern Q and starts the process of distributed *GPM*. Q is then sent to all machines of the cluster. Each machine M_i computes partial matches via *edge isomorphism* over its partition P_i and returns back these matches to the coordinator. The coordinator collects and merges partial matches in a centralized and parallelized manner in order to get all matches of Q in G^d.

Figure 19 represents our distributed framework showing a fragmented data graph where each fragment is hosted at a separate Neo4J AuraDB instance (i.e. a node on the cloud), and a coordinator machine that sends queries to the different machines, receives partial results and extracts complete results from them. The algorithms that run at the coordinator machine will be explained later.

Our model operates without requiring inter-machine communication. As outlined in [153], this model reduces *visit time*[1] and *data shipment*[2] while it may increases

[1] *Visit time*: machine visiting times at the cluster.
[2] *Data shipment*: amount of messages shared between machines.

makespan[1]. Precisely, size of partial matches may be huge which requires a high merging time at the coordinator. To reduce this time, we propose a parallelized merging algorithm. In addition, it has been remarked earlier that even simple graph queries may yield millions of matches since graph data are very large. This pushed researchers to think about special queries that retrieve less results and execute quickly (e.g., *diversified and top-k queries* for property graph data [151, 152, 161] and RDF data [162]). Our computation model, based on centralized data assembly, is very suitable for this kind of queries which are receiving more attention form the research community and allow to reduce makespan.

While the decentralized model reduces makespan (data assembly time), it requires inter-machine communication via *message passing*, increasing data shipment and visit time. However, most of cloud services (e.g. *AuraDB* of *AWS*) prohibit such communication, forcing companies to develop custom distributed *GPM* solutions (e.g., *GraphScope* [122]) with decentralized assembly. As [163] highlights, many companies cannot afford the high costs of such implementations. Our computation model leverages affordable cloud services to enable efficient distributed *GPM* with lower expenses.

3.3 Algorithm

We propose a distributed *GPM* with a centralized assembly of data. Our algorithm, referred to as DistMatcher, is given in Algorithm 4 and follows our computation model discussed in the previous section. It consists of three phases as follows.

Phase 1: Firstly, the Cypher query C is issued at the coordinator over the distributed graph data $G^d = \{P_1, \cdots, P_k\}$. It is then simplified into C^* by simply removing its Where clause in order to get its equivalent graph pattern Q. Next, the *BSP* set S_Q is extracted using our algorithm BSP_Discover. Recall that S_Q is extracted based on the boundary node labels $\mathcal{BL}$ and each *BSP* from S_Q, i.e., $S_{1\leq j\leq m}$, is a graph subpattern representing one possible structure of partial matches that can be detected over partitions of G^d. These partial matches should be extracted over all k machines. However, the subpatterns in S_Q are formal and cannot be executed over a real graph database system. That is why the coordinator translates each $S_j \in S_Q$ into a Cypher query C_j. Finally, the coordinator sends C^* and the Cypher query set $\{C_1, \cdots, C_m\}$ to all k participating machines so they can compute complete and partial matches locally.

[1] *Makespan*: total *GPM* time from query to matches completion.

Algorithm 4 Algorithm for distributed *GPM*.

Algorithm: **DistMatcher**$(C, G^d, \mathcal{BL})$
Input: A Cypher query C, a distributed graph data $G^d = (P_1 \cdots, P_k)$ where each partition P_i is placed at machine M_i, and a set of boundary node labels $\mathcal{BL}$.
Output: The match result $C(G^d)$ of C in G^d.

COORDINATOR:
Phase 1: ***Query transformation and broadcasting***
Let C^* be the simplified version of C by removing Where clause;
$Q :=$ ***CypherToPattern***(C^*);
$S_Q :=$ BSP_Discover$(C^*, \mathcal{BL})$;
for each *BSP* $S_{1 \leq j \leq m} \in S_Q$ **do**
    $C_j :=$ ***PatternToCypher***(S_j);
end for
Send C^* and $\{C_1, \cdots, C_m\}$ to all participating k machines;

Phase 3: ***Matches merging and filtering***
Let $\mathcal{M}^*$ be the complete match result of C^* on G^d;
Receive complete and partial matches from all k machines;
Let $\mathcal{M}^* := \bigcup_{1 \leq i \leq k} \mathcal{M}_i^*$;
for each *BSP* $S_{1 \leq j \leq m} \in S_Q$ **do**
    $\mathcal{M}^j := \bigcup_{1 \leq i \leq k} \mathcal{M}_i^j$;
end for
$\mathcal{M}^* := \mathcal{M}^* \cup$ PM_Assembler$(\{S_1, \cdots, S_m\}, \{\mathcal{M}^1, \cdots, \mathcal{M}^m\}, \mathcal{BL})$;
Let $\mathcal{M} := \emptyset$ /* *the final match result of* C *on* G^d */;
for each *match* $t \in \mathcal{M}^*$ **do**
    if (t satisfies Where clause of C) **then**
        $\mathcal{M} := \mathcal{M} \cup \{t\}$;
    end if
end for
Return $\mathcal{M}$;

MACHINE M_i**: (AURA DB INSTANCE)**
Phase 2: ***Total and partial matches extraction***
Receive C^* and $\{C_1, \cdots, C_m\}$ from the coordinator;
Let $\mathcal{M}_i^*$ be the complete match result of C^* in partition P_i;
for each Cypher query $C_{1 \leq j \leq m}$ **do**
    Let $\mathcal{M}_i^j$ be its match result in the partition P_i;
end for
Send $\{\mathcal{M}_i^*, \mathcal{M}_i^1, \cdots, \mathcal{M}_i^m\}$ to the coordinator;

Phase 2: Once a machine M_i receives C^* and the Cypher query set $\{C_1, \cdots, C_m\}$, it first computes the complete matches of C^* over the local partition P_i (i.e., the match result $\mathcal{M}_i^*$). Similarly, each Cypher query $C_{1 \leq j \leq m}$ is executed locally to obtain its match result M_i^j. This match result represents partial matches of the initial Cypher query C^*. The machine M_i finishes its work by simply sending back to the coordinator the complete match result M_i^* as well as the set $\{M_i^1, \cdots, M_i^m\}$ containing partial matches of different structures obtained all over the partition P_i.

Phase 3: Once all machines finish their work, the coordinator receives complete and partial matches extracted locally at each partition P_i for the Cypher query C^*. All complete matches are merged into the set $\mathcal{M}^*$. Moreover, for each possible structure S_j of partial matches, the coordinator receives a match set $\mathcal{M}_i^j$ of this structure from each machine M_i. The set $\mathcal{M}^j$ is computed by merging partial matches from all machines that correspond to the same structure S_j. These partial matches have m different

Algorithm 5 Algorithm for parallel and centralized assembly of partial matches.

Algorithm: PM_Assembler$(\{S_1, \cdots, S_m\}, \{\mathcal{M}^1, \cdots, \mathcal{M}^m\}, \mathcal{BL})$
Input: The *BSP* set S_Q, match result $\mathcal{M}^j$ (i.e., partial matches) of each subpattern $S_j \in S_Q$ (i.e., possible structure), and a set of boundary node labels $\mathcal{BL}$.
Output: A set $\mathcal{MM}$ of complete matches obtained by merging partial matches.

1: **Master Thread:**
2: Reorder subpatterns in S_Q such that: S_j and S_{j+1} must share vertex label(s) between them;
3: Send the subpatterns in S_Q with their corresponding matches into a parallel merger and obtain the merge result $\mathcal{M}$;
4: Return $\mathcal{M}$;

5: **Slaves Thread:**
6: Receive a subpattern set $\{S_1, \cdots, S_l\}$ with the corresponding matches $\{\mathcal{M}^1, \cdots, \mathcal{M}^l\}$;
7: **if** $(l = 1)$ **then**
8: Return $\mathcal{M}^1$;
9: **else if** $(l = 2)$ **then**
10: Return $\mathcal{M}^1 \bowtie \mathcal{M}^2$;
11: **else**
12: Divide S into two subsets:
13: $S^1 = \{S_1, \cdots, S_{l/2}\}$ and $S^2 = \{S_{l/2+1}, \cdots, S_l\}$;
14: Send the subpatterns in S^1 with their matches to a parallel merger and obtain the merge result $\mathcal{M}^1$;
15: Send the subpatterns in S^2 with their matches to another parallel merger and obtain the merge result $\mathcal{M}^2$;
16: $\mathcal{M} := \mathcal{M}^1 \bowtie \mathcal{M}^2$;
17: Return the result $\mathcal{M}$ to the parent thread;
18: **end if**

structures and must be merged to obtain complete matches of the Cypher query C^*. This task is ensured by the algorithm PM_Assembler that takes in input all m possible structures with their corresponding matches, and extracts from them complete matches of C^*. The new obtained matches are simply added into the set $\mathcal{M}^*$. At this stage, the set $\mathcal{M}^*$ contains all complete matches of C^* extracted locally and across machines. The final step of the algorithm consists in filtering matches in $\mathcal{M}^*$ to retain only those that satisfy the Where clause of C. In this manner, the match result of the original Cypher query C is extracted and returned to the user.

3.4 Centralized Assembly of Matches

Algorithm 5 presents PM_Assembler, our algorithm for the parallel and centralized assembly of partial matches. Given a set $\mathcal{M}^1, \cdots, \mathcal{M}^m$ of partial matches computed across different machines, along with their corresponding structures $S_1, \cdots, S_m$. The algorithm begins by determining an ordering of these structures that enables parallel merging of matches. Specifically, each subpattern S_j in S_Q must be followed by a subpattern S_{j+1} that shares at least one vertex label with it. This shared label allows the matches of the two subpatterns to be merged. Next, the algorithm follows the *Divide and Conquer* paradigm in a parallel fashion. In summary, each thread divides the input subpattern set S into two subsets, S_1 and S_2. The matches of S_1 (resp. S_2) are merged in parallel by invoking additional threads, producing the sets $\mathcal{M}^1$ (resp. $\mathcal{M}^2$). The current thread then merges these two intermediate results (i.e., $\mathcal{M}^1$ and $\mathcal{M}^2$)

Table 8: Partitioning of the *StackExchange* graph data.

	Fragment					
	F1	F2	F3	F4	F5	F6
Nodes	163,444	194,787	162,559	161,268	152,560	137,993
Relationships	167,113	171,055	179,142	147,832	161,074	143,375

into the set $\mathcal{M}$, which is sent back to its parent thread. In this way, the merging of all matches $\mathcal{M}^1, \cdots, \mathcal{M}^m$ is performed in parallel and returned as the final result. A naive merge of match sets of sizes n_1 and n_2 has $O(n_1 \times n_2)$ complexity and can produce a set of that size. Merging m sets this way escalates to $O(n_1 \times \cdots \times n_m)$ time. Our *Divide and Conquer* algorithm is designed to reduce this merging time significantly through parallel processing.

4 Experimental study

We conduct experimental study based on real-life data in order to check effectiveness of our approach.

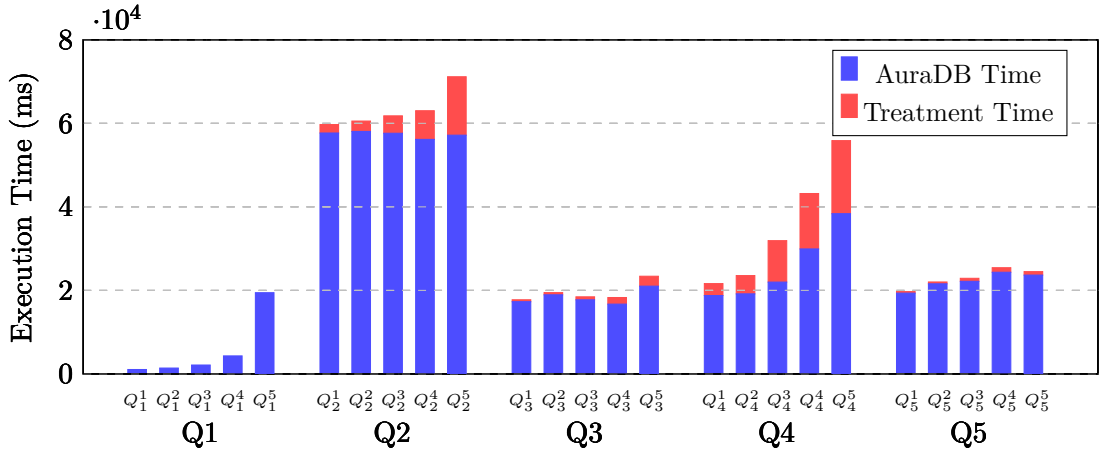
General Settings. The experiments were conducted on a workstation (coordinator) with an Intel(R) $i7-12700K$ CPU, $32GB$ of memory and $512GB$ of SSD storage. The distribution is ensured by *Neo4j AuraDB*: a fully managed cloud graph database service. We created different instances of AuraDB to simulate separated machines, each one has $1GB$ of memory and stored a partition of the underlying graph data.

Real-life graph data. We evaluated our approach using the real-world *StackExchange* graph data[1]. The graph data contains 972520 nodes and 969500 relationships. We partitioned this graph into 6 fragments using the *vertex-cut strategy*, resulting in 91 boundary nodes across fragments. Table 8 provides some details of our fragments.

Query workload. We used real Cypher queries expressed over the schema of the *StackExchange* data. We initially designed 4 path queries of different lengths (queries $Q1-Q4$) and one cyclic query ($Q5$). As our approach supports expressive queries with complex conditions, we generated 5 extensions of each query $Q_{1\leq i\leq 5}$ (denoted by $Q_i^1, \cdots, Q_i^5$) by varying values of its attribute conditions over its nodes and relationships as is given in Appendix 3. In this way, we get a concrete evaluation of our approach

[1] https://archive.org/download/stackexchange

Figure 20: Efficiency checking of our distributed *GPM* framework.



by varying the query features.

4.1 Correctness Checking

We validated our distributed *GPM* approach by comparing its results for 25 queries against a centralized baseline. Both the total number of matches and a tuple-by-tuple comparison confirmed perfect equivalence, proving that there was no data loss. We note that the order of results differs, but this is irrelevant for correctness, as it varies even between different database systems.

4.2 Efficiency Checking

We execute each Cypher query Q_i^j $(i, j \in [1, 5])$ over the distributed graph data and report the total answering time in Figure 20. For each query, the answering time is divided into two parts: a) time consumed by the 6 machines for complete and partial matches extraction; and b) time consumed by the coordinator to merge and filter matches. Table 9 reports the number of matches returned by each Cypher query.

Firstly, we remark that the time consumed by the coordinator is negligeable in case of small result set while it becomes significant in case of large result set. For instance, consider the two queries Q_4^1 (with 6219 matches) and Q_4^5 (with 30831 matches). Remark in Figure 20 that the time of the machines increase slightly between Q_4^1 and Q_4^5, while the time taken by the coordinator becomes six times higher. This suggests to use a more sophisticated machine as a coordinator and to apply optimized techniques for data assembly (e.g., parallelization, join order optimization, and pre-filtering based on the WHERE clause). Secondly, the time consumed by the machines grows proportionally w.r.t the query complexity. For instance, from $1056ms$ consumed for the simple path

Table 9: Size of match result per query.

Q_1^1 (1154)	Q_1^2 (2254)	Q_1^3 (4646)	Q_1^4 (12273)	Q_1^5 (62740)
Q_2^1 (20939)	Q_2^2 (30874)	Q_2^3 (49493)	Q_2^4 (83857)	Q_2^5 (150065)
Q_3^1 (1128)	Q_3^2 (1591)	Q_3^3 (2697)	Q_3^4 (5588)	Q_3^5 (7484)
Q_4^1 (6219)	Q_4^2 (9983)	Q_4^3 (19366)	Q_4^4 (23030)	Q_4^5 (30831)
Q_5^1 (1248)	Q_5^2 (1513)	Q_5^3 (1828)	Q_5^4 (2519)	Q_5^5 (2832)

query Q_1^1 (with 1154 matches) to $18771ms$ consumed for the complex path query Q_4^1 (with 6219 matches). Finally, the scalability of our approach has been checked by varying the number of returned matches of each query in order to check whether a huge data extraction, transfer, merging and filtering will reduce efficiency. For each of the 5 queries, Q_i^j returns fewer matches than Q_i^{j+1}. Notice that the answering time increases slightly between Q_i^j and Q_i^j $(i, j+1 \in [1,5])$, demonstrating the scalability of our approach.

4.3 Comparison with Neo4j single-machine processing

A direct comparison with a single Neo4j machine was impossible because the complex queries on our large data graphs caused it to freeze. This demonstrates the necessity of our distributed framework, which successfully executes these queries in a reasonable time.

Our Findings. 1) Our distributed framework has worked correctly with real-life setting (i.e. Cypher queries, real graph data, and cloud machines). 2) Even with a limited configuration of AuraDB instances ($1GB$ of memory) and only 6 machines, the total answering time of our approach remains acceptable. For instance, the query Q_2^5 returns a high number of matches (150065) while it takes only 0,9527 minute. This tells that more instances of AuraDB with a higher configuration will significantly reduce the total answering time. 3) The coordinator takes more time for queries returning large result set, however, users are often interested by only few relevant results, and not all of them. Therefore, the emerged *diversified top-k* queries [151, 152] would work very well using our framework. 4) Our distributed *GPM* scales well with the increasing number of returned matches.

5 Conclusion

In this chapter, we proposed a formal distributed *GPM* framework that can work for arbitrary graph database systems. The formalization is ensured through non-trivial algorithms that can be easily implemented, compared to other approaches, or even extended in the future. As a use case, the framework has been successfully implemented based on AuraDB instances (the Neo4j cloud database solution) and using the Cypher query language. The experimental demonstrated the effectiveness of our framework over real commercial solutions (Neo4j-AuraDB and Cypher) and real graph data.

Chapter IV: R2G-ETL Tool

1 Introduction

For decades, relational databases have served as the foundation of data management, widely adopted in both academia and industry due to their intuitive design, strong data integrity, reduced redundancy, and standardized querying via SQL[1]. However, the emergence of Big Data characterized by massive volumes of highly interconnected data (e.g. social networks have billions of linked entities [131]) has revealed significant limitations of the relational database model. Primarily, complex relationships between entities are hardly represented and analyzed within a relational database, resulting in convoluted SQL queries and prohibitive computational costs [132]. Graph databases have emerged as a powerful alternative, gaining prominence over the past decade due to their ability to natively represent and query interconnected data. Graph database systems (e.g. Neo4j[2]) excel in applications such as social network analysis, fraud detection, and recommendation engines. They enable intuitive relationship visualization, efficient traversal, and advanced analytical algorithms capabilities that are cumbersome or inefficient in relational frameworks [4].

To leverage these advantages, the mapping of relational databases to graph databases has received increasing attention in recent years [111, 112, 113, 114, 118]. Existing mapping approaches face critical challenges: they consider only partial database elements (e.g., neglecting schemas and constraints), inadequately translate queries and do not consider updates, rely on impractical theoretical models, or distort the original schema. Moreover, no approach has studied all the mapping properties. For instance, our evaluation of the *Neo4j-ETL*[3] revealed structural inconsistencies with source databases and inefficient processing times.

2 R2G-ETL Presentation

The GUI of the *R2G-ETL* tool is shown in Figure 21. The modules of *R2G-ETL* are developed with Java while its GUI is developed with JavaFX. It leverages the latest database APIs versions to ensure compatibility with major database management systems. The following are the features of *R2G-ETL* as shown in Figure 21:

1. An interface to configure a connection to a relational database system (e.g. MySQL, PostgreSQL, Oracle);

[1] https://db-engines.com/en/
[2] https://neo4j.com/
[3] https://neo4j.com/developer/neo4j-etl/

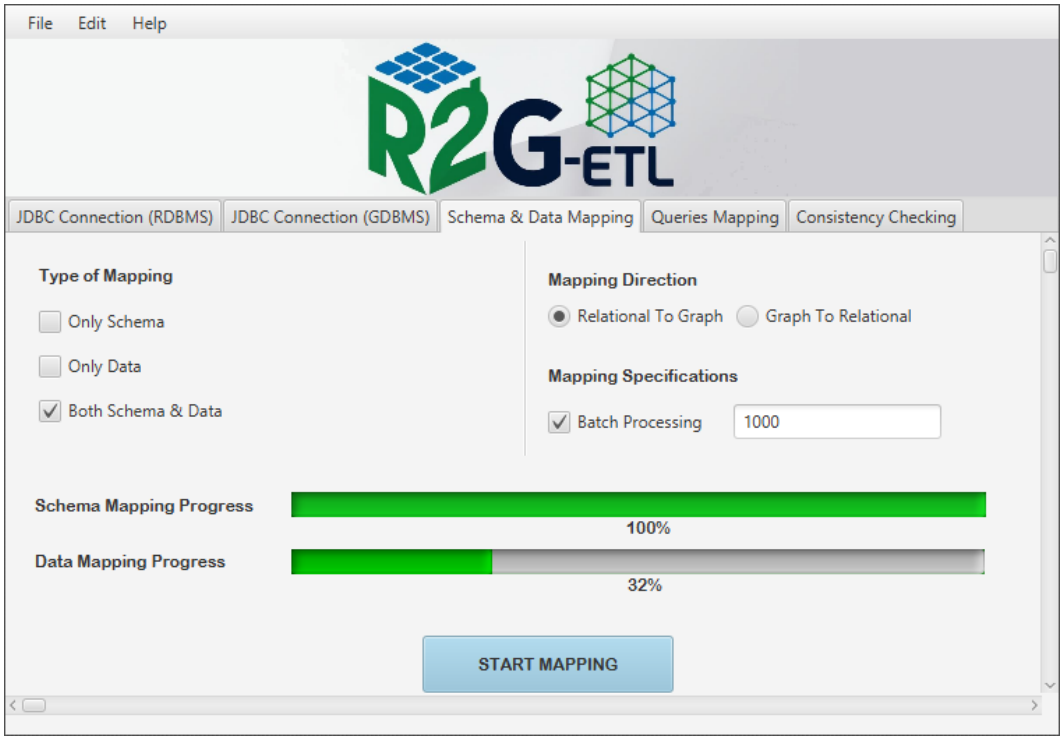


Figure 21: The *R2G-ETL* GUI.

2. An interface to configure a connection to the Neo4j graph database system via the Bolt protocol;

3. An interface that allows mapping of schemas (including constraints) and data between relations and graphs (in any direction);

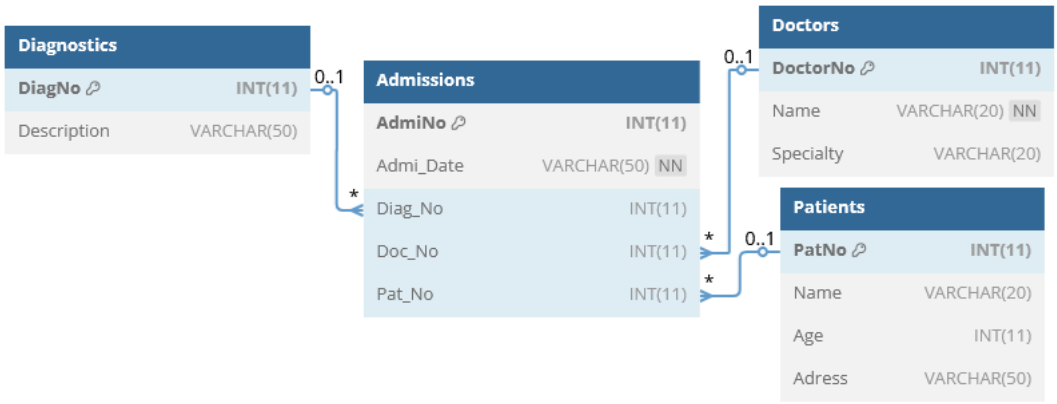
4. An interface that allows the user to write an SQL query (resp. update) and shows its equivalent Cypher query (resp. update);

5. An interface that allows the user to check whether a graph data is consistent w.r.t its schema graph, ensuring that all integrity constraints (e.g., *not null*, *unique*, *check*) defined in the schema are satisfied.

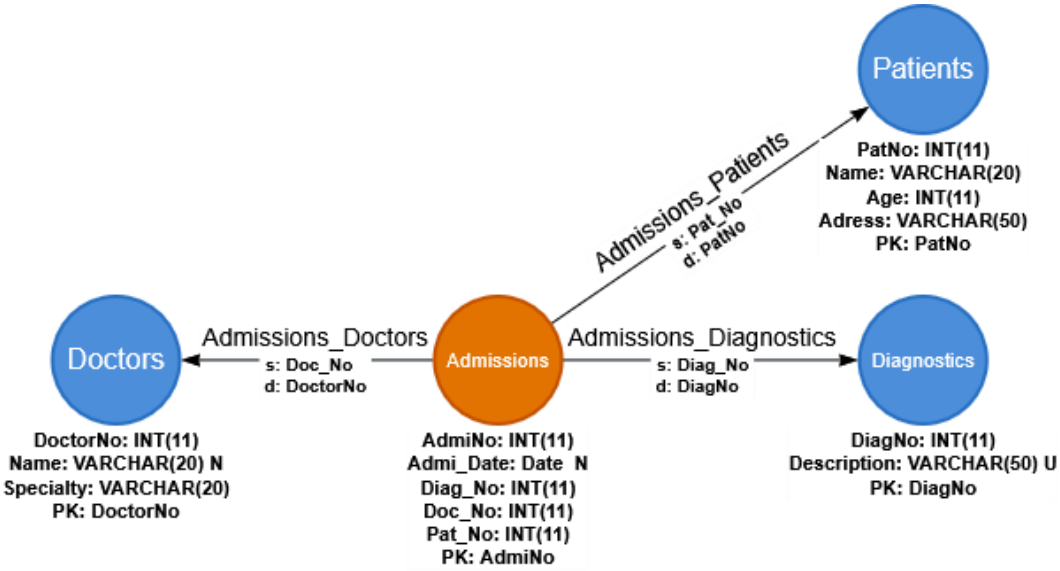
Once the mapping is completed by *R2G-ETL*, two graphs are created and stored within Neo4j: schema graph and data graph. The user can visualize these graphs via Neo4j and access/modify them via Cypher queries.

2.1 Schema and Constraints Mapping

Figure 22 illustrates an example of a schema mapping done by *R2G-ETL*. Figure 22(a) shows a portion of a relational schema where each relation name has primary and/or foreign keys, and attributes with data types. Figure 22(b) presents its



(a) Relational Schema



(b) Schema Graph

Figure 22: Output example of *R2G-ETL* schema mapping.

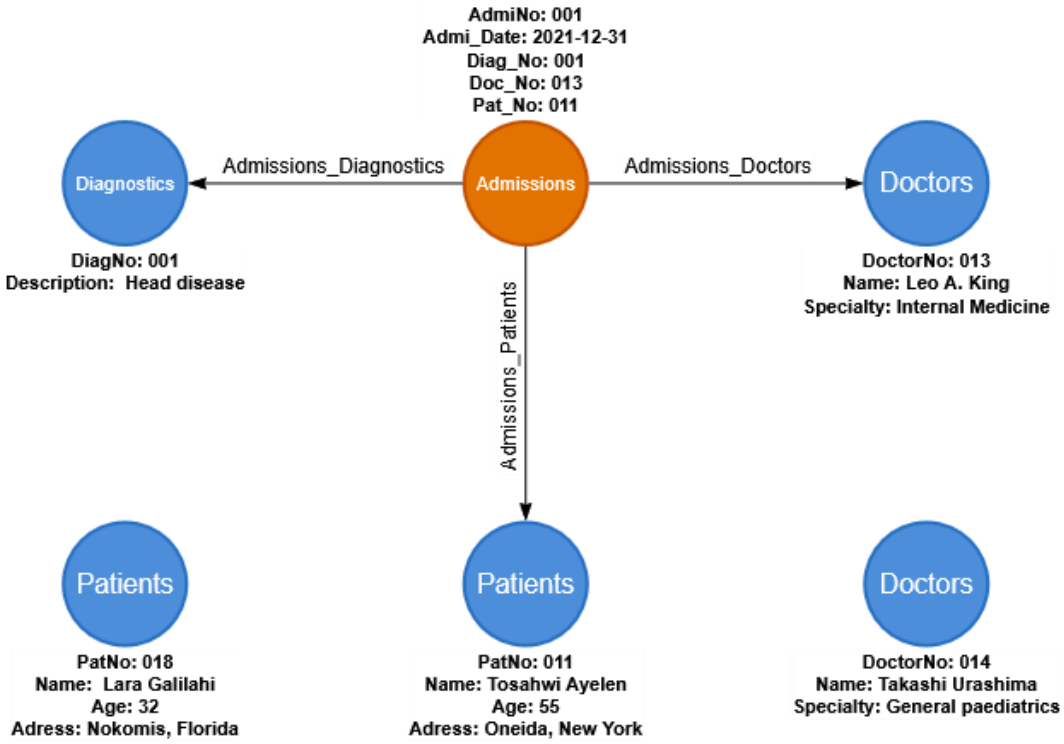
	AdmiNo [PK] integer	Admi_Date character varying	Diag_No integer	Doc_No integer	Pat_No integer
1	1	31-12-2021	1	13	11
*					

	DiagNo [PK] integer	Description character varying
1	1	Head disease
*		

	DoctorNo [PK] integer	Name character varying	Specialty character varying
1	13	Leo A. King	Internal Medicine
2	14	Takashi Urashima	General paediatrics
*			

	PatNo [PK] integer	Name character varying	Age integer	Adress character varying
1	11	Tosahwi Ayelen	55	Oneida, New York
2	18	Lara Galilahi	32	Nokomis, Florida
*				

(a) Relational Data



(b) Graph Data

Figure 23: A data mapping example via *R2G-ETL*.

```
SQL :
SELECT * FROM Admissions AS T1, Doctors AS T2
WHERE T1.Doc_No = T2.DoctorNo AND T1.Admi_Date < 01-01-2024;
```

```
CYPHER :
MATCH (a:Admissions)-[:Admissions_Doctors]->(d:Doctors) WHERE a.Admi_Date < 01-01-2024
RETURN *;
```

Figure 24: A query mapping example via *R2G-ETL*.

corresponding schema graph where: node represents a relation name (e.g. *Admissions* and *Doctors*) and attributes of this node represent attributes of the source relation name with their data types. Moreover, an edge represents a foreign key between two entities.

Therefore, *R2G-ETL* maps any relational schema into a schema graph by preserving the structure of the relational schema, its data types and its constraints.

2.2 Data Mapping

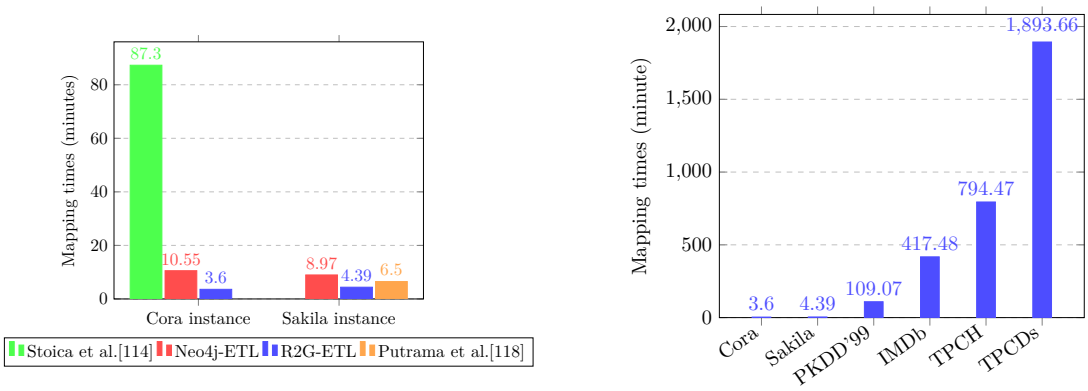
Figure 23 illustrates an example of a data mapping done by *R2G-ETL*. Figure 23(a) displays a sample relational data, while Figure 23(b) shows the equivalent graph representation of this data.

2.3 Query/Update Mapping

A key feature of *R2G-ETL* is its sophisticated query mapping engine, which automatically converts SQL queries and updates into semantically equivalent Cypher operations. This ensures that query logic remains consistent during migration from relational to graphs. Figure 24 illustrates a concrete example, mapping an original SQL query into an equivalent Cypher query.

2.4 Consistency checking

In order to check whether a graph data G satisfies all constraints of a schema graph S_G, *R2G-ETL* generates a Cypher query that takes all constraints of S_G in consideration. This query is then executed over G and any unsatisfaction is reported within the result.



(a) R2G-ETL v.s Other approaches.

(b) Instance Mapping Times of R2G-ETL.

Figure 25: Efficiency and Scalability checking of R2G-ETL.

3 R2G-ETL Evaluation

Using real-life and synthetic datasets[1], we have conductd extensive experimental study in order to verify the effectiveness, efficiency and correctness of *R2G-ETL*. We used three real-life datasets: CORA (57,353 Rows), PKDD'99 (1,090,086 Rows), IMDb (5,647,694 Rows)). Moreover, three synthetic datasets have been used: Sakila (47,273 Rows), TPCH (6,885,051 Rows), TPCDs (21,005,545 Rows). The experimental evaluation was conducted on a PC running Windows 11, equipped with an Intel Core i5-1135G7 processor (2.4 GHz base frequency), 8GB RAM, and 500GB SSD. For database management, we utilized MySQL 8.0.30 for relational operations and Neo4j Community Edition 5.2.0 for graph database functionality.

3.1 R2G-ETL Efficiency Checking

We mapped the two relational datasets Cora and Sakila using *R2G-ETL* and other approaches , and we reported the results in Figure 25(a). Notice that *R2G-ETL* outperforms all other existing academic (resp. commercial) tools.

3.2 R2G-ETL Scalability Checking

The times required by *R2G-ETL* to map six relational datasets are reported in Figure 25 (b). Clearly, the data mapping time of *R2G-ETL* increases proportionally with the scale of the relational data. The time required for mapping real-life data (e.g. IMDb) is still acceptable by taking into account the huge volume of this data and the fact that current mapping tools (e.g. *Neo4j-ETL*) take more time.

[1]https://relational.fit.cvut.cz

4 Conclusion

In this chapter, We proposed a tool called *R2G-ETL* that performs a complete and valid mapping from relational database into an equivalent graph database by considering both schema, constraints, data, queries and updates. *R2G-ETL* preserves many mapping properties, and maps any SQL query/update into an equivalent Cypher query/update.

General Conclusion and Further Directions

This thesis has tackled by the first the fundamental challenge of bridging the gap between relational and graph database paradigms. We moved beyond simple data conversion to establish a robust, complete and formal foundation for mapping the complete semantics of a relational database including its schema, constraints, data, and queries into an equivalent graph-based representation.

Our work began with a comprehensive analysis of existing models for formalizing database mapping, which revealed a common focus on data instance translation while often neglecting the critical aspects of schema integrity and query semantics. To address this gap, our first major contribution was the introduction of a **Complete Mapping (CM) process**. This process is anchored by a novel formal definition of a **schema graph** that captures relational structures and integrity constraints (e.g., unique, not null, check). We further formalized the concept of **graph consistency** to ensure instance graphs adhere to these constraints. A key strength of our approach is its demonstrable preservation of essential properties: information preservation, query preservation, semantic preservation, and monotonicity. We also extended this theory to include **update preservation**, guaranteeing the correct translation of data modification operations. A proof of concept has been realized by mapping both real-life and synthetic relational databases to graph databases, moving from PostgreSQL to Neo4j, that confirms the practical

effectiveness, efficiency, and scalability of our mapping methodology.

The second core contribution of this thesis is a **formal framework for distributed graph pattern matching (GPM)**. This framework is designed to work across any graph system that supports an implementation of the ISO-GQL (e.g, Cypher). Its formalization is realized through non-trivial yet practical algorithms that are readily implementable and extensible. As a proof of concept, we successfully implemented this framework using Neo4j's AuraDB and the Cypher query language. An extensive experimental study demonstrated the significant effectiveness and performance advantages of our framework compared to native single-machine processing solutions on real-world graph data from the StackExchange platform.

Finally, to synthesize our theoretical contributions into a practical tool, we designed and implemented **R2G-ETL**, an end-to-end software system that operationalizes our complete mapping process, automating the translation of relational databases into semantically equivalent graph databases. **R2G-ETL** comprehensively handles schema, constraints, data, queries, and updates, ensuring the preservation of the mapping properties established in our formal model.

Future Research Directions

The work presented in this thesis opens up several promising avenues for future research, which we have categorized according to our main contributions.

A. Enhancing the Complete Mapping (CM) Model:

1. **Extended Relational Feature Support:** We plan to incorporate a broader range of relational features, such as complex functional dependencies and other key types, to capture even deeper semantic

relationships during the mapping process.

2. **Access Control Mapping:** An important question is whether the access control policies of a relational system can be faithfully mapped into an equivalent policy for the resulting graph. This would allow administrators to maintain consistent security postures across the data migration by mapping relational access control into dedicated graph access control [149].

3. **Incremental Mapping:** To efficiently handle dynamic data, we will investigate an incremental mapping approach that propagates only the specific updates made to the source relational database to the target graph, rather than remapping the entire dataset.

4. **Scalable and Parallel Processing:** Addressing the time complexity of mapping large-scale databases, we aim to develop a parallel and distributed processing framework. This would feature an intelligent schema-based data partitioning strategy and sophisticated load-balancing algorithms to drastically reduce data mapping time.

5. **Uncertain and Fuzzy Data:** Exploring mappings for relational models that incorporate uncertainty or fuzziness would extend the applicability of our approach to a wider class of modern data problems.

6. **Specialized Data Challenges:** Future versions could address the mapping of complex data types, such as incomplete (missing) data and temporal data, which pose significant challenges for semantic preservation.

B. Advancing the Distributed GPM Framework:

1. **Diversified Top-k Queries:** An immediate extension is to integrate support for diversified top-k graph queries [151], which are increasingly important for applications requiring result variety and relevance.

2. **Adaptive Data Partitioning:** We aim to generalize the framework to support arbitrary data partitioning strategies, enabling dynamic optimization based on workload patterns and data characteristics.

We tried throughout this thesis to advance research in database interoperability and distributed graph processing by providing a formal and practical bridge between relational and graph paradigms. We hope our results may serve as a solid foundation for future work, both in refining the theoretical underpinnings of graph databases and in building more powerful and efficient data management systems. It is clear that graph database management is still in its first stage and many theoretical and practical foundations should be given; it is our sincere hope that the contributions of this thesis will form a meaningful part of that essential groundwork.

Appendices

1 Proof of Theorem 1

We prove Theorem 1 by providing a computable mapping $CM^{-1} : D_G \rightarrow D_R$ that reproduces the original relational database from the generated graph database. Similarly to the CM process, its inverse process CM^{-1} contains two parts: (1) SM^{-1} that reproduce the original relational schema from the generated graph schema; and (2) IM^{-1} that reproduces the original relational instance from the generated instance graph. We provide hereafter complete definitions of SM^{-1} and IM^{-1}.

SM^{-1} transformation rules. Given a schema graph $S_{_G} = (V_{_S}, E_{_S}, L_{_S}, A^a_{_S}, A^c_{_S}, Pk, Fk, N_{_S}, C_{_S}, U_{_S}, D_{_S})$, the reversed schema mapping SM^{-1} produces a relational schema $S_R = (R, A, T, \Sigma, N, C, U, D)$ as follows:

- For each vertex $v_r \in V_{_S}$, we create a relation $r \in R$ with name $L_s(v_r)$. We denote by v_r the vertex that corresponds to the relation r;
- For each pair $a \in A^a_{_S}(v_r)$ and $A^c_{_S}(v_r, a) = t$, we add the attribute a to $A(r)$ with $T(a) = t$;
- For each $Pk(v_r) = \{a_1, \cdots, a_n\}$, we add a *primary key* $r[a_1, \cdots, a_n]$ to the set Σ;
- For each edge $e = (v_r, v_s) \in E_{_S}$ with : (a) $L_{_S}(e) = s\_r$; (b) $Fk(e, s)$= $\{a_1, \cdots, a_n\}$; and (c) $Fk(e, d) = \{b_1, \cdots, b_n\}$, we add a *foreign key* $r[a_1, \cdots, a_n] \rightarrow s[b_1, \cdots, b_n]$ to the set Σ.
- For each $N_{_S}(v_r)$, we add a *not null* constraint $N(r)$ to $r \in R$;
- For each $C_{_S}(v_r)$, we add a *check* constraint $C(r)$ to $r \in R$;
- For each $U_{_S}(v_r)$, we add a *unique* constraint $U(r)$ to $r \in R$;
- For each $D_{_S}(v_r, a)$, we add a *default* attribute property $D(r, a)$ to each attribute $a \in A(r)$ in relation $r \in R$;

Notice that the special pair $(vid : Integer)$ is ignored by the mapping SM^{-1} since it has no equivalent part in the relational schema. However, it allows instance graphs to preserve the tuples identification.

IM^{-1} transformation rules. Given an instance graph $I_{_G} = (V_{_I}, E_{_I}, L_{_I}, A^a_{_I}, A^c_{_I})$, the reversed instance mapping IM^{-1} produces the original relational instance I_R as follows:

```
SQL :
    SELECT name,year FROM  movies  WHERE rank >= 8.0;
CYPHER :
    MATCH (m : movies)  WHERE m.rank >= 8.0 RETURN m.name,m.year;
```

Figure 26: Q_1 : Simple SQL query and its equivalent Cypher query

```
SQL :
    SELECT r.role,m.name,a.first_name,a.last_name
    FROM roles AS r , movies AS m ,actros AS a
    WHERE r.movie_id = m.id AND r.actor_id = a.id
    AND a.gender = 'M';
CYPHER :
     MATCH (r : roles) -[:roles_movies]->(m:movies)
     MATCH (r)-[:roles_actors]->(a:actors)
     WHERE a.gender = 'M'
     RETURN r.role,m.name,a.first_name,a.last_name;
```

Figure 27: Q_2 : Join SQL query and its equivalent Cypher query

- For each vertex $v_t \in V_I$ with $L_I(v_t) = r$, we create a tuple $t \in I(r)$.
- For each $a \in A_I^a(v_t)$, $A_I^c(v_t, a) = c$, we create a tuple $t \in I(r)$ and each attribute a with $t(a) = c$ and $a \neq tid$;

From the definition of IM mapping (Section 4.3), one can verify that for any two vertices $v_1, v_2 \in V_I$, if $L_I(v_1) = L_I(v_2)$ then v_1 and v_2 have different values for attributes vid. Based on this remark, tuples generated by IM^{-1} mapping satisfy the following condition: for any two tuples $t_1, t_2 \in I(r)$, $t_1(tid) \neq t_2(tid)$.

2 SQL queries and their corresponding Cypher queries

We give hereafter details of the five SQL queries, that are used in Sections 5 and 6, along with their equivalent Cypher queries.

3 Query Workload

We give hereafter details of the 25 Cypher queries, that are used in Section 4.

```
SQL :
     INSERT INTO movies (name,year,rank)
     VALUES ('Breaking bad',2008,9.6);
CYPHER :
      CREATE (:movies { name: 'Breaking bad', year: '2008', rank: 9.6 });
      MATCH (a:moviesdirectors) ,(b:movies)
      WHERE a.movie_id = b.id
      CREATE (a)-[:moviesdirectors_movies]->(b);
      MATCH (a:moviesgenres) ,(b:movies)
      WHERE a.movie_id = b.id
      CREATE (a)-[:moviesgenres_movies]->(b);
      MATCH (a:roles) ,(b:movies)
      WHERE a.movie_id = b.id
      CREATE (a)-[:roles_movies]->(b);
```

Figure 28: Q_3 : Insert SQL query and its equivalent Cypher query

```
SQL :
     UPDATE directors SET  id = 11450 , first_name = 'jack', last_name = 'adam'
     WHERE id = 2;
CYPHER :
    Match (l:directors) WHERE l.id = 2
    WITH l.id as id
    OPTIONAL MATCH (l)<-[oldedges:directorsgenres_directors]-(s:directorsgenres{
    director_id:id})
    DELETE oldedges
    OPTIONAL MATCH (l)<-[oldedges:moviesdirectors_directors]-(s:moviesdirectors{
    director_id:id})
    DELETE oldedges
    SET l.id = 11450 ,l.first_name = 'jack',l.last_name = 'adam';
    MATCH (a:directorsgenres) ,(b:directors)
    WHERE a.director_id = b.id
    CREATE (a)-[:directorsgenres_directors]->(b);
    MATCH (a:moviesdirectors) ,(b:directors)
    WHERE a.director_id = b.id
    CREATE (a)-[:moviesdirectors_directors]->(b);
```

Figure 29: Q_4 : Update SQL query and its equivalent Cypher query

```
SQL :
     DELETE FROM roles WHERE id = 8;
CYPHER :
     MATCH (r : roles) WHERE r.id = 8 DETACH DELETE r;
```

Figure 30: Q_5 : Delete SQL query and its equivalent Cypher query

Figure 31: Extensions of Query Q_1.

Query Q_1^1 :
```
MATCH (c:comments {score:"4"})-[s:comments_users]->(u:users)
RETURN *;
```

Query Q_1^2 :
```
MATCH (c:comments {score:"3"})-[s:comments_users]->(u:users)
RETURN *;
```

Query Q_1^3 :
```
MATCH (c:comments {score:"2"})-[s:comments_users]->(u:users)
RETURN *;
```

Query Q_1^4 :
```
MATCH (c:comments {score:"1"})-[s:comments_users]->(u:users)
RETURN *;
```

Query Q_1^5 :
```
MATCH (c:comments {score:"0"})-[s:comments_users]->(u:users)
RETURN *;
```

Figure 32: Extensions of Query Q_2.

Query Q_2^1 :
```
MATCH (c:comments {score:"7"})-[s:comments_users]->(u:users)
-[s1:users_badges]->(b:badges)
RETURN *;
```

Query Q_2^2 :
```
MATCH (c:comments {score:"6"})-[s:comments_users]->(u:users)
-[s1:users_badges]->(b:badges)
RETURN *;
```

Query Q_2^3 :
```
MATCH (c:comments {score:"5"})-[s:comments_users]->(u:users)
-[s1:users_badges]->(b:badges)
RETURN *;
```

Query Q_2^4 :
```
MATCH (c:comments {score:"4"})-[s:comments_users]->(u:users)
-[s1:users_badges]->(b:badges)
RETURN *;
```

Query Q_2^5 :
```
MATCH (c:comments {score:"3"})-[s:comments_users]->(u:users)
-[s1:users_badges]->(b:badges)
RETURN *;
```

Figure 33: Extensions of Query Q_3.

Query Q_3^1 :

```
MATCH (p:posts {score:"11"})-[s0:posts_comments]->(c:comments {score:"1"})
-[s1:comments_users]->(u:users)-[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_3^2 :

```
MATCH (p:posts {score:"8"})-[s0:posts_comments]->(c:comments {score:"1"})
-[s1:comments_users]->(u:users)-[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_3^3 :

```
MATCH (p:posts {score:"7"})-[s0:posts_comments]->(c:comments {score:"1"})
-[s1:comments_users]->(u:users)-[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_3^4 :

```
MATCH (p:posts {score:"4"})-[s0:posts_comments]->(c:comments {score:"1"})
-[s1:comments_users]->(u:users)-[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_3^5 :

```
MATCH (p:posts {score:"1"})-[s0:posts_comments]->(c:comments {score:"1"})
-[s1:comments_users]->(u:users)-[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Figure 34: Extensions of Query Q_4.

Query Q_4^1 :

```
MATCH (p:posts {score:"8"})-[s3:posts_posthistory]->(ph:posthistory),
      (p)-[s0:posts_comments]->(c:comments {score:"1"})
      -[s1:comments_users]->(u:users)
      -[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_4^2 :

```
MATCH (p:posts {score:"7"})-[s3:posts_posthistory]->(ph:posthistory),
      (p)-[s0:posts_comments]->(c:comments {score:"1"})
      -[s1:comments_users]->(u:users)
      -[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_4^3 :

```
MATCH (p:posts {score:"3"})-[s3:posts_posthistory]->(ph:posthistory),
      (p)-[s0:posts_comments]->(c:comments {score:"1"})
      -[s1:comments_users]->(u:users)
      -[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_4^4 :

```
MATCH (p:posts {score:"2"})-[s3:posts_posthistory]->(ph:posthistory),
      (p)-[s0:posts_comments]->(c:comments {score:"1"})
      -[s1:comments_users]->(u:users)
      -[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_4^5 :

```
MATCH (p:posts {score:"1"})-[s3:posts_posthistory]->(ph:posthistory),
      (p)-[s0:posts_comments]->(c:comments {score:"1"})
      -[s1:comments_users]->(u:users)
      -[s2:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Figure 35: Extensions of Query Q_5.

Query Q_5^1 :

```
MATCH (p:posts {score:"8"})-[s4:posts_posthistory]->(ph:posthistory),
      (p)-[s1:posts_comments]->(c:comments {score:"1"})
      -[s2:comments_users]->(u:users)
      -[s3:users_posts]->(p),
      (u)-[s0:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_5^2 :

```
MATCH (p:posts {score:"7"})-[s4:posts_posthistory]->(ph:posthistory),
      (p)-[s1:posts_comments]->(c:comments {score:"1"})
      -[s2:comments_users]->(u:users)
      -[s3:users_posts]->(p),
      (u)-[s0:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_5^3 :

```
MATCH (p:posts {score:"4"})-[s4:posts_posthistory]->(ph:posthistory),
      (p)-[s1:posts_comments]->(c:comments {score:"1"})
      -[s2:comments_users]->(u:users)
      -[s3:users_posts]->(p),
      (u)-[s0:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_5^4 :

```
MATCH (p:posts {score:"2"})-[s4:posts_posthistory]->(ph:posthistory),
      (p)-[s1:posts_comments]->(c:comments {score:"1"})
      -[s2:comments_users]->(u:users)
      -[s3:users_posts]->(p),
      (u)-[s0:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

Query Q_5^5 :

```
MATCH (p:posts {score:"3"})-[s4:posts_posthistory]->(ph:posthistory),
      (p)-[s1:posts_comments]->(c:comments {score:"1"})
      -[s2:comments_users]->(u:users)
      -[s3:users_posts]->(p),
      (u)-[s0:users_badges]->(b:badges {name:"Yearling"})
RETURN *;
```

1) From relational model to property graph: mapping data, schema, constraints and queries

Progress in Artificial Intelligence
https://doi.org/10.1007/s13748-025-00373-0

REGULAR PAPER

From relational model to property graph: mapping data, schema, constraints and queries

Abdelkrim Boudaoud[1] · **Houari Mahfoud**[1] · **Azeddine Chikh**[1]

Received: 10 July 2024 / Accepted: 13 April 2025

Abstract
The graph database systems (e.g. Neo4j, TigerGraph) are becoming widely used both by researchers and practitioners in order to deal with real-life problems such as fraud detection and social network marketing. Their success is due to their graph data model which demonstrates high efficiency when representing highly connected data and performing complex analytical tasks over it. As a huge amount of relational data still exists on the Web, many works have studied the mapping of this data into graphs. The proposed models are not suitable for real-life applications for many reasons (e.g. obfuscate the original schema; rely on theoretical query languages; do not investigate the mapping of constraints). Therefore, this article proposes a complete mapping process that translates any relational database into a graph database by mapping data, schema, constraints as well as read and update queries. Our process preserves all mapping properties: information preservation, as well as semantic, query, update and monotonicity preservation. The two first properties ensure that no information nor semantics of the original data is lost during the mapping. The query (resp. update) preservation is ensured via an algorithm that translates any SQL query (resp. update) into an equivalent Cypher query (resp. update). The monotonicity ensures that updates on the relational data do not require re-computing the equivalent data graph from scratch. Finally, we conducted an experimental study to validate our results. To our knowledge, this is the first work that studies all aspects of the mapping and provides a practical solution that can be easily integrated between two systems such as MySQL and Neo4j.

Keywords Data mapping · Property graph · Schema graph · SQL · Cypher · Neo4j

1 Introduction

For many decades, relational databases have been extensively studied by researchers and widely used by practitioners. The relational model gained its popularity [1] due to a straightforward design, a high data accuracy, a minimal redundancy, and the presence of a standard query language (SQL). The advent of Big Data has been marked by an explosion of Web data in terms of volume and interconnections. For instance, social networks (e.g. Facebook, Youtube) contain billions of entities [2] (e.g. Persons, Videos) which are interconnected with each other. Faced with this kind of data, the relational model has reached its limits in the sense that relational tables do not provide a clear view of the relationships that exist between the data. Furthermore, querying this data requires SQL queries that are often complex and time-consuming [3]. This has led researchers to adopt a new model for Web data. Indeed, the graph database model has received more attention during the last decade. Actually, graph database systems (e.g. Neo4j [4], JanusGraph [5], ArangoDB [6] and TigerGraph [7]), are widely used in many real-life applications, e.g. social network analysis, fraud detection, recommendation engines. Graph database model allows natural presentation of complex data where interconnections can be visualized more easily and clearly than in the relational model. In addition, several graph algorithms can be used to achieve efficient analytical tasks over graph database [8]. A real-life example has been discussed in [9]: investigative journalists have recently found, via the graph database model, surprising social relationships between executives of companies within the Offshore Leaks financial social network, linking com-

Abdelkrim Boudaoud
abdelkrim.boudaoud@univ-tlemcen.dz

Houari Mahfoud
houari.mahfoud@univ-tlemcen.dz

Azeddine Chikh
azeddine.chikh@univ-tlemcen.dz

[1] Abou-Bekr Belkaid University and LRIT Laboratory, Tlemcen, Algeria

Published online: 30 May 2025

1) Towards a Complete Direct Mapping From Relational Databases To Property Graphs

Towards a Complete Direct Mapping from Relational Databases to Property Graphs

Abdelkrim Boudaoud(✉), Houari Mahfoud, and Azeddine Chikh

Abou-Bekr Belkaid University & LRIT Laboratory, Tlemcen, Algeria
{abdelkrim.boudaoud,houari.mahfoud,azeddine.chikh}@univ-tlemcen.dz

Abstract. It is increasingly common to find complex data represented through the graph model. Contrary to relational models, graphs offer a high capacity for executing analytical tasks on complex data. Since a huge amount of data is still presented in terms of relational tables, it is necessary to understand how to translate this data into graphs. This paper proposes a *complete mapping* process that allows transforming any relational database (schema and instance) into a property graph database (schema and instance). Contrary to existing mappings, our solution preserves the three fundamental mapping properties, namely: *information preservation, semantic preservation* and *query preservation*. Moreover, we study mapping any *SQL* query into an equivalent *Cypher* query, which makes our solution practical. Existing solutions are either incomplete or based on non-practical query language. Thus, this work is the first complete and practical solution for mapping relations to graphs.

Keywords: Direct mapping · Complete mapping · Relational database · Graph database · SQL · Cypher

1 Introduction

Relational databases (RDs) have been widely used and studied by researchers and practitioners for decades due to their simplicity, low data redundancy, high data consistency, and uniform query language (SQL). Hence, the size of web data has grown exponentially during the last two decades. The interconnections between web data entities (e.g. interconnection between YouTube videos or people on Facebook) are measured by billions or even trillions [6] which pushes the relational model to quickly reach its limits as querying high interconnected web data requires complex SQL queries which are time-consuming. To overcome this limit, the graph database model is increasingly used on the Web due to its flexibility to present data in a normal form, its efficiency to query a huge amount of data and its analytic powerful. This suggests studying a mapping from RDs to graph databases (GDs) to benefit from the aforementioned advantages. This kind of mapping has not received more attention from researchers since only a few works [4,5,13,14] have considered it. A real-life example of this mapping has been discussed in [13]: "*investigative journalists have recently found, through graph analytics, surprising social*

P. Fournier-Viger et al. (Eds.): MEDI 2022, LNAI 13761, pp. 222–235, 2023.
https://doi.org/10.1007/978-3-031-21595-7_16

MEDI
2022
Nile University
جامعة النيل الأهلية
Certificate
of Recognition
3rd Place Best Paper Award
Presented to
Abdelkrim Boudaoud, Houari Mahfoud,
Azeddine Chikh
for the Paper Titled
Towards a Complete Direct Mapping
from Relational Databases to Property Graphs
in The 11th International Conference on Model & Data Engineering (MEDI2022)
Nile University, Egypt.
21-23 Nov. 2022
Prof. Ahmed Hassan
Nile University, Egypt
Philippe Fournier-Viger
College of Computer Science and Software
Engineering, Shenzhen University, China
Ladjel Bellatreche
ISAE-ENSMA Poitiers, France

2) A Framework for Querying Distributed Graph Data using Cypher

A Framework for Querying Distributed Graph Data using Cypher

Abdelkrim Boudaoud
LRIT Laboratory
Abou-Bekr Belkaid University
Tlemcen, Algeria
abdelkrim.boudaoud@univ-tlemcen.dz

Houari Mahfoud
LRIT Laboratory
Abou-Bekr Belkaid University
Tlemcen, Algeria
houari.mahfoud@univ-tlemcen.dz

2025 International Conference on Intelligent Computer Systems, Data Science and Applications (IC2SDA) | 979-8-3315-6892-4/25/$31.00 ©2025 IEEE | DOI: 10.1109/IC2SDA68097.2025.11331565

***Abstract*—During the last decade, graph databases have emerged as a flexible and expressive solution for handling highly interconnected and knowledge data. Graph pattern matching (*GPM*) is a core problem for these databases, aiming to find all subgraphs in a graph G that are isomorphic to a query graph Q. Commercial systems support *GPM* through an expressive graph query language, *Cypher*, which is an implementation of the *ISO-GQL* standard. However, for large graph data, *GPM* remains computationally intensive. Existing distributed approaches suffer from limitations: they use non-standard query languages; lack formal background, hindering comparison; or are designed for specific systems, hindering integration. Furthermore, distributed evaluation of graph queries has received less attention than for relational or RDF queries. In this paper, we provide a formalization for executing *GPM* in a distributed context using an *ISO-GQL* implementation like *Cypher*. The novelty is a new algorithm that fragments the query based on the boundary nodes of distant machines. After receiving partial results, a parallel algorithm merges and filters them to obtain complete results, controlled by a coordinator in a centralized data assembly architecture. This formalization can be easily implemented over any graph system and for any *ISO-GQL* implementation. We validate our framework through an implementation on *Neo4j AuraDB* and experiments on real-world datasets, demonstrating significant performance gains for complex queries in a distributed context compared to intractable single-node execution.**

***Index Terms*—Distributed Graph Database, Knowledge Graph, Graph Pattern Matching, Isomorphism, Neo4j, Cypher.**

I. Introduction

Graph pattern matching (*GPM*), which is quite crucial in graph databases, involves the intricate process of extracting insights (modeled as graph patterns) from big graph databases. This challenge is of particular interest in different areas such as data mapping [11], [12], access control [13], finance, cybersecurity, social networks, and chemistry. Formally, *GPM* requires the identification of all subgraph isomorphisms between the graph data and the graph query, and has been shown to be an NP-Complete problem [14]. That is, computing *GPM* within large-scale graph data is time-consuming and even infeasible on a single machine. To deal with this cost/infeasibility, some works have studied the extraction of only the most relevant results of *GPM* so its computation can be done efficiently in a single machine [15], [16]. On the other hand, computing *GPM* over distributed graph data (i.e. across different machines) has received a lot of attention during the last decade both from researchers and practitioners. Some studies [2], [17], [18] tackled distributed *GPM* with limited settings (very simple graph data and queries) and under the semantics of *graph simulation*. This semantics [19] aims to reduce the cost of *GPM* from NP-Complete to PTIME by relaxing the semantics of the isomorphism and returning approximate results. However, real-world scenarios often require a rich representation of graph data (e.g. with attributes and labels), complex graph queries (e.g. with conditions, subqueries), and exact answers. Moreover, all existing graph database systems (e.g. *Neo4j*, *Memgraph*) support *GPM* under isomorphism semantics only. Other studies [9], [10] propose practical solutions for distributed *GPM*, however, they lack formalization to integrate these solutions in different contexts or compare them with other approaches. A comparison made by the Neo4j group [1] shows that the *Cypher* query language [20] is very close to the new ISO standard of graph query language (denoted by ISO GQL [21]). Cypher is currently supported by many graph database systems (e.g. *Neo4j*, *MemGraph*, *RedisGraph*, *Amazon Neptune*).

The most challenge with Cypher relies on the semantics of its implementation, i.e. all graph systems use isomorphism semantics to execute Cypher queries, which is an NP-Complete problem, and consequently makes Cypher-based querying intractable. To reduce this time, some systems (e.g. *Neo4j*) allow the use of federated graph databases which is a fragmentation of the initial large graph data into independent subgraphs. The answering of Cypher queries in this case remains simple as it requires to execute the query on each fragment and to merge all results by a simple union. The task becomes more intriguing in the case of distributed data where the fragments (e.g. subgraphs) may contain common nodes and edges between them. That is, a query result may exist across one or many machines. Despite the wide use and high expressivity of Cypher, it is surprising that no work has studied the answering of Cypher queries over distributed graph data. Moreover, no graph database system allows such an answering. Furthermore, a formalization of this distributed answering is necessary to ensure the interoperability of the solution and to

[1] https://neo4j.com/blog/cypher-and-gql/cypher-path-gql (Accessed:2025-06-23)

1) R2G-ETL : A Tool for Mapping any Relational database to Graph database

R2G-ETL : A Tool for a Complete Mapping of Relational Databases to Graph Databases

Abdelkrim Boudaoud
LRIT Laboratory
Abou-Bekr Belkaid University
Tlemcen, Algeria
abdelkrim.boudaoud@univ-tlemcen.dz

Houari Mahfoud
LRIT Laboratory
Abou-Bekr Belkaid University
Tlemcen, Algeria
houari.mahfoud@univ-tlemcen.dz

***Abstract*—The graph data model has proven a high flexibility and efficiency when it comes to represent hyper connected data and make analytic tasks. Since most of the web data still resides in relational databases, efficient mapping methods are essential to enable graph-based analysis of relational data. This paper introduces *R2G-ETL*, a tool for mapping schemas, data, constraints, queries and updates from the relational model into their graph counterparts. The tool works seamlessly between any relational database system (e.g. PostgreSQL) and graph database system (e.g. Neo4j, MemGraph). Unlike prior tools, *R2G-ETL* guarantees critical mapping properties: *information preservation*, *semantic preservation*, *monotocity*, *query preservation* and *update preservation*. These properties prove that *R2G-ETL* preserves all aspects of the relational database being mapped to graph database. In addition, any SQL query/update is mapped into an equivalent Cypher query/update. This tool is the first one that allows a complete and a valid mapping from relations to graphs.**

***Index Terms*—Complete mapping, Relational database, Graph database, SQL, Cypher, Neo4j.**

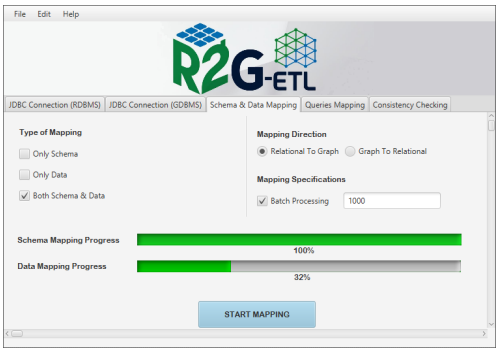


Fig. 1: The *R2G-ETL* GUI.

I. Introduction

For decades, relational databases have served as the foundation of data management, widely adopted in both academia and industry due to their intuitive design, strong data integrity, reduced redundancy, and standardized querying via SQL[1]. However, the emergence of Big Data characterized by massive volumes of highly interconnected data (e.g. social networks have billions of linked entities [1]) has revealed significant limitations of the relational database model. Primarily, complex relationships between entities are hardly represented and analyzed within a relational database, resulting in convoluted SQL queries and prohibitive computational costs [2]. Graph databases have emerged as a powerful alternative, gaining prominence over the past decade due to their ability to natively represent and query interconnected data. Graph database systems (e.g. Neo4j[2]) excel in applications such as social network analysis, fraud detection, and recommendation engines. They enable intuitive relationship visualization, efficient traversal, and advanced analytical algorithms capabilities that are cumbersome or inefficient in relational frameworks [3].

To leverage these advantages, the mapping of relational databases to graph databases has received increasing attention in recent years [4]–[8]. Existing mapping approaches face critical challenges: they consider only partial database elements (e.g., neglecting schemas and constraints), inadequately translate queries and do not consider updates, rely on impractical theoretical models, or distort the original schema. Moreover, no approach has studied all the mapping properties. For instance, our evaluation of the *Neo4j-ETL*[3] revealed structural inconsistencies with source databases and inefficient processing times.

To deal with these limits, we propose a complete and valid mapping solution, the *R2G-ETL* tool. The theoretical aspects of our tool have been studied and validated within a previous international conference paper [9] and a journal article being published. The *R2G-ETL* tool is composed by four modules that enable:

- Mapping relational schemas into graph schemas by supporting integrity constraints (*not null*, *unique*, and *check*);
- Mapping relational data into graph data;
- Mapping SQL queries (resp. updates) into Cypher queries (resp. updates);
- Checking the consistency of graph data w.r.t its schema graph.

[1]https://db-engines.com/en/
[2]https://neo4j.com/
[3]https://neo4j.com/developer/neo4j-etl/

References

[1] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[2] Information technology — semantic web — resource description framework (rdf), May 2024. URL https://www.w3.org/TR/rdf-concepts/. Accessed: 22/08/2025.

[3] Renzo Angles. The property graph database model. 2018.

[4] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc, Cambridge, 2015.

[5] B. Berger and K. Pack. Learning to recommend with social trust ensemble. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, 2007.

[6] D. Savage, X. Zhang, X. Yu, P. Chou, and Q. Wang. Anomaly detection in online social networks. *Social Networks*, 39:62–70, 2014.

[7] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, and J. Taylor. Industry-scale knowledge graphs: lessons and challenges. *Communications of the ACM*, 62(8):36–43, 2019.

[8] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2246–2258, 2022.

[9] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, page 7, 2016.

[10] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of SIGMOD*, pages 1433–1445, 2018.

[11] Information technology — database languages — gql, April 2024. URL https://www.iso.org/standard/76120.html. Accessed: 22/08/2025.

[12] OpenLink Software. OpenLink Virtuoso. https://virtuoso.openlinksw.com/. Accessed: 2025-08-25.

[13] The Apache Software Foundation. Apache Jena. https://jena.apache.org/. Accessed: 2025-08-25.

[14] Ontotext. GraphDB. https://www.ontotext.com/products/graphdb/. Accessed: 2025-08-25.

[15] Blazegraph. Blazegraph Database. https://blazegraph.com/. Accessed: 2025-08-25.

[16] Stardog Union. Stardog Knowledge Graph Platform. https://www.stardog.com/. Accessed: 2025-08-25.

[17] Neo4j, Inc. Neo4j Graph Database. https://neo4j.com/. Accessed: 2025-08-25.

[18] Memgraph. Memgraph. https://memgraph.com/. Accessed: 2025-08-25.

[19] TigerGraph. TigerGraph Graph Database. https://www.tigergraph.com/. Accessed: 2025-08-25.

[20] ArangoDB Inc. ArangoDB. https://www.arangodb.com/. Accessed: 2025-08-25.

[21] Amazon Web Services, Inc. Amazon Neptune. https://aws.amazon.com/neptune/. Accessed: 2025-08-25.

[22] Iso/iec 9075-16:2023 information technology – database languages sql – part 16: Property graph queries (sql/pgq). URL https://www.iso.org/standard/79473.html. Accessed: 22/08/2025.

[23] GSQL. https://www.tigergraph.com/gsql/. Accessed: 2025-08-25.

[24] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1421–1432, 2018.

[25] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23 (1):31–42, 1976.

[26] Stephen A Cook. The complexity of theorem-proving procedures. In *Logic, automata, and computational complexity: The works of Stephen A. Cook*, pages 143–152. 2023.

[27] Pavol Hell and Jaroslav Nesetril. *Graphs and homomorphisms*, volume 28. OUP Oxford, 2004.

[28] MR Garey and DS Johnson. Computers and intractability–a guide to np-completeness.(1979). *Google Scholar*, pages 155–158.

[29] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[30] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, volume 45, pages 43–53, 2006.

[31] Pasquale Foggia, Gennaro Percannella, and Mario Vento. Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(01):1450001, 2014.

[32] Mario Vento. A long trip in the charming world of graphs for pattern recognition. *Pattern Recognition*, 48(2):291–301, 2015.

[33] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23 (1):31–42, 1976.

[34] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento, et al. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.

[35] Vincenzo Carletti, Pasquale Foggia, and Mario Vento. Vf2 plus: An improved version of vf2 for biological graphs. In *Graph-Based Representations in Pattern Recognition: 10th IAPR-TC-15 International Workshop, GbRPR 2015, Beijing, China, May 13-15, 2015. Proceedings 10*, pages 168–177, 2015.

[36] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):804–818, 2017.

[37] Vincenzo Carletti, Pasquale Foggia, Antonio Greco, Mario Vento, and Vincenzo Vigilante. Vf3-light: A lightweight subgraph isomorphism algorithm and its experimental evaluation. *Pattern Recognition Letters*, 125:591–596, 2019.

[38] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.

[39] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14:1–13, 2013.

[40] Alpár Jüttner and Péter Madarasi. Vf2++—an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81, 2018.

[41] Huahai He and Ambuj K Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418, 2008.

[42] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: towards ultrafast and robust

subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, pages 337–348, 2013.

[43] Xuguang Ren and Junhu Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment*, 8(5):617–628, 2015.

[44] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214, 2016.

[45] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 international conference on management of data*, pages 411–426, 2018.

[46] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 international conference on management of data*, pages 1429–1446, 2019.

[47] Shixuan Sun and Qiong Luo. Subgraph matching with effective matching order and indexing. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):491–505, 2020.

[48] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of the 12th international conference on extending database technology: advances in database technology*, pages 192–203, 2009.

[49] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.

[50] Xuguang Ren and Junhu Wang. Multi-query optimization for subgraph isomorphism search. *Proceedings of the VLDB Endowment*, 10(3):121–132, 2016.

[51] Todd Plantenga. Inexact subgraph isomorphism in mapreduce. *Journal of Parallel and Distributed Computing*, 73(2):164–175, 2013.

[52] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. *Proceedings of the VLDB Endowment*, 6(4):265–276, 2013.

[53] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300, 2014.

[54] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*, pages 299–315, 2015.

[55] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[56] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, 2015.

[57] Miao Qiao, Hao Zhang, and Hong Cheng. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment*, 11(2):176–188, 2017.

[58] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, 2016.

[59] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment*, 12(10):1099–1112, 2019.

[60] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. Enumerating subgraph instances using map-reduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 62–73, 2013.

[61] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *Acm Sigmod Record*, 42(4):5–16, 2014.

[62] Todd L Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[63] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. *Proceedings of the VLDB Endowment*, 11:691–704, 2018.

[64] Per Fuchs, Peter Boncz, and Bogdan Ghit. Edgeframe: Worst-case optimal joins for graph-pattern matching in spark. In *Proceedings of the 3rd joint international workshop on graph data management experiences & systems (GRADES) and network data analytics (NDA)*, pages 1–11, 2020.

[65] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076*, 2019.

[66] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. Qfrag: Distributed graph search via subgraph isomorphism. In *proceedings of the 2017 symposium on cloud computing*, pages 214–228, 2017.

[67] Bibek Bhattarai, Hang Liu, and H Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1447–1462, 2019.

[68] Shixuan Sun and Qiong Luo. Parallelizing recursive backtracking based subgraph matching on

a single machine. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–9, 2018.

[69] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. Benu: Distributed subgraph enumeration with backtracking-based framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 136–147, 2019.

[70] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall Englewood Cliffs, 1989.

[71] Monika Rauch Henzinger, Thomas A Henzinger, and Peter W Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 453–462, 1995.

[72] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Graph pattern matching: From intractable to polynomial time. *Proceedings of the VLDB Endowment*, 3(1-2):264–275, 2010.

[73] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Capturing topology in graph pattern matching. *Proceedings of the VLDB Endowment*, 5(4), 2011.

[74] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 38(3):1–47, 2013.

[75] I. S. Abuhaiba. Offline signature verification using graph matching. *Turkish Journal of Electrical Engineering & Computer Sciences*, 15(1):89–104, 2007.

[76] Andreas Fischer, Ching Y Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. A fast matching algorithm for graph-based handwriting recognition. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 194–203, 2013.

[77] Josep Llados and Gemma Sanchez. Graph matching versus graph parsing in graphics recognition—a combined approach. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(03):455–473, 2004.

[78] Xu Xiaogang, Sun Zhengxing, Peng Binbin, Jin Xiangyu, and Liu Wenyin. An online composite graphics recognition approach based on matching of spatial relation graphs. *Document Analysis and Recognition*, 7:44–55, 2004.

[79] Sharat Chikkerur, Alexander N Cartwright, and Venu Govindaraju. K-plet and coupled bfs: a graph based fingerprint representation and matching algorithm. In *International Conference on Biometrics*, pages 309–315, 2006.

[80] Seyed Mehdi Lajevardi, Arathi Arakala, Stephen A Davis, and Kathy J Horadam. Retina verification system based on biometric graph matching. *IEEE transactions on image processing*, 22(9):3625–3635, 2013.

[81] Laurenz Wiskott, Jean-Marc Fellous, Nobert Krüger, and Christoph Von Der Malsburg. Face

recognition by elastic bunch graph matching. In *Intelligent biometric techniques in fingerprint and face recognition*, pages 355–396. 2022.

[82] Bindita Chaudhuri, Begüm Demir, Lorenzo Bruzzone, and Subhasis Chaudhuri. Region-based retrieval of remote sensing images using an unsupervised graph-theoretic approach. *IEEE Geoscience and Remote Sensing Letters*, 13(7):987–991, 2016.

[83] Sheheryar Khan, Mehmood Nawaz, Xu Guoxia, and Hong Yan. Image correspondence with cur decomposition-based graph completion and matching. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(9):3054–3067, 2019.

[84] Mang Ye, Jiawei Li, Andy J Ma, Liang Zheng, and Pong C Yuen. Dynamic graph co-matching for unsupervised video-based person re-identification. *IEEE Transactions on Image Processing*, 28(6):2976–2990, 2019.

[85] Wen fei Fan, Xin Wang, and Yinghui Wu. Expfinder: Finding experts by graph pattern matching. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1316–1319, 2013.

[86] Soufiana Mekouar, Nabila Zrira, and El Houssine Bouyakhf. Community outlier detection in social networks based on graph matching. *International Journal of Autonomous and Adaptive Communications Systems*, 11(3):209–231, 2018.

[87] Gorka Sadowski and Philip Rathle. Fraud detection: Discovering connections with graph databases. *White Paper-Neo Technology-Graphs are Everywhere*, 13:1–13, 2014.

[88] Adnan Anwar and Abdun Naser Mahmood. Anomaly detection in electric network database of smart grid: Graph matching approach. *Electric Power Systems Research*, 133:51–62, 2016.

[89] Georgiy Levchuk, John Colonna-Romano, and Mohammed Eslami. Application of graph-based semi-supervised learning for development of cyber cop and network intrusion detection. In *Disruptive Technologies in Sensors and Sensor Systems*, volume 10206, pages 67–82, 2017.

[90] Alain Menelet and Charles-Edmond Bichot. Characterization of android malware based on subgraph isomorphism. *arXiv preprint arXiv:2104.03566*, 2021.

[91] Neo4j, 2025. URL https://neo4j.com/. Accessed: 2025-06-25.

[92] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. gstore: answering sparql queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, 2011.

[93] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018.

[94] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[95] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying graphs*. Morgan & Claypool Publishers, 2018.

[96] Norbert Martinez-Bazan, Victor Muntes-Mulero, Sergio Gomez-Villamor, Jordi Nin, Miquel A Sanchez-Martinez, and Josep Lluis Larriba-Pey. Dex: High-performance exploration on large graphs for information retrieval. *Proceedings of the 16th ACM conference on Conference on information and knowledge management*, pages 573–582, 2007.

[97] Neo4j Vector Index and Search. https://neo4j.com/developer/genai-ecosystem/vector-search/. Accessed: 2025-08-25.

[98] Neo4j. Neo4j Fabric. https://neo4j.com/blog/cypher-and-gql/getting-started-with-neo4j-fabric/. Accessed: 2025-08-25.

[99] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Gabriel Kliot. Horton: Online query execution engine for large distributed graphs. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1289–1292, 2013.

[100] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. On directly mapping relational databases to RDF and OWL. pages 649–658, 2012.

[101] World Wide Web Consortium et al. R2rml: Rdb to rdf mapping language. 2012.

[102] Christian Bizer and Andy Seaborne. D2rq-treating non-rdf databases as virtual rdf graphs. In *Proceedings of the 3rd international semantic web conference (ISWC2004)*, volume 2004. Springer Hiroshima, 2004.

[103] Pham Thi Thu Thuy, Nguyen Duc Thuan, Yongkoo Han, Kisung Park, and Young-Koo Lee. Rdb2rdf: completed transformation from relational database into rdf ontology. In *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication*, pages 1–7, 2014.

[104] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. Mapping RDF databases to property graph databases. *IEEE Access*, pages 86091–86110, 2020.

[105] Kashif Rabbani, Matteo Lissandrini, Angela Bonifati, and Katja Hose. Transforming RDF graphs to property graphs using standardized schemas. *Proceedings of the ACM on Management of Data (SIGMOD)*, 2(6):242:1–242:25, 2024. doi: 10.1145/3698817.

[106] Hirokazu Chiba, Ryota Yamanaka, and Shuji Matsumoto. G2GML: graph to graph mapping language for bridging RDF and property graphs. In *International Semantic Web Conference (ISWC)*, pages 160–175. Springer, 2020.

[107] E. Haihong, Penghao Han, and Meina Song. Transforming RDF to property graph in hugegraph. In *Proceedings of the 6th International Conference on Engineering MIS*, 2020. doi: 10.1145/3410352.3410833.

[108] HugeGraph system. https://hugegraph.apache.org/. Accessed: 2025-08-25.

[109] Marco Brandizi, Ajit Singh, and Keywan Hassani-Pak. Getting the best of linked data and property graphs: rdf2neo and the knetminer use case. In *Semantic Web Applications and Tools for Life Sciences (SWAT4LS)*, 2018.

[110] Guanghui Zhang, Bo Yu, and Liping Bu. Bi-mapping between RDF and property graphs. In *International Conference on Technology Innovation and Data Engineering (ICTech)*, pages 34–39. IEEE, 2023.

[111] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. Converting relational to graph databases. page 1, 2013.

[112] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. R2g: A tool for migrating relations to graphs. pages 640–643, 2014.

[113] Radu Stoica, George Fletcher, and Juan F. Sequeda. On directly mapping relational databases to property graphs. pages 1–4, 2019.

[114] Radu-Alexandru Stoica. R2pg-dm: A direct mapping from relational databases to property graphs. Master's thesis, Eindhoven University of Technology, 2019.

[115] Ognjen Orel, Slaven Zakošek, and Mirta Baranović. Property oriented relational-to-graph database conversion. *Automatika*, pages 836–845, 2017.

[116] Neo4j ETL. https://neo4j.com/developer/neo4j-etl/, .

[117] Shunyang Li, Zhengyi Yang, Xianhang Zhang, Wenjie Zhang, and Xuemin Lin. *SQL2Cypher: Automated Data and Query Migration from RDBMS to GDBMS*. 2021.

[118] I Made Putrama and Péter Martinek. An automated graph construction approach from relational databases to neo4j. In *2022 IEEE 22nd International Symposium on Computational Intelligence and Informatics and 8th IEEE International Conference on Recent Achievements in Mechatronics, Automation, Computer Science and Robotics (CINTI-MACRo)*, pages 000131–000136, 2022.

[119] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. Distributed graph pattern matching. In *Proceedings WWW*, pages 949–958, 2012.

[120] Xiaole Wen, Shuai Zhang, and Haihang You. Drone: A distributed subgraph-centric framework for processing large scale power-law graphs. *arXiv preprint arXiv:1812.04380*, 2018.

[121] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. GRAPE: parallelizing sequential graph computations. *Proc. VLDB Endow.*, 10(12):1889–1892, 2017.

[122] Jingbo Xu, Zhanning Bai, Wenfei Fan, Longbin Lai, Xue Li, Zhao Li, Zhengping Qian, Lei Wang, Yanyan Wang, Wenyuan Yu, and Jingren Zhou. Graphscope: A one-stop large graph processing system. *Proc. VLDB Endow.*, 14(12):2703–2706, 2021.

[123] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. Processing SPARQL queries over distributed RDF graphs. *VLDB J.*, 25(2):243–268, 2016.

[124] Yanyan Song, Yuzhou Qin, Wenqi Hao, Pengkai Liu, Jianxin Li, Farhana Murtaza Choudhury, Xin Wang, and Qingpeng Zhang. Optimizing subgraph matching over distributed knowledge graphs using partial evaluation. *World Wide Web (WWW)*, 26(2):751–771, 2023.

[125] Peng Peng, Lei Zou, and Runyu Guan. Accelerating partial evaluation in distributed SPARQL query evaluation. In *Proceedings of ICDE*, pages 112–123. IEEE, 2019.

[126] Vasileios Trigonakis, Jean-Pierre Lozi, Tomás Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. adfs: An almost depth-first-search distributed graph-querying system. In *Proceedings of USENIX*, pages 209–224, 2021.

[127] Kongzhang Hao, Zhengyi Yang, Longbin Lai, Zhengmin Lai, Xin Jin, and Xuemin Lin. Patmat: A distributed pattern matching engine with cypher. In *Proceedings of CIKM*, pages 2921–2924, 2019.

[128] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. Nebula graph: An open source distributed graph database. *CoRR*, abs/2206.07278, 2022.

[129] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. GRAPE: parallelizing sequential graph computations. *Proc. VLDB Endow.*, 10(12):1889–1892, 2017.

[130] Database trend (2023). https://db-engines.com/en/.

[131] Wenfei Fan, Xin Wang, and Yinghui Wu. Answering pattern queries using views. *IEEE Trans. Knowl. Data Eng.*, pages 326–341, 2016.

[132] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. An empirical study on recent graph database systems. In *KSEM*, pages 328–340, 2020.

[133] Neo4j. https://neo4j.com/, .

[134] Janusgraph. https://janusgraph.org/.

[135] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Tigergraph: A native MPP graph database. *arXiv preprint arXiv:1901.08248*, 2019.

[136] Abdelkrim Boudaoud, Houari Mahfoud, and Azeddine Chikh. Towards a complete direct mapping from relational databases to property graphs. In *Model and Data Engineering*, pages 222–235, 2023.

[137] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.

[138] David Maier. The theory of relational databases. 1983.

[139] Jeffrey D. Ullman. Principles of database and knowledge-base systems, volume I. 14, 1988.

[140] Jan Paredaens, Paul De Bra, Marc Gyssens, and Dirk Van Gucht. The structure of the relational database model. 17, 1989.

[141] Slavica Aleksic, Milan Celikovic, Sebastian Link, Ivan Lukovic, and Pavle Mogin. Faceoff: Surrogate vs. natural keys. In *Advances in Databases and Information Systems - 14th East European Conference, ADBIS 2010, Novi Sad, Serbia, September 20-24, 2010. Proceedings*, pages 543–546, 2010.

[142] Sebastian Link, Ivan Luković, and Pavle Mogin. Performance evaluation of natural and surrogate key database architectures. 2010.

[143] Paolo Guagliardo and Leonid Libkin. A formal semantics of SQL queries, its validation, and applications. *Proc. VLDB Endow.*, pages 27–39, 2017.

[144] Gremlin. https://tinkerpop.apache.org/gremlin.html.

[145] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs*. 2018.

[146] Neo4j-ETL architecture diagram. https://neo4j.com/labs/etl-tool/1.5.0/.

[147] Abdelkrim Boudaoud, Houari Mahfoud, and Azeddine Chikh. Towards a complete direct mapping from relational databases to property graphs. In *MEDI*, pages 222–235, 2022.

[148] Abdelkrim Boudaoud, Houari Mahfoud, and Azeddine Chikh. From relational model to property graph: mapping data, schema, constraints and queries. *Progress in Artificial Intelligence*, pages 1–25, 2025.

[149] Adil Achraf Bereksi Reguig, Houari Mahfoud, and Abdessamad Imine. Towards an effective attribute-based access control model for neo4j. In *Proceedings of MEDI*, pages 352–366, 2023.

[150] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, pages 1367–1372, 2004.

[151] Houari Mahfoud. Expressive top-k matching for conditional graph patterns. *Neural Computing and Applications*, pages 1–17, 2021.

[152] Houari Mahfoud. Diversified top-k answering of cypher queries over large data graphs. In *Proceedings of AICCSA*, pages 1–8. IEEE, 2023.

[153] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. Distributed graph pattern matching. pages 949–958. ACM, 2012.

[154] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. Parallelizing sequential graph computations. *ACM Trans. Database Syst.*, 43(4):18:1–18:39, 2018.

[155] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of FOCS*, pages 453–462, 1995.

[156] Per-Olof Fjällström. *Algorithms for graph partitioning: A survey.* Linköping University Electronic Press, 1998.

[157] Tewodros Alemu Ayall, Huawen Liu, Changjun Zhou, Abegaz Mohammed Seid, Fantahun Bogale Gereme, Hayla Nahom Abishu, and Yasin Habtamu Yacob. Graph computing systems and partitioning techniques: A survey. *IEEE Access*, 10:118523–118550, 2022.

[158] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33 (8):103–111, 1990.

[159] Neo4j. Cypher versus the iso gql, 2025. URL https://neo4j.com/blog/cypher-and-gql/cypher-path-gql/. Accessed: 2025-06-23.

[160] Houari Mahfoud. Conditional graph pattern matching with a basic static analysis. In *Proceedings of MedPRAI*, pages 298–313, 2020.

[161] Wenfei Fan, Xin Wang, and Yinghui Wu. Diversified top-k graph pattern matching. *Proceedings of the VLDB Endowment*, pages 1510–1521, 2013.

[162] Xintong Guo, Hong Gao, Yinan An, and Zhaonian Zou. Diversified top-k querying in knowledge graphs. In *Proceedings of APWeb-WAIM*, pages 319–336, 2020.

[163] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. Minigraph: Querying big graphs with a single machine. *Proc. VLDB Endow.*, 16(9):2172–2185, 2023.

Abstract

During the past decade, we have witnessed a wide speed use of graph databases for modeling and analyzing highly interconnected data, the relational model quickly reaches its limit when it comes to query and analyze such data. Despite that, the relational model still dominates in the Web. The first motivation of this thesis is to bridge this gap by presenting a framework for mapping and efficiently querying relational data within a graph paradigm, with contributions spanning theoretical foundations, scalability solutions, and practical implementations. First, we establish a rigorous formal foundation by introducing a complete mapping process that maps any relational database, along with its schema, data, constraints, and operations, to an equivalent graph database. This process is the first to holistically guarantee critical mapping properties: *information* and *semantic preservation* to prevent loss of meaning; *query* and *update preservation* through algorithms that automatically convert SQL (relational query language) to Cypher (graph query language); and *monotonicity* for efficient incremental updates. Next, we synthesize these foundations into **R2G-ETL**, an end-to-end software tool that automates the complete mapping process from relational systems to graph systems. As the first tool to enforce all formal preservation properties, it provides a validated and practical pipeline for real-world adoption. The second contribution of this thesis deals with the computational complexity of graph pattern matching on large-scale data. As querying of graph data is based on the subgraph isomorphism, an NP-Complete problem, querying a large graph data may be time-consuming even with the more efficient systems like Neo4j. One of the possible solutions is to distribute the large graph data into separated machines, execute the query in each machine, collect, and merge the results. To our

knowledge, no work has proposed a complete formalization of this problem. We then formalize a distributed querying framework for Cypher, the most widely used graph query language. This framework is centered on three algorithms, the first one handles user query fragmentation for distributed execution; the second one focuses on assembling the partial results returned from each fragment; while The core algorithm coordinates all these operations between the coordinator and the fragments, managing them in a parallel, multi-threaded manner. The distributed framework can integrate seamlessly within any graph database system such as Neo4j, overcoming the limitations of previous approaches.

keywords: Data mapping, Property graph, Graph Pattern Matching, Distributed Graph Pattern Matching, Cypher, Neo4j.

Résumé

Au cours de la dernière décennie, les bases de données orientées graphes se sont largement imposées pour modéliser et analyser des données fortement interconnectées, là où le modèle relationnel atteint rapidement ses limites. Toutefois, ce dernier demeure dominant sur le Web. Cette thèse vise à combler ce fossé en proposant un cadre permettant de transformer et d'interroger efficacement des données relationnelles selon un paradigme graphe, en apportant des contributions à la fois théoriques, techniques et pratiques. Nous introduisons d'abord un processus formel de transformation garantissant la préservation de l'information, de la sémantique, des requêtes et des mises à jour, ainsi que la monotonie des opérations. Ce processus est concrétisé dans R2G-ETL, un outil logiciel assurant une migration automatisée et fiable de bases relationnelles vers des systèmes graphe, en particulier via la conversion de SQL vers Cypher. La seconde contribution porte sur l'éfficacité du requêtage de graphes à grande échelle, reposant sur l'isomorphisme de sous-graphes, un problème NP-complet. Pour y faire face, nous proposons un cadre de requêtage distribué pour Cypher, structuré autour de trois algorithmes : (i) la fragmentation de la requête pour exécution distribuée, (ii) l'assemblage des résultats partiels, et (iii) un algorithme central orchestrant ces opérations entre un coordinateur et les fragments de manière parallèle et multi-threadée. Cette approche améliore considérablement l'efficacité du requêtage dans des systèmes comme Neo4j, tout en posant des fondations théoriques solides.

mots-clés: Mapping de données, graph de propriétés, interrogation des graphes de données, graphes de données distribués, Cypher, Neo4j

مُلخّص

شهد العقد الأخير انتشارًا واسعًا لقواعد البيانات البيانية لاستخدامها في نمذجة وتحليل البيانات ذات الترابط العالي، حيث يصل النموذج العلاقي إلى حدوده بسرعة عند التعامل مع هذا النوع من البيانات. ومع ذلك، لا يزال النموذج العلاقي هو النموذج السائد على الويب. تهدف هذه الأطروحة إلى سد هذه الفجوة من خلال تقديم إطار يسمح بتحويل واستعلام البيانات العلاقية بكفاءة ضمن نموذج بياني، مع تقديم مساهمات تغطي الأسس النظرية، والحلول التقنية للتوسّع، والتطبيقات العملية.

نقترح أولًا عملية تحويل رسمية تضمن حفظ المعلومات، والمعنى الدلالي، واستمرارية الاستعلامات والتحديثات، بالإضافة إلى خاصية الرتابة التي تتيح دعم التحديثات التزايدية بكفاءة. وقد تم تجسيد هذا النهج ضمن أداة برمجية شاملة تدعى $R2G-ETL$ ، تقوم بأتمتة عملية التحويل من قواعد البيانات العلاقية إلى البيانية، بما في ذلك التحويل من لغة SQL إلى $Cypher$.

تتمثل المساهمة الثانية في معالجة تعقيد استعلام الأنماط البيانية ضمن بيانات ضخمة، وهو ما يرتبط بمشكلة تماثل تحت الرسوم البيانية، وهي مسألة تنتمي لفئة $NP-Complete$. ولمعالجة ذلك، نقترح إطارًا موزّعًا لاستعلامات $Cypher$ يعتمد على ثلاثة خوارزميات أساسية: (١) تجزئة الاستعلام لتنفيذه بشكل موزّع، (٢) تجميع النتائج الجزئية، و(٣) خوارزمية تنسيق مركزية تنظم هذه العمليات بين المنسّق والأجزاء المختلفة بطريقة متوازية ومتعددة الخيوط $(multi-threaded)$. يقدم هذا الإطار حلًا فعالًا وقابلًا للتكيّف مع أنظمة مثل $Neo4j$ ، متجاوزًا قيود الأساليب السابقة.

الكلمات المفتاحيّة: تعيين البيانات ، الرسم البياني ذو الخصائصي