DESIGN OF FORMAL QUERY LANGUAGES AND SCHEMAS FOR GRAPH DATABASES

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Supervisor
Dr. Roopak Sinha
Dr. Alan T. Litchfield
Dr. Kenneth Johnson

28th April 2021

By
Chandan Sharma
School of Engineering, Computer and Mathematical Sciences

Abstract

The industry-wide adoption of graph databases has been hindered due to the lack of a standard query language. Hence projects such as ISO/IEC 39075 have been proposed to integrate features from existing graph query languages, including Cypher, PGQL and G-Core. Integrating existing query languages requires a systematic comparison so that exclusive characteristics can be identified. Comparisons can be conducted by using graph query language benchmarks which are built on theoretical language formalisms. Literature suggests that existing theoretical language formalisms for graph databases are not expressive enough in formulating different data retrieval queries. Existing benchmarks also utilise the topological information stored in the graph schema to formulate queries for comparing graph query languages. Contemporary graph databases including Neo4j, Oracle and, Apache Tinkerpop, are either schema-less or schema optional to support frequent changes in the structure of data found in domains requiring high flexibility. However, the absence of robust graph schema impacts data consistency, integrity, and analytics in graph databases.

Lack of expressive theoretical language formalisms and robustly-defined graph schemas are the two open problems in current graph database research. This thesis contributes towards solving these problems. We propose novel formalisms of conjunctive and union of conjunctive queries extended with Tarski's algebra that are more expressive than existing theoretical language formalisms. The formalisms are then used to formulate benchmark queries for comparing the expressiveness of Cypher and PGQL on the two core features of graph pattern matching and graph navigation, revealing the standard and exclusive characteristics for these languages. We present a formal algebra FLASc that assists in formulating robust graph schemas. We consider three case studies related to domains such as cyber-physical systems, big data analytics and tourism. These case studies illustrate the use of FLASc for transforming and loading data-sets for heterogeneous sources into graph databases such as Neo4j, thereby ensuring data consistency and integrity.

Findings from this research suggest that formally defined graph schemas help generate efficient benchmark queries, facilitating the comparison of existing graph query languages. Furthermore, graph schemas are vital for ensuring better data manageability and developing future graph query languages to support data definition and data retrieval mechanisms from graph databases. Overall, our study serves as a formal basis for generating robust graph schemas and developing benchmarks for comparing existing graph query languages. This study assists in moving towards query language integration and interoperability between available graph database technologies; therefore, it serves as a basis for upcoming standards such as ISO/IEC 39075.

Contents

List of Tables

List of Figures

Attestation of Authorship

Publications

The publications listed below are a result of the research conducted in fulfilment of the degree of Doctor of Philosophy.

Table 1: List of published research work

Manuscript	Publication	Contribution
1	Sharma, C. & Sinha, R. (2019) A Schema-First Formalism for Labeled Property Graph Databases: Enabling Structured Data Loading and Analytics: Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT). DOI: `https://doi.org/10.1145/3365109.3368782` *(Chapter 3 and Chapter 4)*	Chandan Sharma: 85%, Roopak Sinha: 15%
2	Sharma, C., Sinha, R., & Johnson, K. (2021). Practical and comprehensive formalisms for modeling contemporary graph query languages. Information Systems, Elsevier, 101816. DOI: `https://doi.org/10.1016/j.is.2021.101816`, Volume 102, December 2021 *(Chapter 5 and Chapter 6)*	Chandan Sharma: 85%, Roopak Sinha: 12%, Kenneth Johnson: 3%
3	Sharma, C. & Sinha, R. (2021) FLASc: A Formal Algebra for Labeled Property Graph Schema: Automated Software Engineering, Springer, Manuscript Number: AUSE-D-21-00038 *(Chapter 7 and Chapter 8)* ***Under review***	Chandan Sharma: 90%, Roopak Sinha: 10%

Table 1 continued from previous page

Manuscript	Publication	Contribution
4	Sharma, C., Sinha, R. & Leitao, P. (2019) IASelect: Finding Best-fit Agent Practices in Industrial CPS Using Graph Databases: IEEE 17th International Conference on Industrial Informatics (INDIN). DOI: `https://doi.org/10.1109/INDIN41052.2019.8972272` *(Appendix A and Appendix B)*	Chandan Sharma: 82%, Roopak Sinha: 12%, Paulo Leitao: 6%
5	Sharma, C. (2020) FLUX: From SQL to GQL query translation tool: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). URL: `https://ieeexplore.ieee.org/abstract/document/9286038` *(Appendix C and Appendix D)*	Chandan Sharma: 100%

We, the undersigned, hereby agree to the percentages of participation to the chapters identified above.

Table 2: Signatures of the contributors

Chandan Sharma
Dr. Roopak Sinha
Dr. Kenneth Johnson
Dr. Paulo Leitao

Acknowledgements

Throughout this research, I have received a great deal of support and assistance. I would first like to express my deep and sincere gratitude to my primary research supervisor, Dr Roopak Sinha, for providing me with invaluable guidance. Roopak, your insightful feedback, dynamism, vision and sincerity has deeply inspired me. I also thank you for your friendship and empathy.
I thank my secondary supervisors, Dr Kenneth Johnson and Dr Alan T. Litchfield, for their valuable guidance and support throughout my research journey. Kenneth and Alan, the feedback and suggestions that you gave were critical in my research journey. Thank you.
I would also like to acknowledge my colleagues at Software Engineering Research Lab (SERL) for their support. I really enjoyed our discussions and happy distractions that helped me relax outside of my research.
I am incredibly grateful to my parents (Dr. Pritam Chand Sharma and Mrs. Neelam Sharma) for their love, care, prayers, and sacrifices to educate and prepare me for my future. Mom and Dad, thank you for motivating me and believing in me.
Last but not least, I would like to thank my wife Sudha for being understanding and patient with me. Thank you.

Chapter 1

Introduction

Big data systems need scalable data management solutions. Big data refers to data-sets that are massively large in size and data complexity [3]. For instance, data-sets used in discovering consumer shopping habits, to enable predictive inventory ordering, personalized health plans and personalized marketing are some examples of big data. The size of a data-set refers to the number of records stored, and data complexity refers to the interactions or connections between stored data. Conventional data processing applications are becoming incompetent in handling large data-sets in the current age of big data [4, 5]. The associated storing and querying mechanisms must be altered to manage and curate big data efficiently [6]. Graph databases suit big data as they provide a better alternative for handling highly interconnected data-sets [7, 8].

From a theoretical point of view, graph databases were first introduced as *logical data model* [9] by Kuper in 1985. In the late 1980s application of graph databases was identified in hypertext systems [10, 11]. Furthermore, similar applications were also found in object-oriented databases [12] and semi-structured data [13] in 1990s. In the last two decades, there has been increasing interest from the industry in graph databases. With the dawn of Web 3.0 and application domains such as social media, industrial internet of things (IIoT) and artificial intelligence, more businesses are operating online and generating data-sets that are highly interconnected [14, 15, 16].

Three main factors are contributing towards the popularity of graph databases. The first factor is the semantic web and the Resource Description Framework (RDF) that are used to models online resources as labeled and directed multi graphs [17, 18, 19, 20, 21]. RDF is a W3C[1] data model standard that describes data as subject–predicate–object expressions (or triples). This allows the creation of graphs of knowledge, however unlike more general purpose graph databases, there is no support for properties or labels - everything is represented using triples. New application domains such as social media increased the demand for a less restrictive data model [22] than RDFs. This gave rise to the labeled property graph data model [23, 24, 25, 1, 26, 27] that allow modeling real-world domains including social media [20, 28], bio-informatics [29], astronomy [30, 31], and chemistry [6, 5] as labeled, directed and, attributed multi graphs [25, 32]. Finally, the third factor is the shift of interest in analytic from report generation to discovering complex relationships between data [33, 34, 35, 36, 37].

Labeled property graph data model is more expressive than other graph data models, such as the resource description framework (RDF) [38], as it enables the storage of information inside nodes and edges as attributes (or properties) that exist in the form of key-value pairs [32]. This means that in graph databases following the labeled property graph data model, information can be embedded inside relationships which is an advantage over the RDF data model [15].

1.1 Graph databases

Labeled property graph databases, henceforth referred to as graph databases, are storage systems that use labeled property graph data model as the underlying data structure. A graph database representing social media network is presented in Figure 1.1. The

[1]`https://www.w3.org/TR/rdf-concepts/`

graph database consists of five nodes and eight edges. Each node is labeled as `PERSON`, while edges are labeled as `LIKED, MOTHER_OF` and `FATHER_OF`. For instance, node n_4 is labeled as `PERSON` and has an associated property signifying that name of the `PERSON` is Paul.

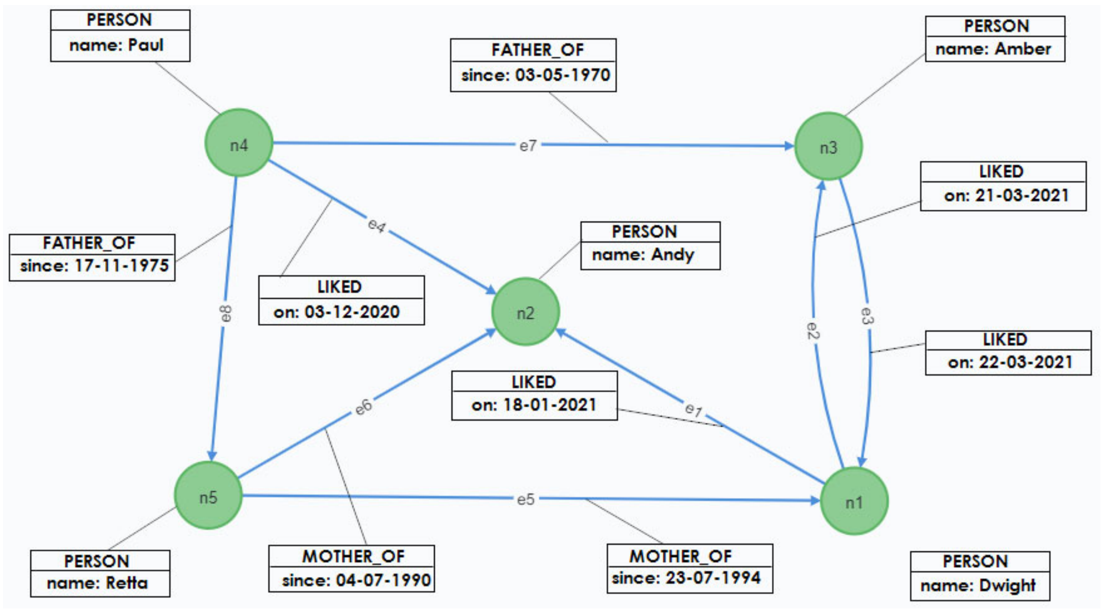


Figure 1.1: A social media network represented as a graph database

Edge labels and direction of edges assist in determining the nature of relationships in a graph database [39]. Properties associated with edges further elaborate the nature of the relationship between any two nodes. As shown in Figure 1.1, a `PERSON` Paul is connected to another `PERSON` Andy where the edge is labeled as `LIKED`. This relationship represents that Paul liked a post by Andy. Properties associated with the edge further elaborate the relationship by providing context that Paul liked Andy's post on 3^{rd} December 2020. Similarly, the edge labeled as `FATHER_OF` between Paul, Retta and Amber signifies that Paul has two daughters and properties associated with these edges represent the date of birth of Retta and Amber.

There are two fundamental capabilities provided by any graph database: the ability to retrieve the stored data and mechanisms to ensure data consistency and integrity.

1.1.1 Data retrieval in graph databases

Graph databases utilize query languages for retrieving the stored data [25, 40]. Graph query languages are declarative, and they have been motivated from query languages for relational databases proposed by Edgar F. Codd in the early 1970s [41, 32]. The significant advantage of using a declarative query language is that computational logic can be expressed without describing the control flow. A user only needs to describe "what" data to retrieve from the graph database without defining "how" the data must be extracted from the graph database [1, 42]. For example, in the graph database presented in Figure 1.1 one can ask information about Paul's daughters. To describe such a query in a graph query language, a user only needs information about entities in a graph database and relationships between the entities.

1.1.2 Mechanisms to ensure data consistency and integrity

Graph schemas are efficient in capturing the information about entities and relationships of a graph database [23]. Furthermore, graph schemas enable the enforcement of integrity constraints that ensure data consistency and integrity [43, 44, 45, 46]. In graph databases relationships are as important as data itself [23, 47, 48, 49, 50]. Creation of any incorrect relationship may result in corrupting the entire graph database. For instance, altering the direction of an edge between Paul and Retta of the graph database shown in Figure 1.1 can result in data corruption. Graph schemas are essential for systematic data retrieval and ensuring data consistency and integrity in graph databases.

1.1.3 Lack of robust schemas and standard query language for graph databases

Contemporary graph databases such as Neo4j [51], Oracle [52] and, Apache Tinkerpop [53] opt to be either schema-less or schema optional. Lack of graph schema has impacted the industry-wide adoption of graph databases. Absence of graph schema is disadvantageous in dynamic domains such as social media that are bound to frequent updates; hence, chances of data corruption are higher.

Another major factor that has affected the adoption of graph databases is a lack of a standard graph query language. Hence, projects such as ISO/IEC 39075 have been proposed that aim at developing a standardised graph query language by comparing and integrating existing query languages. Comparisons can be conducted using graph query language benchmarks built on existing theoretical language formalisms. However, as suggested by authors in [33, 54, 55] existing theoretical formalisms for graph query languages are not expressive enough.

1.2 Research objectives and contributions

Given the limitations in existing graph database technologies, the work presented in this thesis primarily focuses on achieving the following two research objectives.

RO1: Extend the existing theoretical language formalisms to propose novel formalisms that can be used to build benchmark queries for comparing the expressiveness of graph query languages.

RO2: Enhance the existing graph data modeling approaches and propose novel methods for constructing graph schemas so that data consistency and integrity can be ensured in labeled property graph databases.

In the following sections, we briefly discuss the background and limitations of existing approaches for querying and modeling graph databases. We also present the major contributions of this work for achieving the research objectives.

1.2.1 Formalisms for graph query languages

Graph query languages use mechanisms to support querying a graph database [56]. These mechanisms are theoretical language formalisms, including conjunctive queries (CQ) [57, 58], union of conjunctive queries (UCQ) [59, 60, 61], conjunctive regular path queries (CRPQ) [62, 63], and union of conjunctive regular path queries (UCRPQ) [62, 64, 65, 34] that are shared across several graph query languages [66]. Primarily these formalisms can be used to express two types of data retrieval queries: *graph pattern matching* and *graph navigation*.

Graph pattern matching queries

Graph pattern matching queries are the starting point of every graph-based data retrieval [40, 25, 32]. In such queries, the main goal is to find similar sub-graph occurrences over a graph database by using a graph pattern matching algorithm [28]. Graph pattern matching queries can only search graph databases in a bounded manner [32, 25]. Graph pattern matching queries are based on the formalism of conjunctive queries and union of conjunctive queries [32].

In graph databases, nodes having important connections might not be directly connected [23, 67, 68]. For instance, to find information about grand children of Paul in the graph database shown in Figure 1.1, we have to first have to search for daughters of Paul and then search for their children. To search nodes that are not connected directly to each other, one needs to navigate multiple intermediate nodes and traverse several edges connecting the nodes also called as graph navigation.

Graph navigation queries

Graph navigation queries are extensions of graph pattern matching queries and provide mechanisms to navigate through the graph database. This an important feature where the main goal is to search for the existence of paths between any two given nodes of a graph database [69, 25, 32, 70, 71] where a path is a (possibly infinite) sequence of nodes and edges. For instance, in the graph database presented in Figure 1.1 there exist a path connecting nodes labeled as Paul and Dwight. In the path, node Paul is connected to node Retta by an edge labeled as `FATHER_OF` and node Retta is connected to node Dwight by an edge labeled as `MOTHER_OF`.

Graph navigation queries use formalisms such as regular path queries (RPQ) [72], two-way regular path queries (2RPQ) [73], nested regular expressions (NRE) [30] and Tarski's *relation* algebra (TA) [74, 75] to navigate through a graph database. These formalisms share some basic operators such as concatenation, union and, Kleene star [74] that can be used to express and then search for paths in graph navigation queries.

Limitations of graph query language formalisms

Existing formalisms for graph query languages are not expressive enough [62, 33, 76, 30, 72, 71, 65, 77]. Expressiveness refers to the ability of a language to express data retrieval queries. Formalisms on conjunctive regular path queries (CRPQ) combine the expressive power of conjunctive queries and regular path queries. CRPQ based formalisms are used in existing benchmarks such as gMark [78, 79, 77]. However, in CRPQs conjunction is not closed under Kleene star; therefore, paths containing branches cannot be searched in an unbounded manner. Nested regular expressions (NRE) overcome the limitations of CRPQ by providing a branching operator that can be used with the Kleene star operator. However, NREs and CRPQs are incomparable in expressiveness [30]. Tarski's algebra [35, 55, 74] is more expressive than NREs, RPQs and 2RPQs however, they can only be used for expressing graph navigation queries.

Our contribution

This contribution relates to our first research objective RO1. We propose the extension of conjunctive queries and union of conjunctive queries with Tarski's algebra (CQT/UCQT). These formalisms are more expressive than existing formalisms and help compare existing practical graph query languages by enabling the formulation of benchmark queries. In order to demonstrate the utility of CQT/UCQT we present an integrated framework that facilitates the formulation of benchmark queries that are then used to evaluate the expressiveness of two practical graph query languages Cypher and PGQL on the core features of graph pattern matching and graph navigation.

1.2.2 Data modeling approaches for graph databases

Traditional data modeling consists of three stages: *conceptual, logical* and *physical* modeling [80]. In the conceptual modeling stage, requirements related to a problem domain are gathered, and an abstract model of the graph database is created. The abstract model, also called the conceptual graph schema, captures the real-world entities of the problem domain and relationships between them. In the logical modeling stage, certain rules related to the problem domain are defined and enforced over the conceptual graph schema. This process results in a logical graph schema. These rules are also known as integrity constraints and are defined to ensure data consistency in the graph database. Finally, the physical modeling stage represents the realization of conceptual and logical graph schema for transforming and loading data-sets into a graph database [44, 45, 43].

Limitations of modeling graph databases

Existing research on graph database modeling focuses on logical and physical data modeling stages. Subsequent studies [81, 43, 44, 45, 46, 82, 83, 84] focus on the integration of logical and physical modeling stages for graph databases. Conceptual modeling is considered to be trivial and is done in an ad-hoc manner. Graph database vendors such as Neo4j propose the use of visual tools such as Arrow[2] to create conceptual graph schemas. However, the robustness of the conceptual graph schema designed by such tools cannot be assured. Furthermore, the integration of conceptual graph schemas generated by existing tools at the logical and physical modeling stage requires human intervention.

Our contribution

This contribution relates to our second research objective RO2. We present FLASc a formal algebra for formulating conceptual and logical graph schemas for graph databases. Operations defined in FLASc are based on the conceptual graphs proposed by Sowa [47, 48, 49, 50]. We illustrate the use of FLASc to enforce integrity constraints. To show the integration of conceptual, logical and physical modeling stages, we integrate FLASc with the well-known Extract-Transform-Load design pattern. We consider three case studies related to domains such as cyber-physical systems (P2660.1 data-set) [85], big data analytics (`BiDaML` diagram data-set) [86], and tourism (Airbnb data-set) [87] for demonstrating the utility of FLASc for systematically transforming and loading data-sets from heterogeneous sources into graph databases such as Neo4j.

[2]`http://guides.neo4j.com/arrows`

1.3 Organisation

This thesis is structured as follows. In Chapter 2 we present the background and literature review conducted for this research. Conference publication presented in Chapters 3 and 4 is foundational for preparing Chapters 5, 6, 7 and 8. Gaps identified in Chapter 2 confer to the two main contributions of this thesis. The first contribution, related to the novel formalisms of CQT and UCQT, is introduced in Chapter 5 and then presented in Chapter 6. The second contribution related to the formal algebra FLASc is introduced in Chapter 7 and then discussed in Chapter 8. Overall findings, insights and conclusions of this thesis are presented in Chapter 9. Additionally, there are two conference publications presented in this thesis. In Appendix A and B we present a tool `IASelect` for querying graph databases. Appendix B also presents a case study related to cyber-physical systems which has been used in validating the formal algebra presented in Chapter 8. A conference publication related to the future direction of this work is presented as Appendix C and D. A list of acronyms used through out this thesis is presented as Appendix E.

Chapter 2

Literature Review

2.1 Introduction

To identify the limitations and challenges in querying and modeling approaches for graph databases, we first conducted a literature review. In the review process, we looked at the existing research that has been proposed by both industry and academia. Following this approach enabled us to analyze the advances and limitations related to modeling and querying graph databases. The literature review was primarily focused on the following four research questions.

RQ1 What are the existing graph data models and associated query languages that have been proposed by industry and academia?

RQ2 What are the desired core features in graph query languages?

RQ3 What are the theoretical language formalisms used for querying graph databases?

RQ4 What are the existing approaches used for modeling graph databases?

For addressing RQ1, we investigated current literature related to graph data models and query languages. A brief survey related to the history of graph data models and query languages identified for answering RQ1 is presented in Section 2.2. The survey enabled us to identify several aspects of existing graph data models and query languages. Graph query languages share some common and exclusive characteristics, such as core features and theoretical language formalisms. These findings lead us to investigate RQ2 and RQ3 that are answered in Sections 2.3 and 2.4 respectively. Finally, for addressing RQ4, we looked at modeling approaches that have been used to construct graph data models identified in RQ1 and findings for RQ4 are presented in Section 2.5.

2.1.1 Methodology for conducting the literature review

For conducting the literature review, we first formulated the search strings by identifying keywords from research questions RQ1-RQ4. Then the search strings were used to identify research papers on digital libraries such as Scopus, IEEE Explore, ACM Digital library and Science Direct. The AUT library portal was used to access the digital libraries. The inclusion and exclusion criteria of a research paper were based on the publication's venue reputation. This was verified by using web portals such as scimagojr [1] and core [2]. Furthermore, snowballing, expert suggestions and, the relevant paper published in highly reputed database venues such as ACM TODS [3], IEEE TKDE [4], ACM SIGMOD Record [5],VLDB Journal [6], EDBT [7], ICDT [8] and PODS [9] were other inclusion criteria for selecting a research paper.

[1] `https://www.scimagojr.com/`
[2] `http://portal.core.edu.au/conf-ranks/`
[3] `https://dl.acm.org/journal/tods`
[4] `https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=69`
[5] `https://dl.acm.org/newsletter/sigmod`
[6] `https://www.springer.com/journal/778`
[7] `https://dblp.uni-trier.de/db/conf/edbt/index.html`
[8] `https://dblp.uni-trier.de/db/conf/icdt/icdt2019.html`
[9] `https://dblp.uni-trier.de/db/conf/pods/index.html`

2.2 A brief history of graph data models and associated query languages

We present a brief survey related to the history of graph data models and query languages proposed by industry and academia in the last thirty years.

2.2.1 Graph data models and query languages proposed by academia

Graph data models and query languages in late 1980s

The first graph query language G was proposed in [88] as a complement to relational query languages for expressing transitive and recursive queries over graph-structured data. A query in G is represented as a labeled directed multi-graph consisting of nodes and edges, where nodes represent data and edges represent regular expressions over edge labels. The language G+ was proposed as an extension to G language [89]. The prime use case of G+ was the ability to compute simple paths in graph-structured data, and the alphabets of the path must satisfy a regular expression defined over the edge labels of the graph database. However, it is noted in [89] that the problem of finding simple paths in a graph is intractable, which means that the problem can be solved theoretically, but in reality, solving the same problem will require too many computational resources. The language Graphlog was proposed as an extension to G and G+ by adding negation and union operators [11]. Like G and G+, Graphlog also used regular expressions to express and then search for paths in a graph database. All three languages are used to retrieve data from node and edge labeled graph data model.

Graph data models and query languages in early to mid 1990s

A query language with SQL like syntax and uses regular expressions over the node and edge labels to define paths over a graph-based data model called Gram was proposed in [90]. Pattern Matching Language (PaMaL) was a graphical query language proposed in [91] for object-oriented databases. The main feature of PaMaL was pattern matching over graphs and used programming constructs such as loops, procedures and programs. GOAL query language was also proposed for querying object-oriented databases [92]. GOAL supported the use of recursion and used pattern matching for sub-graphs over graph databases. Both PaMaL and GOAL are based on graph-oriented object database model (GOOD) [93] however, PaMaL and GOAL do not support graph navigation queries. A language for querying GraphDB data model was proposed in [94] where graph navigation was a prime focus for the language.

Additionally, GraphDB supports graph pattern matching and the ability to search for shortest paths in a graph database. GraphDB also uses regular expressions defined over edge labels of a graph database to search for paths. WEB query language was proposed in [95] support genome research where graph theory-based concepts are used to perform DNA sequencing. This language provides predefined query templates for users to extract data stored in a graph database. WEB query language uses a limited form of regular expressions to search for paths.

Graph data models and query languages in mid to late 1990s

G-Log query language was proposed in [96] where queries are written as rules that depict labeled directed graphs. These graphs are used to search for similar occurrences of sub-graphs over the database. HyperNode Query Language (HNQL) query language was proposed in [97] for querying hypernode graph data model. In hypernode graphs, nodes can encapsulate another graph [98]. The query language HNQL has operators such as sequential composition, conditionals, for and while loops. Sequential compositions are used for defining paths while for and while loops are used for searching multiple occurrences of paths in a hypernode graph. Lorel query language was proposed in [99] for querying object-oriented and semi-structured data model. Lorel has a SQL like syntax and uses regular expressions to search paths over a graph database.

Graph data models and query languages in early to mid 2000s

Struql query language was proposed in [100] for retrieving data stored in web pages. The query language provides templates for users to specify the search criteria. The output of a query written in Struql is a sub-graph, and regular expressions defined over edge labels are used for graph navigation. UnQL was a functional query language proposed in [101] for querying semi-structured data. UnQL is restricted to be used on tree structured data and uses regular expressions for graph navigation. Hyperlog query language for querying hypernode graph data model was proposed in [102]. Hyperlog uses templates to describe labeled and directed graphs. A query in hyperlog is a set of rules which are used to find sub-graphs. Hyperlog does not use regular expressions for defining paths and supports operators such as composition, conditional constructs, and iteration for graph navigation. Glide query language was proposed in [103] support graph pattern matching and navigation. Glide borrows features from XPath and uses regular expressions for graph navigation.

Graph data models and query languages in mid to late 2000s

A query language for YAGO/NAGA graph data model was proposed in [104]. A query in this approach is expressed as regular expressions defined over the graph database's edge labels. Furthermore, answers are ranked based on the depth of information gathered and the compactness of the query. SocialScope query language was proposed in [105] for presenting and querying information related to search engines such as Yahoo.

Information is stored on a social network graph that has attributed nodes and edges. A uniform algebraic framework is used for retrieving information by using operations such as node selection, edge selection, union, intersection, difference, composition, semi-join, and aggregation. Regular expressions are not used for graph navigation. BiQL query language for querying social network graphs was proposed in [106]. The query language is based on SQL and uses path expressions in the from clause to specify a graph pattern. Path expressions are formed by conjunctive edges, which collectively form a graph pattern. A pattern-matching expression then searches for the occurrences of graph patterns. BiQL supports graph navigation by expressing path expressions as regular expressions defined over the graph database's edge labels. Similar to BiQL in [107] SoSQL query language is proposed. In SoSQL a special PATH clause is used to define a path connecting two nodes. The path comprises several subpaths. In essence, the smallest subpath is an edge of the graph database. The path is made up by concatenating multiple subpaths. Hence regular expressions in the restricted form are used in this language.

Graph data models and query languages in the last decade

GraphQL query language proposed in [108] extends relational algebra. The selection operator is generalized to support graph pattern matching, and the composition operator is used to output the matched graph. GraphQL does not support graph navigation queries. SNQL query language for social network domain was proposed in [109]. SNQL is based on GraphLog and uses regular expressions for graph navigation. G-SPARQL query language for RDF data model was proposed in [110] covers features such as graph pattern matching, shortest path and graph navigation. G-SPARQL uses regular expressions for graph navigation.

2.2.2 Graph data models and query languages proposed by industry

In the context of industry graph databases query languages vary upon the underlying data model used for storing data.

Graph data models and query languages in late 1990s and early 2000s

An RDF data model can be considered as a single table containing three columns (the subject, predicate and object), with indexing to support the traversal and enumeration of predicates (relationships) for a given subject. RDF support a standard query language called a SPARQL [17]. SPARQL has a syntax like SQL and support querying features such as graph pattern matching and graph navigation. Navigational queries are expressed using regular expressions.

Graph data models and query languages in the last decade

Labeled property graph databases are much simpler than RDF. They only consist of nodes and edges, with each node and edge being a simple data structure that consists of keys and values. Gremlin [111] is a query language for labeled property graph databases and supports graph navigation using the repeat operator. OrientQL [112] is a query language for the OrientDB graph database, which provides full SQL support. OrientQL supports graph-based querying by providing the traverse operator, which supports finding repeating patterns for graph navigation. Cypher is a query language proposed by Neo4j [51]. It supports graph pattern matching and a restricted form of graph navigation. Navigational queries are expressed as regular expressions; however, only a few regular expressions can be expressed in Cypher. Oracle proposed the Property Graph Query Language [52] which supports graph pattern matching and graph navigation. PGQL is influenced by SQL and Cypher as syntax wise PGQL is very similar to SQL and

Cypher. Navigational queries are expressed as regular expressions over the edge labels, and PGQL supports more regular expressions than Cypher. Both Cypher and PGQL operate on the labeled property graph data model.

2.2.3 Findings of the survey

All graph data models use slight variations of the basic graph data structure. Examples include graph data models proposed by academia such as GOOD [12], Gram [90], GraphDB [94], HyperNode [102] and GDM [113]. Commercial graph databases such as RDF by W3C [114] use directed and labeled graphs while Neo4j [51], Oracle [52] use directed, labeled, and attributed graphs which are also known as labeled property graph data model [27]. Major query languages focus on graph pattern matching and graph navigation as core features. Furthermore, a majority of the query languages use regular expressions to support graph navigation.

Findings from the survey lead us to formulate RQ2-RQ4. Answering these research questions enabled us to analyze the current body of knowledge related to graph databases, thus obtaining a deeper understanding of the research area. Insights obtained by answering RQ2-RQ4 enabled us to formally design and develop novel formalisms of (CQT/UCQT) and our formal algebra FLASc. Findings for RQ2-RQ4 are presented in the subsequent sections.

2.3 Core features of graph query languages

The Linked Data Benchmark Council (LDBC) [4] suggests that graph pattern matching, graph navigation, shortest path search, graph construction, and graph clustering [1, 26] are core features of every graph query language. Graph navigation is an extension of graph pattern matching, used typically to identify valid paths between node pairs in a graph database. In [1], 40 core use cases for graph query languages from LDBC meetings are presented, out of which 36, 34 and 32 use cases related to graph navigation, graph construction, and graph pattern matching, respectively. Furthermore, in several other works, graph pattern matching and navigation are identified as the critical core features of every graph query language [25, 28, 24].

2.4 Theoretical formalisms for graph database query languages

2.4.1 Formalisms for graph pattern matching

The most basic formalism used by existing query languages for defining graph pattern matching is that of conjunctive queries and union of conjunctive queries [32, 25]. Conjunctive queries is the most basic query language for databases [40, 115, 33].

Conjunctive queries

Conjunctive queries are logical formulas written in a restricted form of first-order logic [66]. Conjunctive queries consist of atomic formulas and/or relations that can be combined by only using conjunction and existential operators of first order logic [116]. In graph pattern matching queries, relations represent edges and atomic formulas to represent some conditions related to nodes and edges of a graph database.

Union of conjunctive queries

Conjunctive queries do not support disjunction between relations and atomic formulas. Hence the union of conjunctive queries have been proposed in the literature [66, 58]. Union of conjunctive queries allow the application of disjunction operator for combining individual conjunctive queries [40].

2.4.2 Formalisms for graph navigation

Graph database queries are navigational in nature; thus, there are many formalisms proposed for graph navigation in the literature. Many of the proposed formalisms are based on the data structure of data words. Figueira [117] defines *data words* as a finite string made up of letters from a set of alphabets and a datum. The formalisms based on data words are as follows.

Data Paths

The formalism of data paths based on data words is proposed in [118]. Data path represents a sequence of data values and alphabets of the English language; for example, $1a2b3c4d1$ represents a data path [37]. In this example, numbers represent nodes, and alphabets represent edge labels. The discussed example represents a cyclic graph where an outgoing edge is labeled as a from node holding data value 1 to node holding data value 2 and a d labeled incoming edge from node holding data value 4 to node holding data value 1. There are various formalisms similar to data paths that have been proposed in the literature.

Register Automata

Formalism based on register automata [119] extends finite state automata (FSA) by adding a finite set of registers to store data values. They work in ways that are similar to FSA and use registers to compare values associated with nodes. However, authors in [120] suggest that register automata are not closed under the complement operator, making them less expressive than regular languages, which are closed under complement.

Pebble Automata

Pebble automata [121] are specifically designed for navigating through the tree data structure. A pebble automaton uses pebbles to indicate that a particular node has been visited. A pebble can be viewed as a counter which is incremented every time a new node is visited. Stack-based restrictions are imposed in pebble automata to ensure expected behaviour. Pebbles can be dropped and lifted from the current node, and the last-placed pebble acts as a head of the automaton. Furthermore, data values at the node with already placed pebbles are used to compare the current node's data values.

First-Order and Monadic Second-Order Logic

The use of formalisms such as first-order logic and second-order logic has been proposed by [122] for querying tree-structured graph database represented as XML documents. Predicate logic is used to check the existence of values associated with nodes by expressing predicates in XPath. A formal definition of each query written in XPath is expressed in equivalent first-order and second-order logic by using logical operators and universal $\forall$ and existential $\exists$ quantifiers. However, the query containment for first-order logic is undecidable and restrictively decidable for Monadic Second-Order Logic.

Linear Temporal Logic

Formalisms based on Linear temporal Logic (LTL) have been proposed as an alternative to first-order logic in [120]. LTL based formalism for querying graph databases use a freeze operator and register automata. The freeze operator uses a register to store and compare values stored inside the data word. However, query evaluation for this formalism over graph databases is undecidable [37].

XPath Fragements

The use of XPath to study the application of Downward Data (DD) automaton, Alternating Top-down Tree on Register Automata (ATRA) and Bottom-Up alternating Tree Automata with one register (BUDA) on XML database has been proposed in [117]. Authors suggest that each automaton's limitations, such as DD automaton, are closed under boolean operators but are limited when comparing data values in the leaf nodes. The other two automatons are not closed under complement operations.

Regular Expressions

Regular expression is an algebra for describing some patterns that can be accepted by a finite state automaton [123]. Regular expressions provide three basic operators of concatenation, union, and Kleene star. These operators are used to formulate path expressions over the set of edge labels of a graph database. The path expressions are then used to search data over a graph database.

Why regular expressions are used as a formalism for graph databases

Formalism, such as first-order logic and linear temporal logic, have an undecidable query containment problem. Register automata have limited expressiveness as they are not closed under complement. Moreover, [37] suggest that formalism based on data words are not suitable for graph querying. Authors in [37] propose extending register automata to create Regular Data Path Queries (RDPQs); however, they assert that RDPQs are not sufficient to be used in a practical query language for graphs. For graph-structured data, queries that allow users to specify the types of paths they are interested in have always played a central role. Most commonly, the specification of such paths has been utilizing regular expressions over the alphabet of edge labels [99].

2.4.3 Use of regular expressions based formalisms for querying graph databases

Formalisms based on regular expressions used to query graph databases are called regular path queries (RPQ). They use three basic operators of concatenation, union and Kleene star over a graph database's edge labels. In order to increase the expressiveness of RPQs that have been extended to include the inverse operator and are called two-way regular path queries (2RPQ) [115]. Formalisms of RPQs and 2RPQs are not expressive enough as they cannot be used to search paths that contain branches on intermediate nodes [34]. Therefore, formalism of nested regular expressions (NRE) has been proposed in [30]. NREs extend 2RPQs by adding a nesting operator; furthermore, Kleene star operator can be applied along with nesting operator to search for paths with branches in an unbounded manner.

In order to increase the expressive power of existing formalisms for querying graph databases, subsequent works [124, 115, 125, 64, 62, 33, 66] propose combining formalisms for graph pattern matching and graph navigation queries. These extensions result in a more expressive class of theoretical language formalisms such as conjunctive regular path queries (CRPQ), conjunctive two-way regular path queries (C2RPQ), the union of conjunctive regular path queries (UCRPQ), the union of conjunctive two-way regular path queries (UC2RPQ), extended conjunctive regular path queries (ECRPQ) with regular and rational relations. Formalisms of C2RPQs and NREs are incomparable in terms of expressiveness [62] since C2RPQs do not allow the application of Kleene star over branching operator while NREs cannot express cyclic graph structures. Therefore, authors in [126, 36, 34] propose conjunctive nested regular expressions (CNRE) and union of conjunctive nested two-way regular path queries (UCN2RPQ) [127].

Limitations of regular expression based formalisms

Formalisms based on RPQs, 2RPQs and NREs are not expressive enough since they cannot be used to search for the path containing cyclic structures in an unbounded manner. A primary reason for this is because in existing formalisms conjunction operator is not closed under the Kleene star operator. Therefore, graph query language formalism based on Tarski's algebra has been studied in [128, 35, 55, 74] that subsume the expressive power of formalisms such as RPQs, 2RPQs and NREs. Furthermore, graph query languages proposed by academia such as Navigational XPath [129] and GXPath [34] utilize fragments of Tarski's algebra for querying graph databases. Tarski's algebra can certainly be used as a theoretical language formalism for querying graph databases. However, formalism based on Tarski's algebra can only be used in graph navigation queries. Therefore, in order to address this gap, we propose the extension of conjunctive queries and union of conjunctive queries with Tarski's algebra (CQT/UCQT). These formalisms enable us to utilize the power of Tarski's algebra for graph pattern matching and graph navigation queries. We present a detailed discussion related to CQT and UCQT in Chapter 6.

2.5 Modeling approaches for graph databases

Graph databases opt to be schema-less or schema optional; hence, this section helps us identify gaps in the current literature related to graph database modeling approaches. Our survey identifies the existing studies that have addressed the *conceptual, logical* and *physical modeling* stages for graph databases.

2.5.1 Conceptual modeling

Conceptual modeling represents the initial stage for modeling a graph database. In this stage, knowledge is collected in the form of requirements and specifications related to a problem domain. Using graphs for representing knowledge was first proposed by Sowa [50, 48, 47, 49]. Furthermore, other works such as [130, 131, 132] use graphs to represent knowledge at the conceptual modeling stage. Graphs provide a natural and intuitive interface for understanding the semantics of data [50, 80]. Knowing the semantics of data is vital for understanding the overall structure of the database [44] that aids in creating, modifying, and retrieving data. In a conceptual graph schema, real-world entities are modelled as nodes. Interrelationships between those entities are modelled as binary edges [47, 132] which means that an edge in a conceptual graph schema cannot be used to connect more than two nodes. Conceptual graph schemas provide a level of abstraction that aids in the natural modeling of data [133]. They are used to define what entities belong to the database and how the information is structured [80]. Moreover, determining nodes, edges, and the direction of edges are vital for conceptual modeling [39].

2.5.2 Logical modeling

Logical modeling is used to enforce integrity constraints on the graph schema produced in the conceptual modeling stage. Integrity constraints serve as mechanisms to ensure data consistency and integrity. They are broadly classified into two categories *(i) graph entity integrity* and *(ii) semantic constraints* [134]. Graph entity integrity constraints are related to basic database design principles. These include constraints such as *key identity constraints* which ensure that nodes and edges in graph database have unique identifiers[23, 45, 133, 134, 46]. *Label uniqueness constraints* ensure that nodes and edge labels are unique [23, 44, 133, 134, 45]. Constraints such as *property data type* ensure that node and edge properties have predefined data types [44, 46] and *mandatory property* ensure that existence of some node and edge properties are compulsory [135, 44]. *Edge pattern constraints* ensure that the topology of the graph schema defined in the conceptual modeling stage is maintained in the graph database [46, 134, 135, 81, 45].

Enforcing semantic constraints requires knowledge of the problem domain captured in the graph schema designed during the conceptual modeling stage [134]. These constraints are used to guarantee the conformity of graph database with domain-specific rules and require intervention from end-users. These include *cardinality constraints* used to ensure the minimum and maximum number of edges that can exist between two nodes of a graph database [44, 46, 133, 134, 81, 84, 83]. *Path pattern constraints* ensure that a sequence of edges conforms to a path in the graph database. The use of formalisms such as Extended Conjunctive Regular Path Queries (`ECRPQs`) to enforce path pattern constraints has been proposed in [46]. However, the use of `ECRPQ` based formalisms can be problematic if appropriate evaluation semantics are not utilized in the underlying graph database and query language [72, 73, 136]. *Graph pattern constraints* ensure that creation of a graph is dependent upon the existence of a certain pre-existing graph in the graph database [46, 134, 133, 135].

Other integrity constraints discussed in literature include *type checking* to ensure that the graph schema and graph database are consistent with each other [23, 133, 135]. This constraint's primary requirement is that all nodes and edges in the graph database must conform to the nodes and edges in the graph schema. *Node/Edge property value constraints* ensure that nodes and edges in graph database are assigned values based on some predefined condition [81]. *Functional dependencies* are used to ensure if an element can be used to determine value of another element [23, 44, 133, 137, 138, 139]. However, functional dependencies cannot be represented easily in graph databases [23].

2.5.3 Physical modeling

Physical modeling represents the realization of the graph schema designed during conceptual and logical modeling into the actual database [140]. While conceptual and logical modeling can be performed independently of the database platform, physical modeling depends on the underlying database's specifications and semantics. This stage is difficult to understand by the end-users, and domain expertise such as knowledge of the query languages and database-specific application programming interfaces (APIs) are required. Physical modeling stage is of two types *(i) pre-deployment* and *(ii) post-deployment*. In the pre-deployment stage, graph schema designed during the previous two stages is used to prepare database creation scripts that capture the structure of graph schema and maintain the integrity constraints. These scripts are written in query languages specific to a particular graph database. In the post-deployment stage, the main focus is on improving the existing database's performance, reducing input/output, database maintenance, and other database administration tasks, which a database administrator usually performs.

There are two approaches discussed in literature for the pre-deployment stage: *integrated* and *layered* approach [83]. In the integrated approach, changes are made in the source code of the database system, and query languages are modified to support the enforcement of integrity constraints over the graph schema. Database creation scripts are created and directly deployed on the database platform. In the layered approach, APIs specific to the database platform creates an additional layer that communicates with the database. This consist of wrappers written in programming languages such as Java, Python that contains database creation scripts and logic to enforce the integrity constraints.

There exist many studies to support the physical modeling aspects of graph databases. For instance, [134] follow a layered approach and propose the construction of a wrapper that can be used to enforce integrity constraints, including graph and path pattern constraints over Neo4j graph database. An integrated approach to extend the source code of OrientDB to support the enforcement of integrity constraints, including uniqueness, key, cardinality, and edge degree constraints, has been studied in [81]. Similarly, the extension of Cypher query language to support additional integrity constraints such as uniqueness, node property, required edges and mandatory properties is presented in [45, 82]. A layered approach to demonstrate uniqueness integrity constraint on two different graph databases Neo4j and Apache Tinkerpop is proposed in [83]. The use of integrated and layered approach together to perform graph database manipulation operations on Neo4j graph database is proposed in [46]. Authors in [141] propose the model-driven engineering-based approach for converting and loading of UML diagrams into Tinkerpop blueprints[10]. A formal approach for designing a labeled property graph schema and demonstrate the use of GraphQL[11] to specify graph schema is proposed in [142].

2.5.4 Limitations of existing modeling approaches for graph databases

Conceptual modeling stage is vital for capturing the semantics of a problem doamin. A sound conceptual graph schema ensures that logical and physical modeling stages are also robust [143]. The graph data modeling approaches proposed so far do not provide the means to create robust conceptual graph schemas. Authors in [144, 145, 141] propose the use of existing visual modeling tools such as Entity Relationships diagrams (ERD) and Unified Modeling Language (UML) for creating conceptual graph schemas. However, the use of such approaches requires an understanding of notations that are specific to these tools. Furthermore, additional efforts are required for adopting the

[10]`https://github.com/tinkerpop/blueprints`
[11]`https://graphql.org/code/`

conceptual graph schemas generated by visual modeling tools at the logical and physical data modeling stages. Contemporary graph database vendors such as Neo4j propose using tools such as Arrow[12] for designing conceptual graph schema. However, the informal nature of such tools cannot ensure the robustness of generated conceptual graph schemas.

Therefore, we present FLASc, a simple yet sturdy formal tool that assists in the formulation of robust conceptual graph schemas which is an advancement over existing studies in graph database modeling. The majority of integrity constraints presented in the existing studies can be specified in graph schemas generated by FLASc. Furthermore, syntax and semantics of FLASc presented in this study assist in its implementation at the physical modeling stage. FLASc assists in the integration of conceptual, logical and physical modeling stages, which currently is lacking in graph database research. We present a detailed discussion related to FLASc in Chapter 8.

2.6 Conclusions

The literature review presented in this chapter enables us to identify gaps in the existing body of knowledge related to querying and modeling approaches used in graph databases. By answering RQ1, we observed common characteristics of graph data models and associated query languages. RQ2 helped identify the two core features of graph pattern matching and graph navigation that are desirable in every graph query language. Answering RQ3 helped us to identify that existing theoretical language formalisms are not expressive enough. Therefore, existing formalisms cannot be used to objectively and comprehensively compare practical graph query languages. We in Chapter 6 discuss the formal extension of conjunctive queries and union of conjunctive queries with Tarski's algebra. We present an integrated framework based on CQT and UCQT that

[12]`http://guides.neo4j.com/arrows`

is then used to compare practical graph query languages on the core features of graph pattern matching and graph navigation. This process is demonstrated by comparing two contemporary graph query languages Cypher and PGQL. RQ4 was designed to identify existing approaches used for modeling graph databases. Findings from the literature review suggest that current approaches used for modeling graph databases lack the integration of conceptual, logical, and physical modeling stages. The conceptual modeling stage is fundamental for capturing the semantics of a problem domain. Furthermore, this stage is foundational in logical and physical modeling stages. Therefore, in Chapter 8 we present our formal algebra FLASc that can be used to formulate robust conceptual and logical graph schemas. We also demonstrate the merger of FLASc with the well-known Extract-Transform-Load design pattern that enables us to integrate the conceptual, logical, and physical modeling stages for graph databases.

Chapter 3

Introduction to Manuscript 1

Graph databases provide better support for highly interconnected datasets than relational databases. However, labeled property graph databases, which have become increasingly popular, are schema-optional, making them prone to data corruption, especially when new users switch from relational databases to graph databases. In this work, we provide a schema-driven formalism for graph databases. This formalism enables schema-driven loading of graph databases from other sources, such as relational databases. Also, this formalism enables schema-driven data analytics that allows for a more structured analysis of data stored in graph databases. Such analytics are based on a boilerplate approach allowing users who are not experts in the use of graph database query languages to carry out analytics efficiently. We showcase the utility of the proposed formalism by considering a case study from Airbnb for illustrating schema-based loading procedures. The proposed schema-driven analytics process is illustrated using another case study from an industrial cyber-physical systems standard. Overall, the schema-driven formalism provides several useful features, such as preventing both data corruption and long-term degradation of graph database structures.

Chapter 4

A Schema-First Formalism for Labeled Property Graph Databases: Enabling Structured Data Loading and Analytics (Manuscript 1)

4.1 Introduction

Graph databases have become increasingly popular as they provide better support for highly interconnected datasets [7, 8]. Highly interconnected datasets are found in most big data applications including social networks [20, 28], bioinformatics [29] and astronomy [30, 31]. Such data can be more easily expressed using the nodes and edges of a graph database than the table-based structure offered by relational databases.

Labeled Property Graphs (LPGs) have gained popularity over other graph database variants due to their flexibility [146]. Graph database variants like Resource Description Framework (RDF) are schema-dependent [114, 17, 18, 19, 20, 21]. RDFs enable the modeling of the entities of a knowledge domain as classes that have properties. RDFs use concepts such as inheritance to model relationships between classes. Newer big data application domains such as social media increased the demand for a less restricting data model than RDFs [22]. This led to the creation of LPGs which provide high flexibility and expressiveness by allowing nodes and edges to store additional arbitrary attributes [23, 1, 25, 27]. XML also provides a schema-dependent structure for graph databases, but this structure only supports trees. Trees are more restrictive subsets of graphs, and hence XML has limited expressiveness when compared to both RDFs and LPGs.

A key challenge in migrating to graph databases, and more precisely, LPGs, is the fact that they are *schema-optional*, which leaves them more susceptible to data corruption. In relational databases, a schema defines precisely how data needs to be organized into tables. Schema are an integral part of any database [27] as it assists in categorizing and relating data [147], and inferring patterns [148] for efficient data extraction and analytics. Moreover, any changes to a database must comply with the schema. In graph databases, a schema describes the allowed node and edge *types*. However, LPGs like Neo4j are schema-optional. Consequently, users can dynamically add a new node or edge types without being restricted by the schema. While this feature provides flexibility, it can also degrade the graph database's structure over time. Currently, organizations moving towards using LPGs are depending primarily on designer/user skills for preventing data corruption.

We propose a schema-driven formalism for LPGs, which provides better data integrity when migrating data from other sources. Also, this formalism allows more structured access to the data stored within a graph database, through the use of template queries based on the schema. The proposed formalism only restricts the *topology* of a graph database, which constrains the use of node and edge types. This formalism does not constrain how data is stored within the node and edge attributes of LPGs, ensuring that the schema-first approach does not sacrifice the key advantage of increased expressiveness that LPGs provide over RDFs and XML. The primary contributions of this paper are:

1. A formal description of Labeled Property Graph databases based on graph and set theory, presented in Sec. 4.2.
2. A schema-driven approach for importing or loading data from other sources into a LPG database, based on the proposed formalism. This approach, presented in Sec. 4.3, prevents data corruption during the loading process. This schema-driven loading process is illustrated through an Airbnb example.
3. A schema-driven approach for data analytics on LPG databases, based on the proposed formalism, presented in Sec. 4.4. We illustrate this approach through a case study on the use of industrial agents in manufacturing systems.

Literature related to each of the three contributions listed above is reviewed and discussed in the corresponding sections of this paper. Concluding remarks and future directions for this research appear in Sec. 4.5.

4.2 Formalisation of Labeled Property Graph Databases

We start with a few basic definitions related to graphs that are used throughout this article.

Definition 1 (Directed Multigraph) *A directed multigraph $\mathcal{G}$ is defined as a pair $(\mathcal{N}, \mathcal{E})$ where $\mathcal{N} = \{n_1 .... n_k\}$ is a finite set of nodes and $\mathcal{E} = \{e_1 .... e_l\}$ is a finite set of edges. Two associated functions, $s : \mathcal{E} \to \mathcal{N}$ and $t : \mathcal{E} \to \mathcal{N}$, map edges to unique source and target nodes, respectively.*

Directed multigraphs are graphs that are (a) *directed*, such that edges have unique source and target nodes, and (b) allow multiple edges between nodes in the graph (nodes with the same source and target nodes are allowed). As we will see later, graph databases are directed multigraphs (henceforth, graphs) with certain restrictions. We use the short hand $n_1 \rightarrow n_2$ to represent an edge e where $s(e) = n_1$ and $t(e) = n_2$.

Graphs can contain *labels* over nodes and edges. A labeling is simply a map $f : S1 \mapsto S2$ such that for every element $a \in S1$, there is a unique element $f(a) \in S2$. We can now define an *edge*-labeled graph as follows. We use the shorthand $S1 \mapsto S2$ to define labelings.

Definition 2 (`Edge-Labeled Graph`) *A graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is called an edge-labeled graph if there exists a labeling $f : \mathcal{E} \mapsto EL$ which maps all edges to labels in a set of edge labels EL. Edge labels are described by the short-hand $n_1 \xrightarrow{l} n_2$ for any $(n_1, n_2) \in \mathcal{E}$ and $f(n_1, n_2) = l$.*

Similarly, we can define a node labeled graph.

Definition 3 (`Node-Labeled Graph`) *A graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is called a node-labeled graph if there exists a labeling $f : \mathcal{N} \mapsto L$ which maps all nodes to labels in a set of node labels L. Node labels are described by the short-hand $n.lbl = l$ for any $n \in \mathcal{N}$ and $f(n) = l$.*

A graph can be both edge and node labeled.

4.2.1 Graph Schema

A graph schema describes the logical structure of a graph database.

Definition 4 (Graph Schema) *A graph schema $\mathcal{G}_S$ is an edge labeled graph $(L_{\mathcal{N}}, \mathcal{E}_S)$ with the edge labeling $\mathcal{E} \mapsto L_{\mathcal{E}}$, where $L_{\mathcal{E}}$ is a set of allowed edge labels over the graph schema.*

A graph schema captures the structural or topographical restrictions on a graph database. The nodes in a graph schema indicate the allowed *node types* that a graph database following the schema can contain.

Example 1 *Figure 4.1 shows a graph schema for the Airbnb dataset. This schema contains six nodes, including* `LISTING` *and* `HOST`*, which are the permitted node types in any database conforming to this schema.*

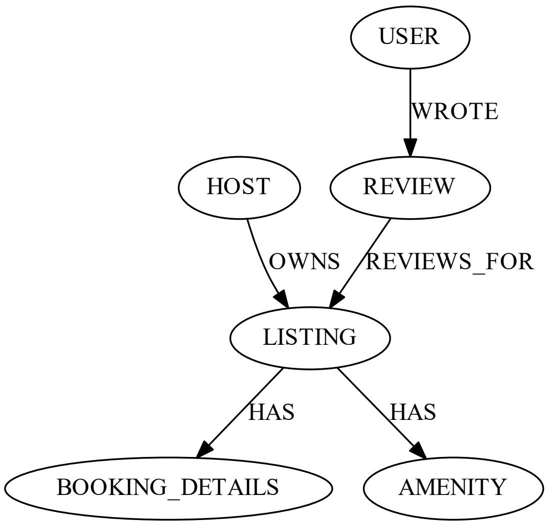


Figure 4.1: A Graph Schema for the Airbnb dataset

The edges of a graph schema describe correct *edge types* that a graph database following the schema can contain. Edges in a graph schema are directed and are constraint to certain edge types in the dataset.

Example 2 *Figure 4.1 shows the permitted edge types in any database conforming to this schema. For example, the edge* $\texttt{HOST} \xrightarrow{\texttt{owns}} \texttt{LISTING}$ *is labeled with* `owns`*, indicating that in the graph database, nodes of type* `HOST` *can have outgoing edges to nodes of type* `LISTING` *and that all such edges must be labeled by* `own`*.*

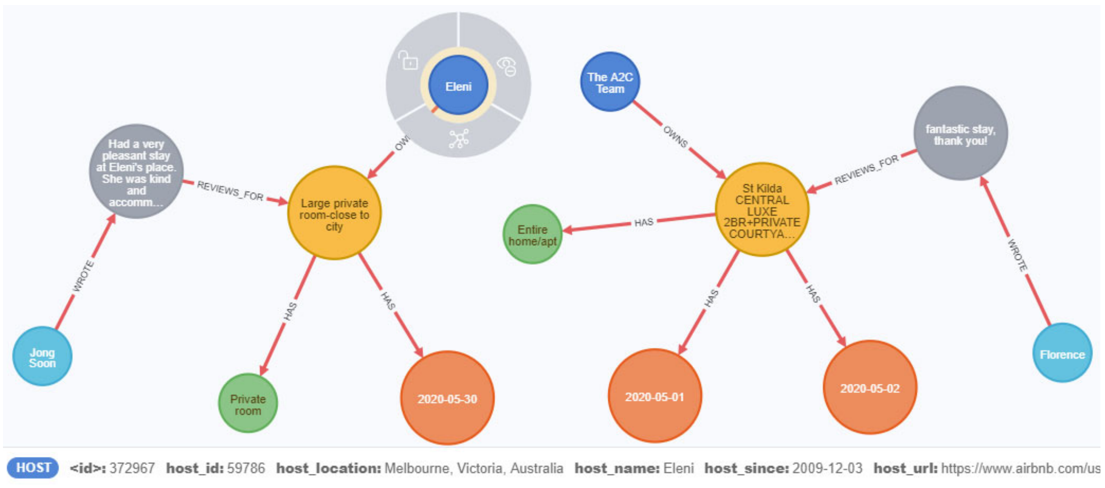
The label and direction of an edge in a graph schema assists in capturing the nature of the relationship between two nodes types. Furthermore, such constraints are necessary for enforcing topological restrictions over the graph database. Overall, the schema provides an intuitive way of knowledge representation and aids in modeling various problem domains.

4.2.2 Graph Database

A graph database structures a given dataset following a graph schema.

Definition 5 (`Graph Database`) *A graph database $\mathcal{G}_{DB}$, defined over a graph schema $\mathcal{G}_S = (L_\mathcal{N}, \mathcal{E}_S)$ with the edge labeling $\mathcal{E} \mapsto L_\mathcal{E}$, is an edge-labeled and node-labeled graph $(\mathcal{N}_D, \mathcal{E}_D)$ such that:*

- *The node labeling $\mathcal{N}_D \mapsto \mathcal{N}$ labels each node in the graph database with a node type defined in the schema. For each node $n \in \mathcal{N}_D$, there must exist a node type $n_s \in \mathcal{N}$, such that $n.lbl = n_s$.*
- *The edge labeling $\mathcal{E}_D \mapsto L_\mathcal{E}$ labels each edge in the graph database with an edge type defined in the schema. For every edge $(n_1 \xrightarrow{l} n_2) \in \mathcal{E}_D$ there must exist a corresponding edge $(n_1' \xrightarrow{l} n_2') \in \mathcal{E}_S$ such that $n_1.lbl = n_1'$ and $n_2.lbl = n_2'$.*


Eleni
The A2C Team
Had a very pleasant stay at Eleni's place. She was kind and accomm...
REVIEWS_FOR
OW
Large private room-close to city
OWNS
St Kilda CENTRAL LUXE 2BR+PRIVATE COURTYA...
REVIEWS_FOR
fantastic stay, thank you!
HAS
Entire home/apt
WROTE
HAS
HAS
HAS
HAS
WROTE
Jong Soon
Private room
2020-05-30
2020-05-01
2020-05-02
Florence

HOST <id>: 372967 host_id: 59786 host_location: Melbourne, Victoria, Australia host_name: Eleni host_since: 2009-12-03 host_url: https://www.airbnb.com/us

Figure 4.2: A sample Graph Database created from Airbnb Dataset

A graph database contains node types and edge types allowed by its schema. A graph database structures a given data-set following a graph schema. The edges of a graph database which also represent the connections between various nodes are formed based on the edges defined in the graph schema.

A labeled property graph (LPG) database contains data stored within its nodes and edges, in the form of attributes or properties. A graph database based on labeled property graph data model provides internal structures inside nodes and edges. That are used to store additional information related to nodes and edges in the form of properties (hence, the name labeled *property* graph where each attribute related to either a node or an edge is a pair of key and value.

Definition 6 (Data property) *A data property a is a variable of any data type, and $\overline{a}$ is any valid assignment of a. Given a set of properties $A = \{a_1, \ldots\}$, and a subset $A' \subseteq A$, $\overline{A'}$ is a set of unique valid assignments to the properties in A'. $\overline{2^A}$ is defined as the set of all valid assignments of all possible subsets of the set of properties A.*

Definition 7 (Labeled Property Graph Database) *A graph database $\mathcal{G}_{DB} = (\mathcal{N}_D, \mathcal{E}_D)$ is a labeled property graph when, given a set of data properties A, there exist two labelings $A_N : \mathcal{N}_D \mapsto \overline{2^A}$ and $A_E : \mathcal{E}_D \mapsto \overline{2^A}$ assigning properties and values to each node and edge in the graph database.*

Figure 4.2 shows a labeled property graph database created for the Airbnb dataset. The structure of the graph database is based on the graph schema shown in figure 4.1. Based on Definition 7, each node and edge of a graph database has some data properties associated with it. We discuss these aspects of LPGs with a few examples.

Example 3 *Figure 4.2 shows that the selected node `Eleni` has the node type `HOST`. The properties associated with the node `Eleni` are $\{$`id, host_id, host_location, host_name, host_since`$\}$. The* *values assigned to the attributes are accessed by using the dot notation such as* `Eleni.host_id` = 59786. *Each node and edge in the LPG contains values for an arbitrary set of properties.*

Based on Definition 5, the structure of the database shown in Figure 4.2 follows the graph schema in Figure 4.1. Edges in the graph database must ,therefore, conform to the edge types allowed by the schema.

Example 4 *In Figure 4.2, the node* `TheA2CTeam` *has an outgoing edge to the node* `St Kilda` *and the edge is labeled with edge type* `OWNS`*. The node* `St Kilda` *is of the type* `LISTING` *while* `TheA2CTeam` *is of type* `HOST`*. Hence, the corresponding edge type must be labeled as* `OWNS`*, as that is the only edge type allowed in the schema between node types* `LISTING` *and* `HOST`*.*

Every edge in a labeled property graph database can have properties associated with it.

Example 5 *The edge* `e` *from* `TheA2CTeam` *to* `St Kilda` *contains one property* `"since"` *with the value* 2004 *that signifies the year since host has owned a listing. The value of this property can be accessed using the dot notation, i.e.,* `e.since` = 2004*.*

The notion of a schema for labeled property graphs proposed previously in [7] is extended into a formal model for property graphs in [27]. A conversion between RDF and property graph data model by using a standard metadata model is proposed in [149]. A schema-driven approach for performing traversals on graph databases appears in [24]. An application domain driven approach for creating graph schemas is presented in [145]. An extension of relational schemas into a meta model that can be converted into graph models appears in [150]. Another model-driven approach converts a property graph data model into a relational data model [146]. However, such existing techniques and formalisms for schemas are only partially used to design data loading procedures and perform data analytics, as proposed in this paper. Furthermore, our schema model only restricts the topology of a graph database, ensuring that LPGs retain the flexibility offered by allowing nodes and edges to contain arbitrary property values.

4.3 Schema-Driven Loading of Graph Databases

Loading new datasets into a graph database require sufficient meta-data, ideally captured as the schema. Loading procedures involve transforming and reorganizing data present in a legacy format, such as files or relational databases, into the nodes and edges structure supported by graph databases. Such transformations require sufficient knowledge or meta-data about the *structure* of the dataset to being processed. The meta-data governs how elements of the original dataset are mapped to the nodes and edges of the graph database, which is precisely the information contained in the schema. Hence, it is intuitive to plan loading processes by first constructing a schema.

Currently, the process of deciding on the structure of the graph database into which a dataset is loaded is primarily informal. Ad-hoc approaches to prepare for loading operations may include preliminary decisions on the structure of the graph database. These initial structures are based mostly on the existing format of the dataset being considered, and may not be optimal for the new graph database.

We propose a schema-driven loading strategy for LPGs. In this approach, a graph schema is defined based on an assessment of the existing format of the dataset being imported. This schema can be validated by users and can also be analysed using automatic tools before loading procedures are initiated.

We illustrate the proposed schema-driven loading process using a case study of Airbnb. Airbnb's datasets are available to be downloaded from their website under a Creative Commons license in comma-separated values (CSV) file format [87]. The Airbnb dataset consists of three CSV files that contain information related to listings, reviews and calendar data. The listings file contains information related to hosts, houses, amenities provided in the listing, the location of the house, etc. The reviews file contains information related to the users who have stayed in the listings and provided feedback in the form of reviews. The calendar file contains information related to *booking details* such as pricing and occupancy. These files contain multiple lines (rows) of data, where each row contains a comma-separated list of values. For instance, the example of a CSV file containing information related to listings from Airbnb's data is shown in Table 4.1.

Table 4.1: Sample data from listing.csv in the Airbnb dataset

Host Name	**Listing ID**	**Listing Name**	**Room Type**	**Street**	**Host ID**
Manju	9835	Beautiful Room & House	Private room	Bulleen, VIC, Australia	33057
Lindsay	10803	Room in Cool Deco Apartment in Brunswick East	Private room	Brunswick East, VIC, Australia	38901
Eleni	15246	Large private room-close to city	Private room	Thornbury, VIC, Australia	59786
Eleni	68482	Charming house inner Melbourne	Entire home/apt	Thornbury, VIC, Australia	59786

The first stage in the proposed process is an analysis of the existing dataset. To create a graph schema for the Airbnb dataset some information related to the structure of the graph is required beforehand. This information describes which entities are

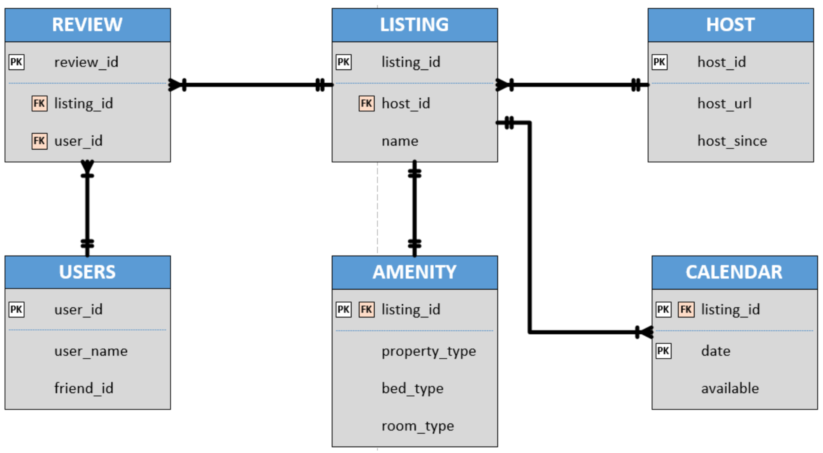


Figure 4.3: Entity Relationship Diagram for the Airbnb dataset

represented as nodes and how these entities are connected to one another. We use the Entity-Relationship Diagram (ERD) for the Airbnb dataset shown in Figure 4.3 that presents a logical view of the dataset. The ERD can be visualized as a graph $\mathcal{G}_{ERD} = (\mathcal{N}_R, \mathcal{E}_R)$, where $\mathcal{N}_R$ represent tables in the ERD and edges in $\mathcal{E}_R$ represent primary key - foreign key relationships between the tables. In Figure 4.3 the nodes and edges are determined as follows:

- $\mathcal{N}_R$ = {`REVIEW, LISTING,HOST,BOOKING-DETAILS, AMENITY,USERS`}.
- $\mathcal{E}_R$ ={(`REVIEW,LISTING`), (`REVIEW,USERS`), (`HOST,LISTING`), (`AMENITY,LISTING`), (`BOOKING-DETAILS,LISTING`)}

The second stage is the creation of a LPG schema. For the Airbnb case study, the CSV files structures are mapped onto nodes and edges of the graph database. The node and edge types are captured into the schema, which can then be verified manually by users. For the Airbnb dataset, the LPG schema shown in Figure 4.1 is constructed by first choosing which fields in the ERD are represented as nodes. While this process is primarily user-guided, we can use certain patterns to decide on node types. Firstly, node types should represent data that is expected to have multiple relationships with other nodes, and not represent data that is associated with single entities and hence better stored as properties. For example, for Airbnb, since hosts can own multiple listings, it is useful to create two separate node types (`HOST` and `LISTING` in Figure 4.1). However, information such as address are specific to every listing, and can be stored as properties. Secondly, node types can be determined based on the expected analytics on the database. For example, the node type `AMENITY` in Figure 4.1 represents the fact that users may want to search listings by amenities.

In order to define the direction of edges in the graph schema we use the `subject-predicate-object` format from semantic web [151] to decide the direction between the entity nodes derived from the ERD. For example, the edge $\texttt{HOST} \xrightarrow{\texttt{OWNS}} \texttt{LISTING}$ signifies the ownership relationship between hosts and listings. Edge direction is important because data can lose its meaning if relationships are created incorrectly. For example, a listing cannot own a host. Overall, the following edge types are allowed by the schema in Figure 4.1.

- $\texttt{USER} \xrightarrow{\texttt{WROTE}} \texttt{REVIEWS}$
- $\texttt{REVIEW} \xrightarrow{\texttt{REVIEWS\_FOR}} \texttt{LISTING}$
- $\texttt{LISTING} \xrightarrow{\texttt{HAS}} \texttt{AMENITY}$
- $\texttt{LISTING} \xrightarrow{\texttt{HAS}} \texttt{BOOKING-DETAILS}$
- $\texttt{HOST} \xrightarrow{\texttt{OWNS}} \texttt{LISTING}$

The third stage is *scripts creation* where graph schema is used to write scripts for loading procedures. For the Airbnb case study, this involves automating the process of loading values from the CSV files into the nodes (and edges) of the graph database. For this research, we have used Neo4j graph database and the associated Cypher query language. The data loading scripts in Cypher have to be designed to conform to the schema designed in the previous step. The `listing.csv` file contains information about *hosts, listings* and *amenities*. However, the graph schema in Figure 4.1 requires this data to reside as separate nodes in the graph database. Furthermore, *review.csv* contains information about *users* who have stayed at different *listings* and/or have provided *reviews*. The *calendar.csv* file contains booking information related to listings. The interrelated information contained within these files has to be organised as per the structure of the schema, which governs the structure of the Cypher scripts. There are three major concerns while constructing load scripts in Cypher.

- Uniqueness constraints have to be defined on node IDs.

- Nodes must be attributed to data in key-value format.
- Edges must be constructed between nodes based on the topology described using the graph schema.

The sample listing file as shown in Table 4.1 has *Listing ID* associated with each listing which serves as its unique identifier. Therefore, before creating the listing nodes, the uniqueness constraint has to be established to reduce the chances of data corruption. Neo4j provides this feature of enforcing uniqueness constraints. As an example, this can be achieved by running the following query written in Cypher.

QUERY 1:

```
CREATE CONSTRAINT ON (list:LISTING)
ASSERT list.listing_id IS UNIQUE
```

The uniqueness constraint ensures that Cypher does not create multiple `LISTING` nodes with the same listing ID. To start loading data related from `listings.csv`, the following Cypher query is used.

QUERY 2:

```
LOAD CSV WITH HEADERS FROM
 "http://data.insideairbnb.com/australia/vic/melbourne
/2019-07-09/visualisations/listings.csv" AS row
WITH DISTINCT row.id AS listing_id
MERGE (list:LISTING {listing_id:toInteger(listing_id)})
```

The Cypher query establishes connection with the `listings.csv` file and then creates nodes with `LISTING` node type. All nodes are unique and contain listing ID as a property. To add other properties to the nodes of type `LISTING`, additional data has to be loaded from `listings.csv` in the form of properties stored as key-value pairs. This can be achieved by Query 3 that assigns property values to only those `LISTING` nodes that match the data in the CSV file via the listing id.

Nodes of type `HOST`, `AMENITY`, `REVIEW`, `USER` and `BOOKING-DETAILS` are created by using a similar procedure than the one illustrated through Queries 1, 2 and

QUERY 3:

```
LOAD CSV WITH HEADERS FROM
 "http://data.insideairbnb.com/australia/vic/melbourne
/2019-07-09/visualisations/listings.csv" AS row
MATCH (list:LISTING )
WHERE list.listing_id = toInteger(row.id)
SET list.listing_url = row.listing_url,
list.name = row.name,
list.summary = row.summary,
list.space = row.space
list.neighbourhood = row.neighbourhood
```

3. Next, edges between nodes are established. For example, Query 4 is used to create edges between nodes of type `HOST` and `LISTING`. Each edge created by using Query 4 is labeled as `OWNS` and represents a valid edge type in the graph schema. Edges are created based on the primary key-foreign key relationship between entities `LISTING` and `HOST` of the ERD for Airbnb dataset. As shown in Figure 4.3 the cardinality between tables `LISTING` and `HOST` is one-to-many which means that a *host* can *own* many *listings* while each *listing* can only be *owned by* one host. For example, in table 4.1 *host* `Eleni` has two associated *listings*. To capture such relationships in graph databases a Cypher query such as Query 4 uses the **`WHERE`** clause to create edges that are based on the primary key-foreign key relationships. Figure 4.4 shows a sub graph which illustrates a one to many relationship between node type *host* labeled as `Eleni` and two node types *listing* labeled as (Charming house, Large private)

QUERY 4:

```
LOAD CSV WITH HEADERS FROM
 "http://data.insideairbnb.com/australia/vic/melbourne
/2019-07-09/visualisations/listings.csv" AS row
MATCH (list:LISTING), (host:HOST)
WHERE list.listing_id = toInteger(row.id)
AND host.host_id = toInteger(row.host_id)
MERGE (host)-[:OWNS {since:row.host_since}]->(list)
```

Entities `AMENITY` and `LISTING` as shown in figure 4.3 have a one-to-one relationship in the ERD. Therefore, Query 5 is used to create edges between nodes of type

`AMENITY` and `LISTING`. Figure 4.4 illustrates one to one relationship between nodes types *listing* labeled as (Charming house, Large private) and *amenity* labeled as (Entire home, Private room).

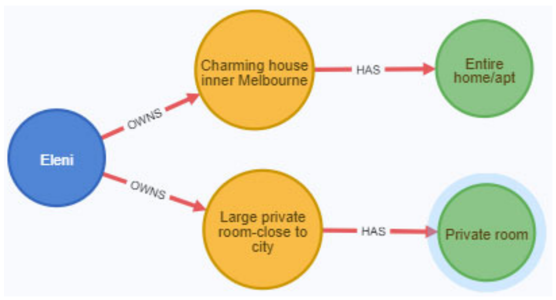

Charming house inner Melbourne
HAS
Entire home/apt
OWNS
Eleni
OWNS
Large private room-close to city
HAS
Private room

Figure 4.4: Sub graph displaying relationships in graph databases

QUERY 5:

```
MATCH (list:LISTING), (amen:AMENITY)
WHERE list.listing_id = amen.listing_id
CREATE (list)-[:HAS]->(amen)
```

Figure 4.5 shows the graph schema for the Airbnb LPG which is created in Neo4j following the process discussed in this section. Queries in Cypher are executed for loading the data from `listings.csv`, `reviews.csv` and `calendar.csv`. The schema of the database can be checked by using the call db.schema() command in Cypher.

Other researchers have discussed schema driven frameworks for data model transformation and data loading. For instance, authors in [152, 153] have proposed a conversion technique from the relational model to NoSQL data stores such as ManGoDB and Apache Cassandra. These studies, rely on ERDs for creating mapping rules that facilitate the conversion. Authors in [154] have proposed relational data model conversion to HBase store by transforming ERDs. Authors in [141] propose the use of UML diagrams as schema for graph databases. In [155] authors demonstrate the importance of schema in NoSQL databases. A model (schema) driven approach for

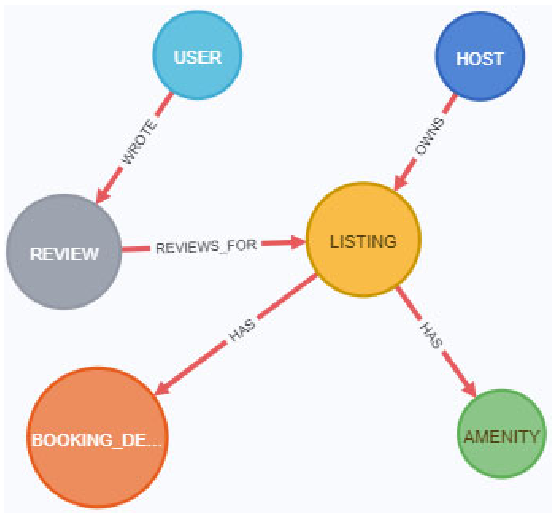


Figure 4.5: Graph schema for Airbnb dataset in Neo4j

efficient database design and data modeling process is presented in [156]. The importance of schemas in graph databases is also discussed in [78, 157, 79, 158] where authors suggest using graph schemas for generating graph databases and formation of queries. In this research, we build upon these existing techniques, and our work highlights the fact that in labeled property graph databases schema are of equally high importance. A formal description of the schema can help with analysis or verification using automated tools, such as model checkers. Formalised schema assist not only in efficient data loading, but they also provide a systematic approach for graph data analytics, as discussed in the next section.

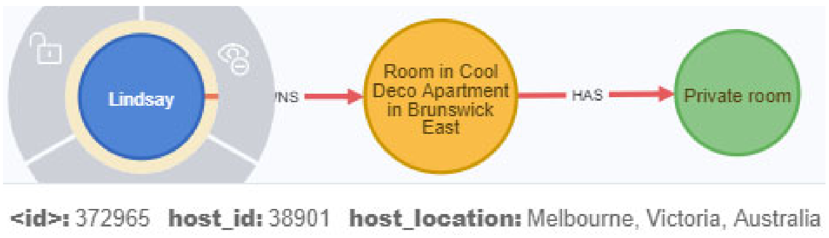
4.4 Schema-Driven Analytics for LPGs

We propose a schema-driven method for carrying out analytics over graph databases. Often, specific users only require analytics on narrow aspects of a graph database, and may not possess the necessary skills to identify how best to construct such analytics. Providing a schema as an abstraction of how data is presented allows users to analyse data in graph databases more easily.

The first step to perform analytics over any database is data extraction which is performed by using a query language. Query languages for the databases are declarative and have been motivated by the relational model proposed by Edgar F. Codd in the early 1970s [41]. The major advantage of using a declarative query language is that computational logic can be expressed without describing the control flow. This means that users specify *what* to extract from the database without describing *how* data is extracted. Cypher is a declarative query language supported by Neo4j. In Cypher, users specify a sub graph of interest and, then the query engine runs a pattern matching algorithm to find matching data in the database. For example, the following query helps in finding the name and location of *hosts* who provide private rooms as *amenities* in their *listings*.

QUERY 6:
```
MATCH (host:HOST)-[:OWNS]->(listings:LISTING),
 (listings:LISTING)-[:HAS]->(amenity:AMENITY) WHERE
 amenity.room_type = "Private room"
RETURN host.host_name, host.host_location
```

The **MATCH** clause of the query is used to express sub graph of interest and topological information from the graph schema is used to construct the sub graph. The **WHERE** clause is used to specify any user-defined restrictions such as identifying *listings* that provide private room as an *amenity*. The **RETURN** clause displays information of hosts to the user. The sub graph returned by Query 6 is shown in Figure 4.6.



<id>: 372965 host_id: 38901 host_location: Melbourne, Victoria, Australia

Figure 4.6: Sub graph returned by Query 6

Traditionally, users wanting to analyse data present in a graph database are expected to possess sufficient skills in using an associated query language, such as Cypher. However, this requirement makes graph databases inaccessible to a large proportion of potential users who are not experts. Moreover, unlike SQL for relational databases, graph databases do not have a single standard query language. Consequently, query languages are highly dependent on the graph database technology being used. Even experienced database users and developers may find it difficult to perform analytics when faced with a new graph database technology.

The proposed schema-driven analytics approach for LPGs contains the following steps. Firstly, the schema is presented to an expert user for analysis and identification of data to be analysed. Next, the expert user employs a boilerplate-based approach to create template queries for the analytics required. Finally, these queries are populated and executed every time a (potentially non-expert) user wants to perform these analytics.

We illustrate the proposed schema-driven analytics approach by using a case study of using graph databases in industrial cyber-physical systems [16]. This research proposes a tool IASelect that uses the proposed boilerplate based approach for finding information related to industrial agent practices based on user preferences. The dataset for IASelect comes from the IEEE standardization project P2660.1 [159, 160] which was provided initially in the form of a two-dimensional adjacency matrix stored as a Microsoft Excel worksheet. The original dataset was loaded into a graph database using the proposed loading process described in Section 4.3. The graph schema of the resulting LPG stored in Neo4j is shown in Figure 4.7.

The graph schema shown in Figure 4.7 describes several node types, such as `OnDevice` and `Hybrid` which are two primary industrial agent practice types. Each node of either type then has additional characteristics stored as node types `Maintenance, Function, Domain` and `Performance Efficiency`. The mapping between practices and a characteristic is represented as edges in the graph schema.

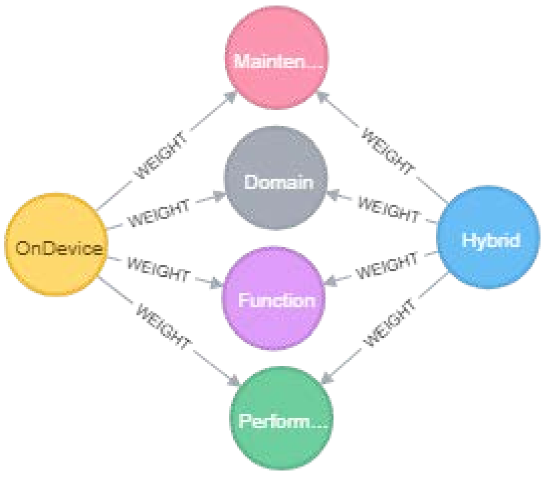
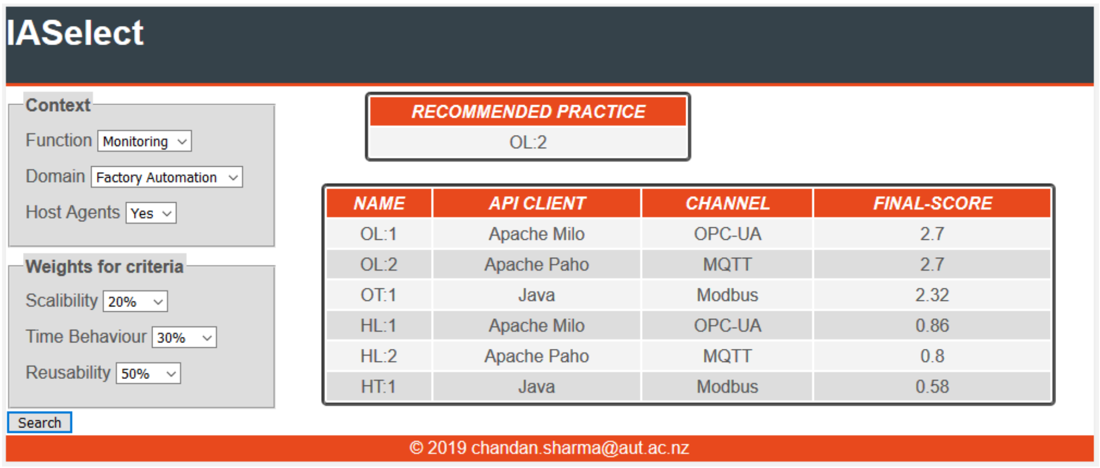


Figure 4.7: Graph Schema for storing information about practices

The IASelect graph database is expected to be used by industrial automation experts who do not have a working knowledge of Cypher. Hence, using the schema-driven analytics process described earlier in this section, we develop template queries that can be used by users to search for industrial practices based on their characteristics. A sample boilerplate template query written in Cypher for IASelect is presented as query 7.


IASelect

Context
Function: Monitoring
Domain: Factory Automation
Host Agents: Yes

Weights for criteria
Scalibility: 20%
Time Behaviour: 30%
Reusability: 50%

Search

RECOMMENDED PRACTICE
OL:2

NAME	API CLIENT	CHANNEL	FINAL-SCORE
OL:1	Apache Milo	OPC-UA	2.7
OL:2	Apache Paho	MQTT	2.7
OT:1	Java	Modbus	2.32
HL:1	Apache Milo	OPC-UA	0.86
HL:2	Apache Paho	MQTT	0.8
HT:1	Java	Modbus	0.58

© 2019 chandan.sharma@aut.ac.nz

Figure 4.8: Web form-based user interface of IASelect

The **MATCH** clause in the boilerplate query 7 represents a sub graph to search within the P2660.1 dataset. The **MATCH** clause describes that the search involves identifying all

QUERY 7:

```
MATCH
(technique)-[functionType:WEIGHT]->(f:Function),
(technique)-[domainType:WEIGHT]->(d:Domain),
(technique)-[hoAgent:WEIGHT]->(p:PerformanceEfficiency),
(technique)-[timeB:WEIGHT]->(p1:PerformanceEfficiency),
(technique)-[reuse:WEIGHT]->(m:Maintenance),
(technique)-[scale:WEIGHT]->(p2:PerformanceEfficiency)
WHERE f.name = [FUNCTION]
AND d.name = [DOMAIN]
AND p.name = [HOST AGENTS]
AND p1.name = Time Behaviour
AND m.name = Reusability
AND p2.name = Scalability
RETURN technique.name AS NAME, technique.apiClient AS API CLIENT,
 technique.channel AS CHANNEL,
CASE
WHEN [HOST AGENTS] = true
THEN
CASE
WHEN hoAgent.value = false
THEN ([TIME BEHAVIOUR]) * timeB.value + ([REUSABILITY]) *
 reuse.value + ([SCALABILITY]) * scale.value
ELSE 0
END
ELSE round((([TIME BEHAVIOUR]) * timeB.value + ([REUSABILITY]) *
 reuse.value + ([SCALABILITY]) * scale.value)*1000)/1000
END AS FINAL-SCORE
```

possible node variables `technique` of types `OnDevice` or `Hybrid` must have outgoing edges to nodes `f, d, p, p1, p2` and `m` of types `Function, Domain, Performance Efficiency` and `Maintenance`, respectively. Furthermore, the **`WHERE`** clause constrains the search over specific values for some properties of `f, d, p, p1, p2` and `m`. It also constrains the edges to be considered, by restricting the search to specific threshold of edge values, which are stored in the graph database as real numbers. In Query 7, **`FUNCTION, DOMAIN, HOST AGENTS, TIME BEHAVIOUR, REUSABILITY`** and **`SCALABILITY`** represent parameters that can be given concrete values by a non-expert user.

To allow non-experts to use IASelect, we develop a user interface which acts as a front-end for populating the boilerplate queries. The user interface acts as an additional layer between a user and the query engine. The user submits a data extraction request by populating the parameters of the boilerplate query. Upon submission, the interface runs the populated template query in the background using a query engine. Figure 4.8 shows the front-end form developed for Query 7 and the subsequent database extract report generated in IASelect that displays user criteria-specific, industrial agent practice recommendations.

Graph databases have become increasingly popular for carrying out data analytics over big data. A tool to visualize requirements based on Neo4j is presented in [161]. Graph databases have found use in similar tools in other domains like chemistry [6] and biology [95]. All these approaches use graph theory concepts to handle and inquiry data from databases. On the other hand, IASelect is focused on increasing the accessibility of analytics over graph databases through boilerplate queries and front-end generation. Such an approach is better suited for big data applications because graph databases scale well. A LPG is a multigraph where two nodes can be connected via multiple edges. Each such edge can contain separate information about the relationship between the two nodes. Adding more information, therefore, does not require a refactoring or restructuring of the database. This is useful in the case of IASelect, because the database can be enriched by merely adding new edges and edge types within the schema if more data to classify industrial agent practices is added to the database. Structural changes can be easily accommodated in LPGs because they are schema-optional [38], but that can increase the chances of data corruption. This risk can be mitigated by using the proposed approach of enforcing topological constraints through the schema and creating user-specific front-ends.

4.5 Conclusions and Future Directions

This paper focuses on the utility of having a well-defined schema for graph databases and more precisely labeled property graphs (LPGs). We propose a formally-described schema that can help create confidence in a LPG's structure through user scrutiny and automatic analysis. We show how this schema can make core processes like populating graph databases with new datasets and helping non-experts perform analytics, more systematic.

The proposed schema-first approach can assist in preventing data corruption. Any update operations can be automatically checked against the schema to generate warnings if they alter the structure of the database. Such constraints can be instrumental in ensuring data integrity when graph databases are used simultaneously by a large number of users in a distributed manner as any Create-Read-Update-Delete (CRUD) operations over the graph database have to be performed following the schema. The schema-driven approach does not sacrifice the flexibility provided by LPGs in the form of unrestricted schema-less updates. Firstly, the schema does not constrain the amount and kind of data being stored in the nodes and edges of a LPG as properties. Also, in its weakest form, the formalised schema can be used to generate warnings and can be automatically updated when the database's structure changes during update operations. A record of changes in the schema, which can be formally characterized through our formalism, can be used to monitor changes to the database over time.

Future directions to this research include extending the proposed formalism to systematically compare different graph database frameworks and query languages, and exploring the possibility of generating an intermediate format for queries written in one language to be automatically transformed into another language.

Chapter 5

Introduction to Manuscript 2

The industry-wide adoption of graph databases has been hindered due to the fragmentation in syntax and semantics of available graph query languages. As a result, several projects have been proposed by industry and academia to develop a standard query language by integrating features from existing practical graph query languages. A significant factor that can impact query language integration is the lack of common theoretical language formalisms. We propose common formalisms by extending conjunctive queries and union of conjunctive queries with Tarski's relation algebra (CQT/UCQT). The common formalisms are them used to propose an integrated framework that uses common graph query patterns to compare the expressive power of (CQT/UCQT) with two practical graph query languages - Cypher and PGQL. In the integrated framework the query languages are analysed on the core features of graph pattern matching and graph navigation, revealing the common and exclusive characteristics for these languages. Overall, our study serves as a formal basis for comparing existing graph query languages and assists the move towards query language integration and interoperability between available graph database technologies.

Chapter 6

Practical and Comprehensive Formalisms for Modeling Contemporary Graph Query Languages (Manuscript 2)

6.1 Introduction

Graph databases are efficient in storing and analysing highly interconnected data [27, 1, 32, 23, 24, 26]. They are gaining popularity due to application in transportation networks, bioinformatics, astronomy and chemistry [32, 31, 6, 29, 5] where analysing interactions between data is vital. Databases use query languages for extracting and analysing the stored data [162]. Graph databases lack a standard query language as SQL is for relational databases. As a result, there exist many graph query languages proposed by industry such as Cypher (Neo4j) [51], PGQL (Oracle) [52], G-Core (LDBC) [1], SPARQL (W3C) [163], Gremlin (Apache) [111] and query languages from academia such as navigational XPath [129] and GXPath [34]. This fragmentation has hindered the industry-wide adoption of graph databases. Therefore, projects such as ISO/IEC 39075[1], openCypher [164] and Linked Data Benchmark Council (LDBC) [4] have been proposed to develop a new standardised graph query language by integrating features from existing query languages such as Cypher, PGQL and G-Core. To integrate query languages, they have to be compared to identify common and exclusive characteristics in each language.

The large number of graph query languages proposed by industry and academia vary in syntax and semantics [32]. SPARQL, Cypher, PGQL and G-Core are declarative query languages while Gremlin is a functional query language. Syntax of SPARQL and PGQL is similar to SQL while Cypher and Gremlin have varying syntax to each other and SQL. The syntax of G-Core is based on SQL, SNQL, SPARQL, Cypher and PGQL. Furthermore, these query languages' semantics also vary [1, 32] and are dependent on the underlying implementation of each query language. Therefore, identifying a common theoretical language formalism for objectively comparing graph query languages is vital. By doing so, query languages can be syntactically compared and, common semantics can also be identified and enforced to show query equivalence. Moreover, this also helps integrate features from different graph query languages and aids in query language interoperability.

Practical graph query languages do not share common underlying formalisms. For instance, query languages Cypher, PGQL and SPARQL use formalisms based on regular path queries (RPQ) and two-way regular path queries (2RPQ) such as conjunctive regular path queries (CRPQ), conjunctive two-way regular path queries (C2RPQ), union of conjunctive regular path queries (UCRPQ) and union of conjunctive two-way regular path queries (UC2RPQ). SPARQL fully supports these formalisms while Cypher and PGQL only provide partial support. More expressive formalisms based on nested regular expressions such as conjunctive nested regular expressions (CNRE) [34] are

[1]`https://www.iso.org/standard/76120.html`

partially supported by SPARQL and Cypher while PGQL fully supports the use of such formalisms. Formalisms such as extended conjunctive regular path queries (ECRPQ) and their extensions based on regular and rational relations [73, 115] are partially supported by SPARQL and Cypher while PGQL does not support the use of such formalisms. Query languages proposed by academia such as navigational XPath and GXPath are based on Tarski's relation algebra [34, 74]. Any practical graph query language does not support formalisms such as context-free path queries (CFP) and their extensions [128]. To the best of our knowledge an explicit mapping between theoretical capabilities demonstrated by existing formalisms and practical query languages is missing. As such theory and practice are being developed independently.

Existing formalisms such as CRPQ, C2RPQ, UCRPQ, UC2RPQ, and CNRE are not expressive enough and are not shared across all graph query languages. To compare practical graph query languages, a theoretical language formalism is required that encompasses the expressive power provided by the majority of graph query language formalisms. Furthermore, the practical implications of an increase in formalisms' expressive power also have to be considered. For instance, formalisms such as ECRPQ provide the ability to output and compare paths. This feature can be problematic in cases when cycles exist in graph databases. In such scenarios, a query can return infinite paths if appropriate evaluation semantics are not considered [32]. Similarly, authors in [72] rule out the practical use of formalism such as Walk logic [165] due to its high evaluation complexity. Hence we focused on three main research objectives:

RO1 Extend existing theoretical language formalisms to propose common formalisms that prevails the expressive power of existing formalisms and can also be used to model practical graph query languages.

RO2 Develop an integrated framework by using the extended formalism proposed by

RO1 to support the design of benchmark queries used to evaluate the expressiveness of practical graph query languages.

RO3 By using the integrated framework proposed by RO2 highlight the common and exclusive characteristics of practical graph query languages Cypher and PGQL, so that findings from this study can be used to model future graph query languages.

This research was carried out using mixed methods research methodology, where we first conducted a literature review to identify gaps related to RO1-RO3 presented in Section 6.2. For RO1, we extend formalisms of conjunctive queries and union of conjunctive queries with Tarski's relation algebra. The evaluation complexity of these formalisms has been studied previously. Moreover, we are only using these formalisms to compare graph query languages syntactically. Hence theoretical results of evaluation complexity related to the extended formalisms are considered as future work. Concerning RO2, we consider common graph query patterns such as *chain* and *cycles* [77, 54, 166] to construct a generalized and comprehensive collection of benchmark queries based on the extended formalisms. For RO3 we use, the integrated framework proposed in RO2 to experimentally compare the expressiveness of graph query languages Cypher and PGQL.

The scope of this study is limited to measuring the expressiveness of declarative query languages for graph databases. We do not consider performance, efficiency, complexity and time to execute the queries. All these factors are based on how well search algorithms are implemented in each proprietary graph database platform, and hence cannot be compared formally. The critical contributions of this work are:

1. We conduct a literature review to identify core features of graph query languages, existing formalisms used for querying graph databases, and common graph query patterns used later to formulate benchmark queries. The literature review is presented in Section 6.2. We consider real-world Airbnb case study to develop

a graph schema and graph database in Section 6.3. The case study is used to illustrate various definitions and benchmark queries presented in this study.

2. We discuss both the syntax and semantics of conjunctive queries and union of conjunctive queries extended with Tarski's algebra in Sections 6.4 and 6.5. The extended formalisms are then used to compare graph query languages on the core features of graph pattern matching, and graph navigation identified in Section 6.2.

3. Extending the formalisms lead to a framework inclusive enough for measuring practical graph query languages' expressiveness. Furthermore, being focused on practical graph query languages, the framework assists in industry and academia's ongoing efforts to create a standard graph query language. The integrated framework is presented in Section 6.6.

4. We use the proposed framework to conduct a comprehensive comparison between two graph query languages Cypher and PGQL which reveals the common and exclusive queries expressed in the two languages. The benchmark queries and query language comparison is presented in Sections 6.7 and 6.8.

6.2 Background and Related Works

We have considered RO1-RO3 as primary factors for conducting the literature review. In this research, we propose common formalisms that can be used to compare the expressiveness of different practical graph query languages, where expressiveness refers to the ability of a query language to express different graph query patterns syntactically. Therefore, we discuss existing formalisms in our review. Due to the diversity in graph query languages, the feature set provided by each language also vary. Hence, we also identify core features that are common in graph query languages. To formulate queries for measuring expressiveness, we identify common graph query patterns that have been proposed in the literature. We focused on three key research questions for conducting the literature review:

RQ1 What are the core features of practical graph query languages proposed by industry and academia?

RQ2 What are the existing formalisms used in practical graph query languages?

RQ3 What are the existing, common graph query patterns studied in industry and academia that can be used to formulate benchmark queries for measuring the expressiveness of practical graph query languages?

The search strings were formulated by extracting keywords from RQ1-RQ3. Then databases such as Scopus, IEEE Explore, ACM Digital library and ScienceDirect were searched for research papers using the search strings. We also utilised journal recommendation tools by Elsevier [2] and Springer [3] to find specific journals for extracting papers based on the search strings. The inclusion and exclusion criteria of a research paper were based on the publication's venue reputation. This was verified by using web portals such as scimagojr [4] and core [5]. Furthermore, snowballing, expert suggestions and the relevant paper published in highly reputed database venues such as ACM TODS [6], IEEE TKDE [7], ACM SIGMOD Record [8],VLDB Journal [9], EDBT [10], ICDT [11] and PODS [12] were other inclusion criteria for selecting a research paper. White-papers from industry were retrieved from official web sources such as W3C [163], Oracle [52] and Neo4j [51]. Findings from the review are summarised in the following sections:

[2]https://journalfinder.elsevier.com/
[3]https://journalsuggester.springer.com/
[4]https://www.scimagojr.com/
[5]http://portal.core.edu.au/conf-ranks/
[6]https://dl.acm.org/journal/tods
[7]https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=69
[8]https://dl.acm.org/newsletter/sigmod
[9]https://www.springer.com/journal/778
[10]https://dblp.uni-trier.de/db/conf/edbt/index.html
[11]https://dblp.uni-trier.de/db/conf/icdt/icdt2019.html
[12]https://dblp.uni-trier.de/db/conf/pods/index.html

6.2.1 Core features of graph query languages

Graph query languages have five core features of graph pattern matching, graph navigation, shortest path search, graph construction, and graph clustering [1, 26].

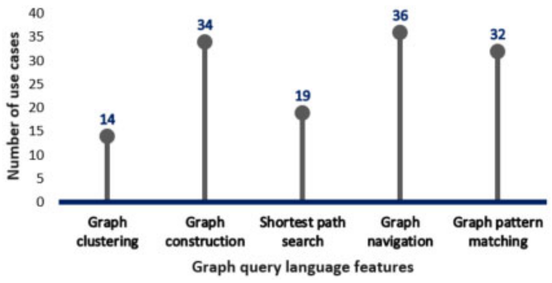


Figure 6.1: Core features of graph query languages (adapted from [1])

As shown in Figure 6.1, out of 40 use cases identified by Linked Data Benchmark Council[13], graph navigation, graph construction and graph pattern matching are the most desired core features in any graph query language. Several other works such as [32, 62, 76, 28, 24] consider graph pattern matching and graph navigation as core features. These two features are primarily used for data extraction. The main difference between the two is that *graph pattern matching* queries search for concrete sub-graph structures in a graph database, while *graph navigation* queries focus on navigating through the topology of a graph database [34, 167]. Queries such as finding nodes connected by paths labeled with certain edge label are graph navigation queries.

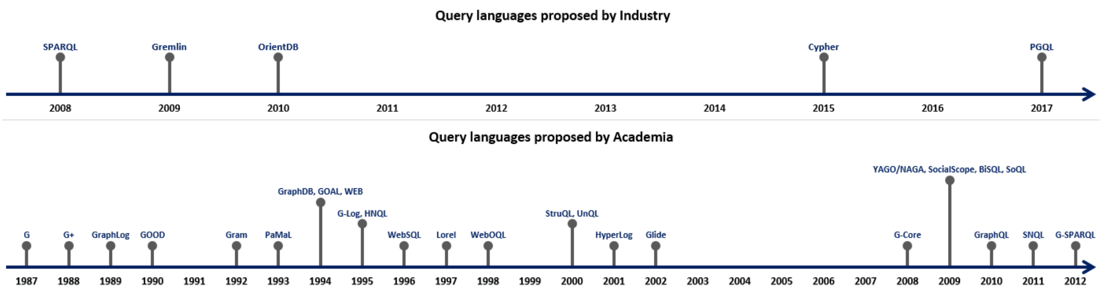


Figure 6.2: History of graph query languages proposed by industry and academia

As shown in Figure 6.2, we surveyed a total of 33 graph query languages proposed

[13]https://github.com/ldbc/tuc_presentations

by industry and academia, and all languages support the core features of graph pattern matching. For graph navigation, 21 query languages support the use of regular expressions defined over the edge labels of a graph database. Query languages such as Navigational XPath [129] and GXPath [34] use Tarski's algebra for graph navigation. In query languages G-Log [96], SNQL [109], GOOD [93] and WebOQL [168] edges are treated as binary relations, and transitive closure property of relations is used to answer graph navigation queries. Query languages such as SoQL [107], SocialScope [105], GOAL [92] and Hyperlog [102] can only express fixed-length graph navigation queries while HNQL [97] and PaMaL [91] use iterative loops, to express graph navigation queries. Query languages proposed by industry such as Cypher, SPARQL and PGQL, irrespective of differences in their syntaxes, use regular expressions for graph navigation.

6.2.2 Existing formalisms for graph query languages

Identifying a common formalism for comparing query languages is vital because formalisms assist in understanding the meaning of query languages [169]. By using a common formalism, queries expressed in different languages can be translated into mathematical formulas. Moreover, these mathematical representations of queries can be used to check the syntactic equivalence of queries, thereby assuring reproducibility of the results [170].

A core formalism for expressing graph pattern matching queries is that of conjunctive queries (CQ) [40, 32, 171, 115, 172, 173, 14]. Conjunctive queries are an essential class of database queries that can be used to show the equivalence between two or more queries [58, 61]. They are further extended to include union, which makes them equivalent to the selection, projection, join and union (SPJU) fragment of Codd's relational algebra [32, 60]. This formalism is called the union of conjunctive queries (UCQ).

For graph navigation queries, a basic formalism used is that of regular path queries (RPQ) [32, 34]. In RPQs a topological order is described using regular expressions defined over the edge labels of a graph database. Essentially, RPQs are used to describe and then search for paths in a graph database. RPQs have limited expressiveness as they can only search for paths in a single direction [62, 37]. Therefore, the formalism of two-way regular path queries (2RPQ) has been proposed [73, 78, 34] that allows the use of *inverse* operation over an edge label and is used to specify opposite direction edges.

Table 6.1: Mapping between existing formalisms and practical graph query languages (SPARQL,Cypher and PGQL)

	Formalisms	**Types of queries**	**Theory/Practice**	**Query Languages**
1	CQ	GPM	T + P	SPARQL,Cypher, PGQL
2	UCQ	GPM	T + P	SPARQL,Cypher, PGQL ⋄
3	RPQ	GN	T + P	SPARQL, Cypher ⋄, PGQL
4	2RPQ	GN	T + P	SPARQL,Cypher ⋄, PGQL
5	CRPQ	GPM + GN	T + P	SPARQL,Cypher ⋄,PGQL
6	C2RPQ	GPM + GN	T + P	SPARQL,Cypher ⋄,PGQL
7	UCRPQ	GPM + GN	T + P	SPARQL,Cypher ⋄,PGQL⋄
8	UC2RPQ	GPM + GN	T + P	SPARQL,Cypher ⋄,PGQL⋄
9	NRE	GN	T + P	SPARQL ⋄,Cypher ⋄ ,PGQL
10	CNRE	GPM + GN	T + P	SPARQL ⋄,Cypher ⋄,PGQL
11	UCN2RPQ	GPM + GN	T + P	SPARQL ⋄,Cypher ⋄,PGQL⋄
12	ECRPQ(reg) & ECRPQ(rat)	GPM + GN	T + P	SPARQL ⋄,Cypher ⋄
13	CFP	GPM + GN	T	N/A
14	TA	GN	T + P	SPARQL ⋄,Cypher ⋄,PGQL⋄

GN = Graph Navigation
GPM = Graph Pattern Matching
T = Theoretical formalism
P = Formalism used in Practical graph query language
⋄ = Limited use of the formalism

In graph databases, one often need to search for the existence of paths that contain topological properties such as branches [34] on the intermediate nodes. Formalisms such as RPQs and 2RPQs are not expressive enough as they cannot be used to search arbitrary-length paths that include branches on the intermediate nodes. Hence, the formalism of nested regular expressions (NRE) has been proposed [174, 73, 37]. NREs extend regular path queries (RPQs) with inverse and nesting operator. The nesting operator helps define branches; furthermore, Kleene star can be applied over nested expression enabling the search of arbitrary-length paths with branches.

Subsequent works [124, 115, 125, 64, 62, 33, 66] have considered adding the expressive power of graph pattern matching formalisms such as conjunctive queries and union of conjunctive queries to graph navigation formalisms. These extensions yield a more expressive class of formalisms such as conjunctive regular path queries (CRPQ), conjunctive two-way regular path queries (C2RPQ), union of conjunctive regular path queries (UCRPQ) and union of conjunctive two-way regular path queries (UC2RPQ). Formalisms of C2RPQs and NREs are incomparable in terms of expressiveness [62], since C2RPQs do not allow the application of Kleene star over branching operator while NREs cannot express cyclic graph structures. Therefore, authors in [126, 36, 34] propose conjunctive nested regular expressions (CNRE) and union of conjunctive nested two-way regular path queries (UCN2RPQ) [127].

CRPQs have been extended to formalisms such as extended conjunctive regular path queries with regular relations (ECRPQ(reg)) and rational relations (ECRPQ(rat)) [73, 115]. ECRPQ based formalisms provide the ability for path comparisons and path based associations to search for sub-words and sub-sequences in paths [33]. Formalisms based on context-free paths (CFP) and their extensions have also been studied in theoretical literature such as [128, 175] and extensions to SPARQL query language have been proposed to support the use of context-free paths [176, 175]. However, most features in these formalisms are not yet available in all practical graph query languages. A primary reason for this is that evaluating queries expressed in context-free path queries' formalism requires significant computational resources for larger graphs [177, 178]. Therefore, such formalisms cannot be used in practical graph query languages.

Table 6.1 shows a mapping between existing formalisms and their use in practical query languages such as SPARQL,Cypher and PGQL. We can observe that Cypher and SPARQL use the formalisms of conjunctive queries(CQ) and union of conjunctive queries (UCQ) for graph pattern matching. PGQL lacks the `UNION` clause; therefore, use of UCQ based formalism is limited in the query language.

Formalism such as regular path queries (RPQ), two-way regular path queries (2RPQ), conjunctive regular path queries (CRPQ) and conjunctive two-way regular path queries (C2RPQ) are used in SPARQL and PGQL. On the other hand Cypher does not allows the use of Kleene star over the concatenation of two or more edge labels hence RPQ, 2RPQ, CRPQ and C2RPQ based formalisms are not fully supported in Cypher. Formalisms such as UCRPQ and UC2RPQ are extensions of CRPQ and C2RPQ; therefore, these formalism are not fully supported in Cypher. Due to the lack of `UNION` clause, PGQL does not fully support UCRPQ and UC2RPQ based formalisms. SPARQL, on the other hand, fully supports the use of UCRPQ`s` and UC2RPQ`s`.

Formalisms such as nested regular expressions (NRE) and conjunctive nested regular expressions (CNRE) are only fully supported in PGQL. Both SPARQL and Cypher do not allow the use of Kleene star over branched structures. In [179], authors propose the extension of SPARQL to support the application of Kleene star over branched structures. The formalism of union of conjunctive nested two-way regular path queries (UCN2RPQ) is extension of CNRE and UC2RPQ so SPARQL and Cypher do not fully support the formalism while PGQL lacks the `UNION` clause. Therefore, the formalism of (UCN2RPQ) is not fully supported in all three query languages.

Extended conjunctive regular path queries based formalism such as ECRPQ(`reg`) and ECRPQ(`rat`) are partially supported by SPARQL and Cypher. In [180, 181, 182] authors propose the extension of SPARQL with path variables and path filtering expressions while Cypher only provides the ability to output paths. PGQL does not support the functionality to compare or output paths. The formalisms of context-free paths (CFP) and their extensions are not used in all the three query languages.

To compare the expressiveness of practical graph query languages, formalisms such as the union of conjunctive nested two-way regular path queries (UCN2RPQ) [127] can be used as they are more expressive than other formalisms such as UC2RPQ and CNRE. However, the main disadvantage of using NRE based formalisms is their inability to search for arbitrary-length paths that contain cyclic or acyclic structures between start and end nodes. Searching for such paths can be vital in graph database application fields such as bioinformatics and chemistry where for instance, a user might be interested in searching for the existence of long polymer chains of arbitrary-length that are formed of repeating acyclic or cyclic structures [31, 6]. Hence, graph navigation formalism based on the relation algebra of Tarski, henceforth Tarski's algebra (TA) [75] has been studied in [183, 22, 35, 55, 129]. Syntax and semantics of Tarski's algebra for querying graph databases are discussed in [74], and authors suggest that Tarski's algebra is more expressive than graph navigation formalisms such as RPQ, 2RPQ and NRE. As shown in Table 6.1 practical graph query languages SPARQL, Cypher and PGQL partially support the use of this formalism for graph navigation queries.

The formalism of Tarski's algebra is purely navigational; hence in this research, we propose the extension of conjunctive queries and union of conjunctive queries with Tarski's algebra. These extensions provide several advantages firstly, being more expressive than other graph query language formalisms such as UC2RPQ and UCN2RPQ, conjunctive queries with Tarski's algebra (CQT) and union of conjunctive queries with Tarski's algebra (UCQT) serve as common formalisms for comparing graph query languages such as Cypher and PGQL. Secondly, queries that search for arbitrary-length paths, containing acyclic and cyclic structures between start and end nodes can also be expressed in CQT and UCQT. Finally, the evaluation complexity of conjunctive queries and Tarski's algebra have been previously studied, and we are only using these extensions to compare the expressiveness of graph query languages syntactically. Moreover, the evaluation complexity is dependent upon the underlying implementation of each practical graph query language.

6.2.3 Common graph query patterns

In order to investigate common graph query patterns, several studies have been conducted. Authors in [184, 166] conduct a detailed study by analysing query logs found in real-world repositories such as OpenLink [185], British museum query logs from LSQ [186] and Wikidata [187]. Findings for these logs suggest that graph query patterns are broadly classified as chain and cycle patterns. Chain patterns are extended to form patterns such as tree, star, star-chain, chainset, and forest [74, 188]. Furthermore, authors in [35] suggest that chains and trees are basic structures for analysing the expressiveness of query languages, and results can be generalised to more complex graph patterns. Authors in [189, 190] suggest that chain and star are basic graph query patterns in SPARQL. Chain patterns can be used to form more structurally complex patterns, such as directed acyclic graphs (petals) and cycles [166]. The analysis presented in [77, 184, 166] suggests that cyclic patterns are not that common in real-world query logs as they tend to increase the evaluation complexity of graph queries. Other cyclic patterns considered in the literature are flowers and bouquets [166]. Majority of tools and benchmarks designed for analysing graph query languages such as SPARQL consider chains and cycles as common graph query patterns. Furthermore, benchmark queries designed by using such tools are based on the formalism of conjunctive queries and conjunctive regular path queries [191, 64, 192, 193, 54, 194]

6.3 Airbnb case study

We use a dataset from Airbnb [195] as running example to illustrate various definitions presented in subsequent sections. Airbnb's datasets are available for download under a Creative Commons license in comma-separated values (CSV) file format [87]. This dataset consists of three CSV files that contain information related to property listings, reviews and calendar data. This data set is highly interconnected and requires rigorous data analytics, making it a prime candidate for graph database implementations. These features also make this database ideal for comparing graph database queries.

This study is adapted from the work presented previously in [15] where we discuss the formalism for labeled property graph schema and database developed for the Airbnb dataset. We briefly discuss these concepts and present some examples.

6.3.1 Labeled Property Graph Schema

A labeled property graph (LPG) schema captures the structural and properties based restrictions on a LPG database [16], where nodes represent entities and edges represent relationships between the entities that belong to a domain such as Airbnb dataset. Let $L_{\mathcal{N}}$ be a set of node labels, $L_{\mathcal{E}}$ be a set of edge labels and P_s be a set of properties of a LPG schema such that each $p_s \in P_s$ contains a key of a specific data type.

Definition 8 (`Labeled Property Graph Schema`) *A Labeled Property Graph schema $\mathcal{G}_s = (\mathcal{N}_s, \mathcal{E}_s, P_s, \eta_s, \xi_s, \Delta_s)$ is a tuple where,*

- *$\mathcal{N}_s$ is a finite set of nodes and $\mathcal{E}_s$ is a finite set of edges of the graph schema.*
- *$\eta_s : \mathcal{N}_s \to L_{\mathcal{N}}$ is a node labeling function which maps all nodes to labels in the set of node labels $L_{\mathcal{N}}$.*
- *$\xi_s : \mathcal{E}_s \to L_{\mathcal{E}}$ is an edge labeling function which maps all edges to labels in the set of edge labels $L_{\mathcal{E}}$.*
- *$\Delta_s : (\mathcal{N}_s \cup \mathcal{E}_s) \to \wp(P_s)^+$ is a property labeling function which maps all nodes and/or edges to the non empty subset of the property set P_s.*

Example 6 *Figure 6.3 shows a LPG schema for the Airbnb dataset. This schema contains six labeled nodes, including `REVIEW`, `USER`, `HOST` and `LISTING`, which are the permitted node labels in any database conforming to this schema.*

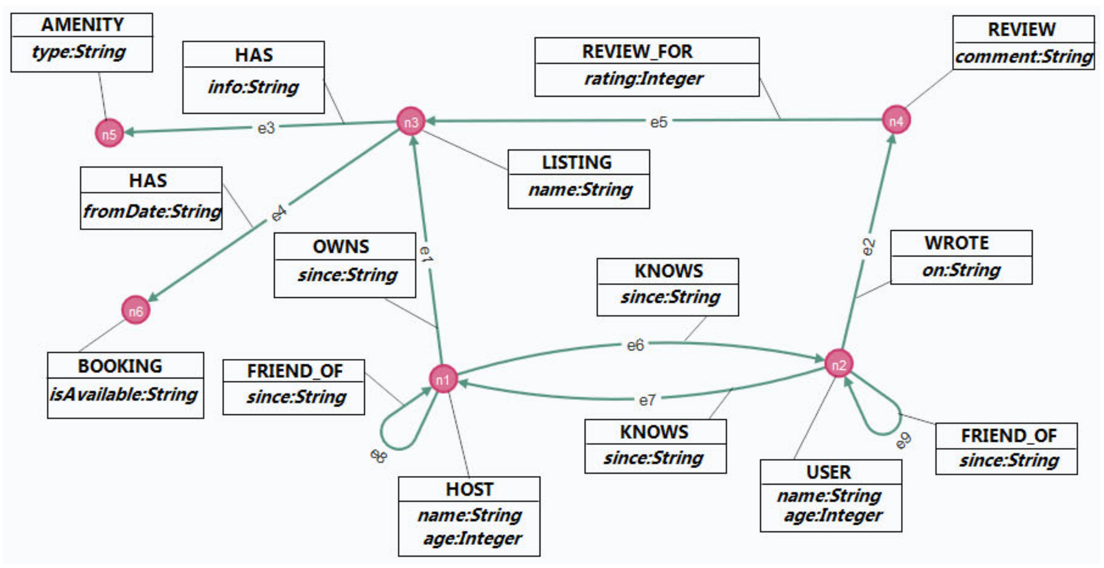


Figure 6.3: A labeled property graph schema for the Airbnb dataset

The edges of a graph schema describe valid *edges* that a graph database following the schema can contain. Edges in a graph schema are directed and are restricted to specific edge labels in the dataset.

Example 7 *Figure 6.3 shows the permitted edge labels in any database conforming to this schema. For example, the edge e_6 is labeled with* `KNOWS`*, indicating that in the graph database, nodes of label* `HOST` *can have outgoing edges to nodes of label* `USER` *and that all such edges must be labeled by* `KOWNS`*.*

Nodes and edges of a graph schema also contain information about the allowed data types stored as properties in any node or/and the edge of a graph database.

Example 8 *Figure 6.3 shows the permitted data types associated with the properties of a node and/or edge of a graph database. For example, node n_1 has two associated properties name and age that have String and Integer as associated data types.*

6.3.2 Labeled Property Graph Database

A labeled property graph (LPG) database uses a graph data structure for storing and managing data, allowing the modelling of real-world entities as nodes and edges [196]. Nodes are used to store data and relationships or interactions between nodes are stored as edges [28, 43]. Let P_d be a set of properties of a graph database such that each $p_d \in P_d$ is a key-value pair where each value has a data type.

Definition 9 (`Labeled Property Graph Database`) *A Labeled Property Graph database $\mathcal{G}_d = (\mathcal{N}_d, \mathcal{E}_d, P_d, \eta_d, \xi_d, \Delta_d)$ is a tuple where,*

- *$\mathcal{N}_d$ is a finite set of nodes and $\mathcal{E}_d$ is a finite set of edges of the graph database.*
- *$\eta_d : \mathcal{N}_d \to L_{\mathcal{N}}$ is a node labeling function which maps all nodes to labels in the set of node labels $L_{\mathcal{N}}$. For each node $n \in \mathcal{N}_d$, there must exist a corresponding node $n' \in \mathcal{N}_s$ such that $\eta_d(n) = \eta_s(n')$.*
- *$\xi_d : \mathcal{E}_d \to L_{\mathcal{E}}$ is an edge labeling function which maps all edges to labels in the set of edge labels $L_{\mathcal{E}}$. For each edge $(n_1, e_1, n_2) \in \mathcal{G}_d$ there must exist a corresponding edge $(n'_1, e'_1, n'_2) \in \mathcal{G}_s$ such that $\eta_d(n_1) = \eta_s(n'_1)$, $\eta_d(n_2) = \eta_s(n'_2)$ and $\xi_d(e_1) = \xi_s(e'_1)$.*
- *$\Delta_d : (\mathcal{N}_d \cup \mathcal{E}_d) \to \wp(P_d)^+$ is a property labeling function which maps all nodes and/or edges to the non empty subset of the property set P_d. For any $n_i \in \mathcal{N}_d$ (or $e_i \in \mathcal{E}_d$), there exists $n'_i \in \mathcal{N}_s$ (or $e'_i \in \mathcal{E}_s$) such that $\Delta_d(n_i) = \Delta_s(n'_i)$ (or $\Delta_d(e_i) = \Delta_s(e'_i)$). The data type of value stored in node (or edge) of graph database is same as the data type of node (or edge) in the graph schema.*

A labeled property graph database created for the Airbnb dataset is shown in Figure 6.4. The structure and properties of nodes and edges in the graph database follow

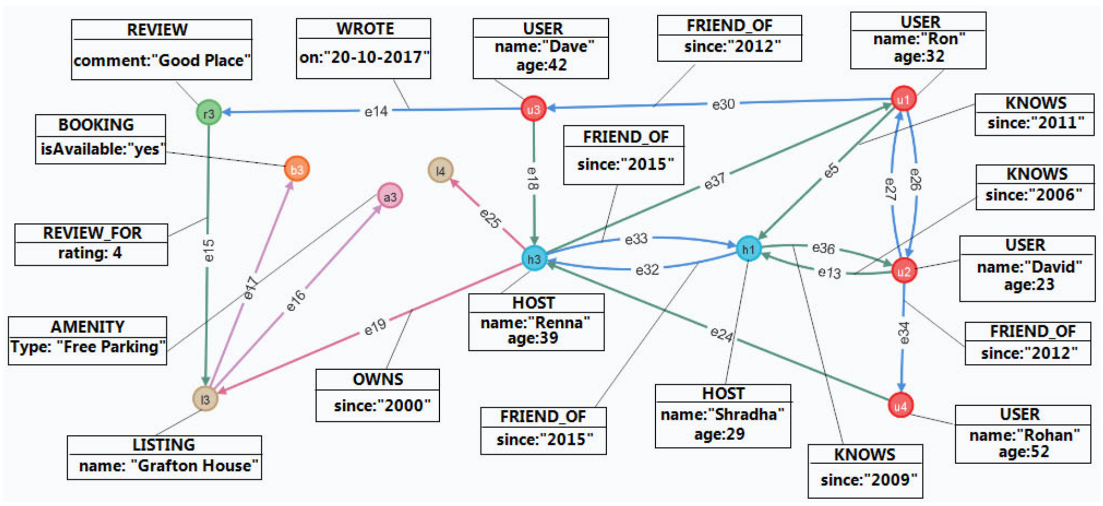


Figure 6.4: Example of a labeled property graph database for Airbnb dataset

the graph schema shown in Figure 6.3. Therefore, the graph database's nodes and edges must conform to the graph schema's valid node and edge labels.

Example 9 *In Figure 6.4, the node u_2 has an incoming edge from the node h_1 and the edge e_{36} is labeled with edge label* `KNOWS`*. The label of node u_2 is* `USER` *while the label of node h_1 is* `HOST`*. Hence the corresponding edge must be labeled as* `KNOWS`*, as that is the only allowed edge label in the graph schema where* `HOST` *is the start node and* `USER` *is the end node.*

Each node and edge of a labeled property graph database has some data properties associated with it.

Example 10 *Figure 6.4 shows that the node u_1 has the node label* `USER` *and two associated properties in a key-value format that is* `name:"Ron"` *and* `age:32`*. The values assigned to the attributes are accessed by using the shorthand notation such as u_1*`.name = "Ron"` *and u_1*`.age = 32` *we can also observe that data type of the properties is String and Integer respectively which is consistent with allowed data type in the graph schema. Similarly, edge e_{36} has a property* `since:"2009"` *associated* *with it. The value of this property can be accessed using the shorthand notation that is* $e_{36}.\texttt{since} = \texttt{"2009"}$ *where the data type of value is String.*

Labeled property graph(LPG) database considered in this study is restricted by the following features. A pair of nodes in a LPG database can have zero or more directed edges; each node and/or an edge in a LPG database can have a single label; each node and/or edge in a LPG database can have one or more properties where each property is a key-value pair. The property values are atomic entities, meaning that values such as maps and lists cannot be associated with a key in a property. Moreover, the values are of data types String and Integer. The definitions do not support multiple labels over nodes and edges. These restrictions are enforced for the sake of simplicity, and as mentioned in [42], these features are not present in all graph database systems and tend to make the definitions of graph schema and graph databases complex.

The most fundamental application of a database is the ability to extract the stored information. The mechanism that assists in data extraction is called as *querying* a database [162, 37, 8]. Querying is usually carried using a *declarative* query language [197], that allows users to describe “what" data is to be extracted or manipulated, without requiring a description of “how" the extraction or manipulation is carried out. We present some motivating examples of queries that can be used to extract data from the Airbnb graph database shown in Figure 6.4.

6.3.3 Examples of querying the Airbnb graph database

Example 11 *A user* `Ron,` *wishes to find a listing that provides* `free parking` *as an amenity. Furthermore, the listing has to be reviewed by* `Ron's` *friend, who knows a host.*

Example 11 represents a graph pattern matching query because, in this query, the exact sub-graph structure to be extracted from the graph database is known beforehand.

The sub-graph structure for answering this query is shown in Figure 6.5 and can be expressed by the formalism of conjunctive queries.

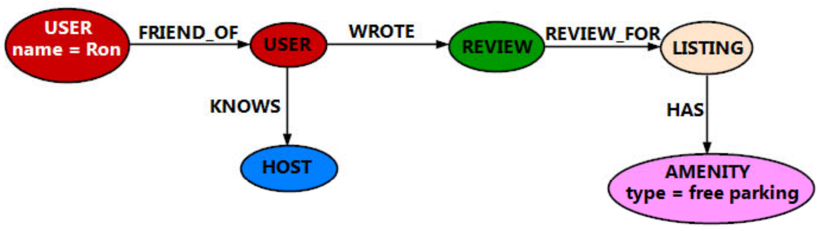


Figure 6.5: Structure of sub graph for query presented in Example 11

The graph database presented in Figure 6.4 only represents a small portion of the graph database developed for Airbnb dataset. In highly interconnected and large datasets such as Airbnb two nodes having meaningful connections might not be close to one another [23, 68, 67]. Furthermore, the total number of edges connecting two nodes can be vast and may not be known beforehand.

Example 12 *A user* `Ron` *wants to find out all people connected to him and each other by* `KNOWS` *relationship. Furthermore, people connected to* `Ron` *by such a path must own a listing and have a friend.*

Example 12 represents a graph navigation query. The query is used to search for paths of arbitrary-length connecting `Ron` to other people such that edges in the path must be labeled as `KNOWS`. Furthermore, the end node of the path must have two outgoing edges labeled as `OWNS` and `FRIEND_OF`. The sub-graph structure that has to be extracted from the graph database is shown in Figure 12, which consists of a path labeled with `KNOWS` relationship. The end node of the path has two outgoing edges labeled by `OWNS` and `FRIEND_OF` relationships. The formalism of conjunctive regular path queries can express this query.

Example 13 *A host* `Renna` *wants to find out friends and friends of friends such that all friends of* `Renna` *and friends of friends either own a listing or have written a review*

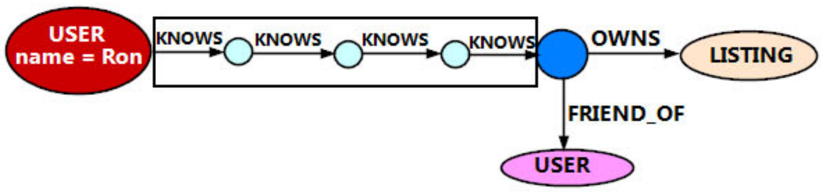


Figure 6.6: Structure of the sub graph for query presented in Example 12

for a listing.

Example 13 represents a graph navigation query, where we search for paths of arbitrary-length connecting `Renna` to other people such that edges in the path must be labeled as `FRIEND_OF`. Furthermore, the intermediate nodes in the path must have an outgoing edge labeled as `OWNS` or `WROTE`. The sub-graph structure that has to be extracted from the graph database is presented in Figure 6.7, which consists of a path with branches in the intermediate nodes. The formalism of nested regular expressions can be used to express such a query.

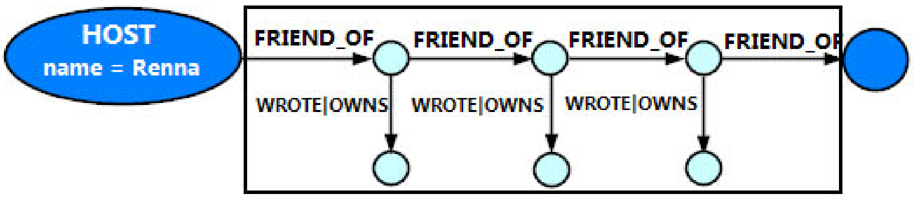


Figure 6.7: Structure of the sub graph for query presented in Example 13

Example 14 *A user* `Ron` *wants to find out all people connected to him and each other by* `KNOWS` *relationship. Furthermore, any two people directly connected by* `KNOWS` *relationship must also have a familiar friend. Additionally,* `Ron` *also wants to find out any reviews that he has written for a listing that provides* `free parking` *as an amenity.*

Example 14 also represents a graph navigation query, where we search for arbitrary-length and fixed-length paths that share a common node represented by the user `Ron`.

Edges in the arbitrary-length path must be labeled as KNOWS. Furthermore, in the path, any two nodes that are adjacent to each other must have outgoing edges labeled as FRIEND_OF to another common node. The fixed-length path is used to search for a review written by Ron for a listing that provides free parking as an amenity. The sub-graph structure that has to be extracted from the graph database is presented in Figure 6.8 which consists of two paths starting from the same node Ron. The arbitrary-length path is formed of repeating acyclic structure between the start and end nodes. Such a query cannot be expressed in existing formalisms.

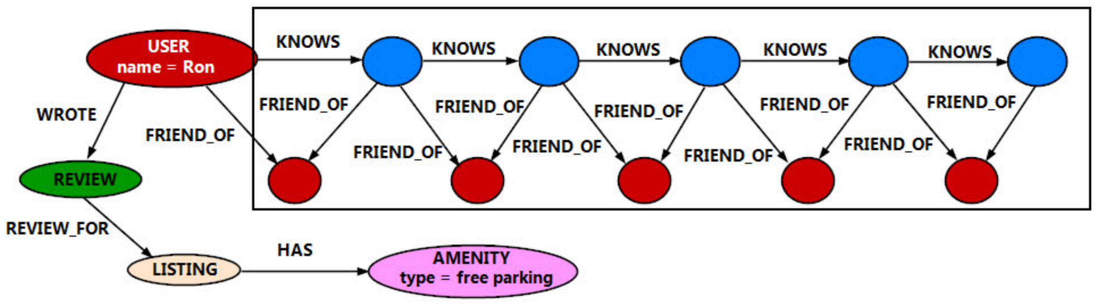


Figure 6.8: Structure of the sub graph for query presented in Example 14

6.3.4 Insights from the examples

Graph pattern matching queries share common formalisms of conjunctive queries and union of conjunctive queries. On the other hand, graph navigation queries do not share a common formalism as shown in Examples 12 and 13 furthermore, existing formalisms are not expressive enough. As presented in Example 14 expressing paths of arbitrary-lengths containing repeating acyclic or cyclic structure between the start and end nodes requires Kleene star operator. In formalisms based on two-way regular path queries and nested regular expressions, conjunction is not closed under Kleene star. On the other hand, in Tarski's algebra conjunction is closed under Kleene star; therefore, the query in Example 14 can be expressed in Tarski's algebra. Furthermore, combining conjunctive queries and union of conjunctive queries with Tarski's algebra yields CQT and UCQT. These formalisms are more expressive than existing formalisms such as C2RPQ, CNRE, UC2RPQ and UCN2RPQ. Moreover, CQT and UCQT serve as common formalisms that can be used to compare graph query languages. We discuss these formalisms in the following section.

6.4 An Integrated Formalism for graph query languages based on Tarski's algebra

This section relates to our first research objective RO1. We use the findings from RQ2 presented in Section 6.2.1, to extend the formalisms of conjunctive queries and union of conjunctive queries with Tarski's algebra. The formalisms are used later to compare practical query languages on the core features of graph pattern matching and graph navigation as presented in Section 6.2.2 for answering RQ1. This formalism allows us to carry out a systematic comparison shown later in Section 6.7. The most important elements of graph pattern matching and navigation queries are patterns, which assist in defining the structure of data that has to be extracted from a graph database [32, 62]. Patterns can represent a simplistic, one-node structure, as well as a complex structure over multiple nodes and along with their relationships. Broadly, patterns are of two types *navigation pattern* and *graph pattern*.

6.4.1 Navigation pattern

Navigation patterns are used to formulate graph navigation queries and are defined as follows:

Definition 10 (`Navigation pattern`) *Given a graph schema $\mathcal{G}_s$ and graph database $\mathcal{G}_d$, let $\mathcal{K}$ be a set of infinite keys, $\mathcal{W}$ be a set of infinite values and $\theta = \{\geq, \leq$*

$, >, <, =, \neq\}$ *be a set containing equality and inequalities. We define a set of expressions* $\mathtt{EXPR}_1 \subseteq (\mathcal{K} \times \theta \times \mathcal{W})$*. A navigation pattern is a graph defined by the tuple* $\mathcal{G}_p = (V_\mathcal{N}, V_\mathcal{E}, \mathcal{E}_p, \eta_p, \xi_p, \Delta_p, \Phi)$ *where:*

- $V_\mathcal{N}$ *is a finite set of node variables and* $V_\mathcal{E}$ *is a finite set of edge variables such that* $V_\mathcal{N} \cap V_\mathcal{E} = \varnothing$*.*
- Φ *is a topological order defined over the set of edge labels* $L_\mathcal{E}$*.*
- $\mathcal{E}_p = \mathcal{E}_{DE} \cup \mathcal{E}_\Phi$ *where,* $\mathcal{E}_{DE} \subseteq V_\mathcal{N} \times V_\mathcal{E} \times V_\mathcal{N}$ *is a relation that represents a directed edges and* $\mathcal{E}_\Phi \subseteq V_\mathcal{N} \times \Phi \times V_\mathcal{N}$ *is a relation that represent path(s) connecting two nodes. In cases when* $\mathcal{E}_p = \varnothing$ *mean that the navigation pattern only represents node(s).*
- $\eta_p : V_\mathcal{N} \to (L_\mathcal{N} \cup \epsilon)$ *is a node labelling function which maps all node variables to labels in a set of node labels* $L_\mathcal{N}$ *and a node variable can have an empty label.*
- $\xi_p : V_\mathcal{E} \to (L_\mathcal{E} \cup \epsilon)$ *is an edge labelling function which maps all edge variables to labels in a set of edge labels* $L_\mathcal{E}$ *and an edge variables can have an empty label.*
- $\Delta_p : (V_\mathcal{N} \cup V_\mathcal{E}) \to \wp(\mathtt{EXPR}_1)$ *is a property labelling function which maps all node and/or edge variables to the powerset of* $\mathtt{EXPR}$ *allowing each element in* $(V_\mathcal{N} \cup V_\mathcal{E})$ *to be mapped by no element, single or multiple elements from the set* $\mathtt{EXPR}$*.*

Let x be an element of the set $(V_\mathcal{N} \cup V_\mathcal{E})$, $k \in \mathcal{K}$ and $w \in \mathcal{W}$ if $\Delta_p(x) = (k, \theta, w)$ then we use the shorthand $x.k\theta w$ to define an expression, where θ can be one of the equality or inequalities.

Example 15 *Figure 6.9 shows a navigation pattern for the graph navigation query presented in Example 12. The relation* $\mathcal{E}_\Phi(u, \mathtt{KNOWS}^*, u_1)$ *is used to represent a* *path where a topological order defined over the edge label* `KNOWS` *(represented as* $(\texttt{KNOWS})^*$*) is specified between node variables u and u_1. Relations $\mathcal{E}_{DE}(u_1, f, u_2)$ and $\mathcal{E}_{DE}(u_1, o, l)$ are used to represent directed edges. Node and edge variables have labels and properties associated with them for instance, the node variable u has an associated label* `USER` *and a property is used to specify that name of the user should be* `Ron` *represented by the shorthand* `u.name = Ron`*.*

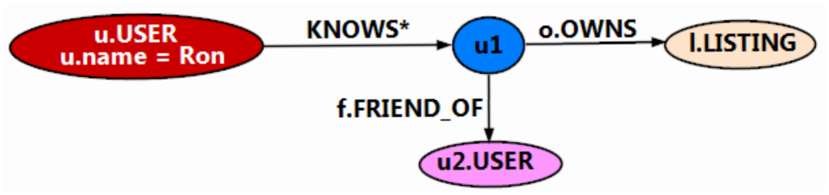


Figure 6.9: Navigation pattern for the graph navigation query presented in Example 12

Graph pattern

A graph pattern is a restricted form of the navigation pattern presented in Definition 10 and is defined as follows:

Definition 11 (Graph pattern) *A navigation pattern is a graph pattern when $\mathcal{E}_{\Phi} = \emptyset$ and $\Phi = \emptyset$ this means that graph patterns only consists of relations that represent directed edges that is $\mathcal{E}_p = \mathcal{E}_{DE}$.*

Example 16 *Figure 6.10 shows a graph pattern for the graph pattern matching query presented in Example 11. The relations $\mathcal{E}_{DE}(u, f_1, u_1), \mathcal{E}_{DE}(u_1, k, h), \mathcal{E}_{DE}(u_1, w, r), \mathcal{E}_{DE}(r, rf, l)$ and $\mathcal{E}_{DE}(l, h, a)$ are used to represent directed edges of the graph pattern. Node and edge variables have labels and properties associated with them for instance, the node variable u has an associated label* `USER` *and a property is used to specify that name of user should be* `Ron` *represented by the shorthand* `u.name = Ron`*.*

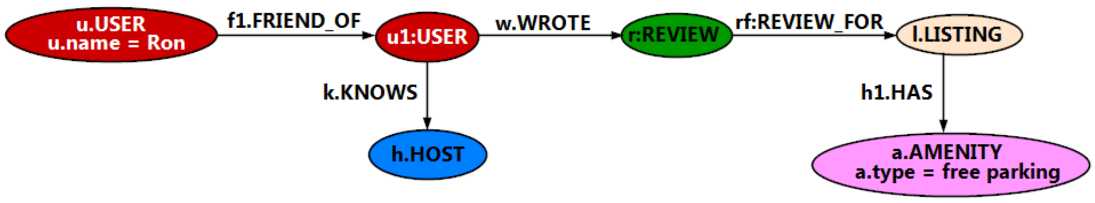


Figure 6.10: Graph pattern for the graph pattern matching query presented in Example 11

Navigation and graph patterns are described in a query language so that graph database's underlying query engine can interpret them [188]. The query engine then runs a pattern matching algorithm [51, 52] and outputs the result set in the form of a table[14]. Practical graph databases lack a standard query language and as discussed in Sections 6.2 and 6.3 do not share a common formalism. Furthermore, existing formalisms are not expressive enough; therefore, we discuss the extension of conjunctive queries and union of conjunctive queries with Tarski's algebra in the following section. We first discuss conjunctive queries and union of conjunctive queries.

6.4.2 Conjunctive queries and union of conjunctive queries

The most basic query language for databases is *conjunctive queries* [40, 115, 33] that is based on restricted formalism of first order logic [58, 198]. We use elements for Definition 10 to describe both graph and navigation patterns as conjunctive queries.

Definition 12 (`Conjunctive query`) *Given a set of head variables H, a set of body variables B, a set of atomic formulas A and a set of relations R, a conjunctive query (CQ) is a logical formula in the $\exists$, $\wedge$–fragment of first order logic [66, 199], that is an expression of the form* $CQ = \Big\{(h_1,\ldots,h_i) \mid \exists\,(b_1,\ldots,b_j)\ a_1 \wedge \ldots \wedge a_k \wedge r_1 \wedge \ldots \wedge r_l \Big\}$ *where:*

- $H = \{h_1,\ldots,h_i\}$ *is a finite set of head variables such that* $H \subseteq (\mathcal{N}_V \cup \mathcal{E}_V)$.

[14]Graph databases such as Neo4j also provide inbuilt graph visualization

- $B = \{b_1, \ldots, b_j\}$ *is a finite set of body variables such that* $B \subseteq (\mathcal{N}_V \cup \mathcal{E}_V)$ *and* $H \cap B = \varnothing$.
- $A = \{a_1, \ldots, a_k\}$ *is a finite set of atomic formulas formed by labelling functions* η_p, ξ_p *and* Δ_p *in definition 10, additionally atomic formulas can also belong to the set* $\texttt{EXPR}_2 \subseteq \big((V_{\mathcal{N}} \cup V_{\mathcal{E}}) \times \theta' \times (V_{\mathcal{N}} \cup V_{\mathcal{E}})\big)$ *where* $\theta' = \{\neq, =\}$. *Atomic formulas from the set* $\texttt{EXPR}_2$ *are used to ensure that node and/or edge variables can be compared in the query.*
- $R = \{r_1, \ldots, r_l\}$ *is a finite set of relations such that* $R = \mathcal{E}_p$.

Based on Definitions 10 and 12 conjunctive query represents a graph pattern if $R = \mathcal{E}_{\text{DE}}$.

Example 17 *The graph pattern in Figure 6.10 is represented as a conjunctive query shown as Query 8 where* $u, f_1, u_1, k, h, w, r, h_1, rf, a$ *are body variables. Relations* $\mathcal{E}_{DE}(u, f_1, u_1), \mathcal{E}_{DE}(u_1, k, h), \mathcal{E}_{DE}(u_1, w, r), \mathcal{E}_{DE}(r, rf, l)$ *and* $\mathcal{E}_{DE}(l, h_1, a)$ *are used to structurally describe the graph pattern. Atomic formulas such as* $\eta_p(u) = \texttt{"USER"}$ *are used to enforce node and/or edge labels based restrictions and* $u.\texttt{name} = \texttt{"Ron"}$ *is used to enforce the restriction that the user name should be* $\texttt{"Ron"}$. l *is a head variable and is used to output the result set.*

QUERY 8: Graph pattern in Figure 6.10 represented as a conjunctive query

$$\text{CQ} = \big\{l \mid \exists(u, f_1, u_1, k, r, w, h, rf, h_1, a)\ \mathcal{E}_{\text{DE}}(u, f_1, u_1) \wedge \mathcal{E}_{\text{DE}}(u_1, k, h) \wedge \mathcal{E}_{\text{DE}}(u_1, w, r) \wedge \mathcal{E}_{\text{DE}}(r, rf, l) \wedge \mathcal{E}_{\text{DE}}(l, h_1, a) \wedge \eta_p(u) = \texttt{USER} \wedge \xi_p(f_1) = \texttt{FRIEND\_OF} \wedge \eta_p(u_1) = \texttt{USER} \wedge \xi_p(k) = \texttt{KNOWS} \wedge \eta_p(h) = \texttt{HOST} \wedge \xi_p(w) = \texttt{WROTE} \wedge \eta_p(r) = \texttt{REVIEW} \wedge \xi_p(rf) = \texttt{REVIEW\_FOR} \wedge \eta_p(l) = \texttt{LISTING} \wedge \xi_p(h_1) = \texttt{HAS} \wedge \eta_p(a) = \texttt{AMENITY} \wedge a.\texttt{type} = \texttt{"free parking"} \wedge u.\texttt{name} = \texttt{"Ron"}\big\}$$

Conjunctive queries do not support disjunction between relations and atomic formulas; hence union of conjunctive queries have been proposed.

Definition 13 (`Union of conjunctive queries`) *Union of conjunctive queries (UCQ) represent the disjunction of conjunctive queries $\vee_{i=1}^{n}(CQ_i)$, where all conjunctive queries $CQ_1, \ldots, CQ_n$ share the same tuple of head variables and follow the disjunctive normal form [58].*

Union of conjunctive queries increase the expressiveness of conjunctive queries, for instance, if one wants to find out listings that were reviewed by a user such that the age of user should be either less than 25 or greater than 30, such a query can only be expressed as a UCQ.

Example 18 *Query 9 presents a query expressed as union of conjunctive queries. The query comprises two conjunctive queries that share the same output variable l, and both queries express the same graph pattern. The first query returns result set for users with age less than 25 while the second query returns the result set for users with age greater than 30.*

QUERY 9: A query represented as a union of conjunctive query

$$\begin{aligned}
&\text{UCQ} = \\
&\Big\{\big(l \mid \exists(w,r,f,u) \quad \mathcal{E}_{\mathtt{DE}}(u,w,r) \wedge \mathcal{E}_{\mathtt{DE}}(r,f,l) \wedge \eta_p(u) = \mathtt{USER} \wedge \xi_p(w) = \mathtt{WROTE} \wedge \\
&\quad \eta_p(r) = \mathtt{REVIEW} \wedge \xi_p(f) = \mathtt{REVIEW\_FOR} \wedge \eta_p(l) = \mathtt{LISTING} \wedge u.\mathtt{age} < \mathtt{25}\big) \\
&\vee \\
&\big(l \mid \exists(w,r,f,u) \quad \mathcal{E}_{\mathtt{DE}}(u,w,r) \wedge \mathcal{E}_{\mathtt{DE}}(r,f,l) \wedge \eta_p(u) = \mathtt{USER} \wedge \xi_p(w) = \mathtt{WROTE} \wedge \\
&\quad \eta_p(r) = \mathtt{REVIEW} \wedge \xi_p(f) = \mathtt{REVIEW\_FOR} \wedge \eta_p(l) = \mathtt{LISTING} \wedge u.\mathtt{age} > \mathtt{30}\big)\Big\}
\end{aligned}$$

Conjunctive queries and union of conjunctive queries represent navigation pattern if $R = (\mathcal{E}_{\mathtt{DE}} \cup \mathcal{E}_{\Phi})$. In case when Φ is defined as regular expressions then this formalism for defining navigation patterns correspond to conjunctive regular path queries (CRPQ), conjunctive two-way regular path queries (C2RPQ), union of conjunctive regular path queries (UCRPQ) and union of conjunctive two-way regular path queries (UC2RPQ). When Φ is defined as nested regular expressions then the formalism for defining navigation patterns correspond to conjunctive nested regular expressions (CNRE) and union of conjunctive nested two-way regular path queries (UCN2RPQ). As discussed in Section 6.3 these formalisms are not expressive enough; therefore, in this study, we define Φ by using Tarski's algebra.

6.4.3 Tarski's algebra for graph navigation

A very basic fragment of Tarski's algebra includes operations of *concatenation*, *union* and *Kleene star* $\mathsf{TA}(., |, *)$. This fragment is equivalent to the formalism of RPQ in expressiveness [128]. Adding *inverse* operation which is only defined over the edge labels of a graph database, yields $\mathsf{TA}(., |, *, ^-)$ which is equivalent to 2RPQ in expressiveness. Adding *projection* operation (π) to Tarski's algebra $\mathsf{TA}(., |, *, ^-, \pi)$makes it equivalent NRE in expressiveness. The projection operation π is used to specify a branched edge at an intermediate node.

To further increase the expressive power, Tarski's algebra is extended to support operations of *intersection* $\mathsf{TA}(., |, *, ^-, \pi, \cap)$ where *intersection* symbol $(\cap)$ can be used to express existence of cyclic or acyclic structures. Moreover, by using *concatenation* along with *intersection* series-parallel patterns can be expressed [74]. Tarski's algebra is further extended to support negation by adding operations of *co-projection* and *difference* $\mathsf{TA}(., |, *, ^-, \pi, \cap, \overline{\pi}, -)$. These operations can be used to specify the absence of certain pattern in a Tarski's algebra expression. Operations of *diversity* and *identity* are added to be specified only over the nodes $\mathsf{TA}(., |, *, ^-, \pi, \cap, \overline{\pi}, -, \mathtt{di}, \mathtt{id})$. *Diversity* operation enables to find all pairs of distinct nodes while *identity* operation enables to find all pairs of identical nodes [74, 183]

A Tarski's algebra expression τ defined over the edge labels, is used to specify a topological order between two nodes of a graph database. Grammar for defining expressions in Tarski's algebra is presented in Equation 6.1.

$$\tau ::= \texttt{id} \big| \texttt{di} \big| l_e \big| l_e^- \big| \pi[\tau] \big| \overline{\pi}[\tau] \big| \tau.\tau \big| \tau|\tau \big| \tau \cap \tau \big| \tau - \tau \big| (\tau)^* \tag{6.1}$$

Evaluating Tarski's algebra expressions

The output produced after evaluating a Tarski's algebra expression (τ) consists all pairs of nodes (start and end node) that satisfy the existence of a path in the graph database such that topological order of the path, matches the topological order defined in the expression (τ) [34, 74]. Given a LPG database $\mathcal{G}_d = (\mathcal{N}_d, \mathcal{E}_d)$ notation $[\![\tau]\!]_{\mathcal{G}_d}$ is used to denote *evaluation* of an expression τ over the graph database $\mathcal{G}_d$ [183, 35]. The semantics for evaluating the expression (τ) have been adopted from [183, 22, 35, 55, 74] and are as follows:

$[\![\texttt{id}]\!]_{\mathcal{G}_d} = \{(n,m) | n, m \in \mathcal{N}_d \wedge n = m\};$

$[\![\texttt{di}]\!]_{\mathcal{G}_d} = \{(n,m) | n, m \in \mathcal{N}_d \wedge n \neq m\};$

$[\![l_e]\!]_{\mathcal{G}_d} = \{(n,m) | n \xrightarrow{l_e} m \in \mathcal{E}_d \wedge l_e \in L_{\mathcal{E}}\}$

$[\![l_e^-]\!]_{\mathcal{G}_d} = \{(n,m) | n \xleftarrow{l_e} m \in \mathcal{E}_d \wedge l_e \in L_{\mathcal{E}}\}$

$[\![\pi[\tau]]\!]_{\mathcal{G}_d} = \{(n,n) | \exists m \in \mathcal{N}_d \wedge (n,m) \in [\![\tau]\!]_{\mathcal{G}_d}\}$

$[\![\overline{\pi}[\tau]]\!]_{\mathcal{G}_d} = \{(n,n) | \exists m \in \mathcal{N}_d \wedge (n,m) \notin [\![\tau]\!]_{\mathcal{G}_d}\}$

$[\![\tau.\tau]\!]_{\mathcal{G}_d} = \{(n,m) | \exists z \in \mathcal{N}_d \wedge (n,z) \in [\![\tau]\!]_{\mathcal{G}_d} \wedge (z,m) \in [\![\tau]\!]_{\mathcal{G}_d}$

$[\![\tau|\tau]\!]_{\mathcal{G}_d} = [\![\tau]\!]_{\mathcal{G}_d} \cup [\![\tau]\!]_{\mathcal{G}_d}$

$[\![\tau \cap \tau]\!]_{\mathcal{G}_d} = [\![\tau]\!]_{\mathcal{G}_d} \cap [\![\tau]\!]_{\mathcal{G}_d}$

$[\![\tau - \tau]\!]_{\mathcal{G}_d} = [\![\tau]\!]_{\mathcal{G}_d} - [\![\tau]\!]_{\mathcal{G}_d}$

$[\![(\tau)^*]\!]_{\mathcal{G}_d} = \cup_{i \geq 0} [\![\tau^i]\!]_{\mathcal{G}_d}$ such that $\tau^k = (\tau.\tau \ldots \texttt{k-times} \ldots \tau)$ where $0 \leq k \leq i$, $\tau^0 = \texttt{id}$ and $\tau^+ = \tau.\tau^*$.

6.4.4 Conjunctive queries and union of conjunctive queries extended with Tarski's algebra

We have seen how Tarski's algebra is more expressive than other formalism such as RPQ, 2RPQ and NRE. We add the expressive power of Tarski's algebra to conjunctive queries and call this formalism as conjunctive query with Tarski's algebra (CQT). In order to define CQT we use elements from the definitions of conjunctive queries (Definition 12), navigation patterns (Definition 10) and grammar of Tarski's algebra presented in Equation 6.1.

Definition 14 (`Conjunctive query with Tarski's algebra`) *Given sets H, B, A and R from Definition 12. A conjunctive query with Tarski's algebra (CQT) is an expression of the form CQT =* $\big\{(h_1, \dots, h_i) \mid \exists\, (b_1, \dots, b_j)\ a_1 \wedge \dots \wedge a_k \wedge r_1 \wedge \dots \wedge r_l\ \big\}$ *where:*

- *The sets* $H = \{h_1, \dots, h_i\}, B = \{b_1, \dots, b_j\}$ *and* $A = \{a_1, \dots, a_k\}$ *have same interpretation as Definition 12.*
- $R = \{r_1, \dots, r_l\}$ *is a finite set of relations such that* $R = \mathcal{E}_{DE} \cup \mathcal{E}_{\tau}$ *where* τ *is a Tarski's algebra expression defined over the set of edge labels of a graph database by using grammar presented in Equation 6.1.*

Example 19 *Figure 6.11 shows a navigation pattern for the graph navigation query presented in Example 14. The relation* $\mathcal{E}_{\tau}(u, (((\mathtt{FRIEND\_OF.FRIEND\_OF^{-}}) \cap \mathtt{KNOWS})^{*} \cap \mathtt{di}), u_2)$ *represents an arbitrary-length path where expression* $\tau = (((\mathtt{FRIEND\_OF.FRIEND\_OF^{-}}) \cap \mathtt{KNOWS})^{*} \cap \mathtt{di})$ *is used to specify a path formed of repeating acyclic structures. The diversity operation* `di` *is used to ensure that start and end nodes of the path are not same . Relations* $\mathcal{E}_{DE}(u, w, r), \mathcal{E}_{DE}(r, rf, l)$ *and* $\mathcal{E}_{DE}(l, h, a)$ *are used to represent directed edges. These three edges represent a path as they share* *common start and end node variables r and l. Furthermore, nodes and edges have associated labels and properties as per Definition 10.*

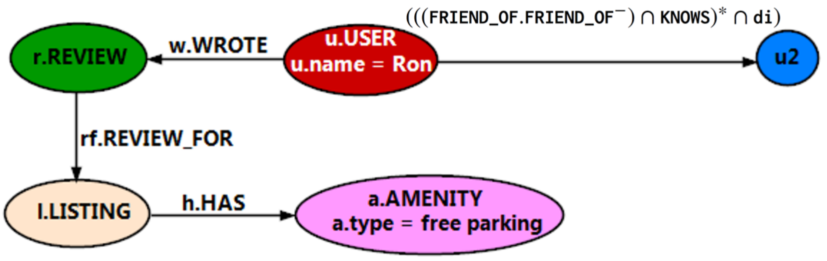


Figure 6.11: Navigation pattern for the graph navigation query presented in Example 14

Example 20 *The navigation pattern in Figure 6.11 is represented as a CQT query as shown in Query 10 where u, w, l, rf, h, a are body variables. Relation $\mathcal{E}_{\tau}(u, (((\mathtt{FRIEND\_OF.FRIEND\_OF^{-}}) \cap \mathtt{KNOWS})^{*} \cap \mathtt{di}), u_2)$ represents an arbitrary-length path while relations $\mathcal{E}_{DE}(u, w, r), \mathcal{E}_{DE}(r, rf, l)$ and $\mathcal{E}_{DE}(l, h, a)$ are used to represent fixed-length path. Both paths share a common variables u and all four relations are used to structurally describe the navigation pattern presented in Figure 6.11. Atomic formulas such as $\eta_p(u) = \mathtt{USER}$ are used to enforce node and/or edge labels based restrictions and $a.\mathtt{type} = \mathtt{free\ parking}$ are used to enforce property based restrictions. u_2 and r are output variables and are used to return the desired output.*

QUERY 10: Navigation pattern in Figure 6.11 expressed as CQT query

$$\text{CQT} = \Big\{u_2, r \;\Big|\; \exists(u, w, l, rf, h, a)\ \mathcal{E}_{\tau}(u, (((\mathtt{FRIEND\_OF.FRIEND\_OF^{-}}) \cap \mathtt{KNOWS})^{*} \cap \mathtt{di}), u_2) \ \wedge\ \mathcal{E}_{\mathtt{DE}}(u, w, r) \ \wedge\ \mathcal{E}_{\mathtt{DE}}(r, rf, l) \ \wedge\ \mathcal{E}_{\mathtt{DE}}(l, h, a) \ \wedge\ \eta_p(u) = \mathtt{USER} \ \wedge\ \xi_p(w) = \mathtt{WROTE} \ \wedge\ \eta_p(r) = \mathtt{REVIEW} \ \wedge\ \xi_p(rf) = \mathtt{REVIEW\_FOR} \ \wedge\ \eta_p(l) = \mathtt{LISTING} \ \wedge\ \xi_p(h) = \mathtt{HAS} \ \wedge\ \eta_p(a) = \mathtt{AMENITY} \ \wedge\ u.\mathtt{name} = \text{“}\mathtt{Ron}\text{”} \ \wedge\ a.\mathtt{type} = \text{“}\mathtt{free\ parking}\text{”}\Big\}$$

Very similar to conjunctive queries we propose the extension of CQT to union of conjunctive queries with Tarski's algebra.

Definition 15 (`Union of CQT`) *Union of conjunctive queries with Tarski's algebra (UCQT) represent the disjunction of conjunctive queries with Tarski's algebra* $\vee_{i=1}^{n}(CQT_i)$*, where all* $CQT_1,\ldots,CQT_n$ *share the same tuple of head variables and follow the disjunctive normal form.*

Union of conjunctive queries with Tarski's algebra increase the expressive power of conjunctive queries with Tarski's algebra as they enable the use of disjunction. Formalisms of CQT and UCQT are incomparable in expressiveness to extended conjunctive regular path queries (ECRPQ). This is because CQT and UCQT cannot compare and/or produced paths as output. On the other hand ECRPQ and their extensions cannot express arbitrary-length paths with branches, acyclic and cyclic structures. Moreover, path comparisons and the ability to output paths is not provided by all practical graph query languages[15].

Formalisms of CQT and UCQT are certainly more expressive than formalisms based on two-way regular path queries and nested regular expressions such as CRPQ, C2RPQ, CNRE, UCRPQ, UC2RPQ and UCN2RPQ. Therefore, we use formalisms of CQT and UCQT to compare practical graph query languages Cypher and PGQL.

The result set returned after evaluating a query depends upon the underlying implementation and evaluation algorithms used by a particular query language. Therefore, in order to objectively compare graph query languages, we have considered two criteria of *(i) syntactic equivalence* and *(ii) semantic equivalence*. We discuss syntactic equivalence in the following section and present semantic equivalence in Section 6.5.

6.4.5 Syntactic equivalence of queries

Based on Definitions 12, 13, 14 and 15 we can observe that formalisms used for expressing graph and navigation patterns such as CQ, UCQ,CQT and UCQT are

[15]Cypher provides the ability to output paths

similar. The only difference between these four formalisms is that CQT and UCQT consist of additional relations that represent path expressions. Two conjunctive queries (or conjunctive query with Tarksi's algebra) are equivalent if head variables, atomic formulas and relations defined for each query are also equal [40, 58]. For checking syntactic equivalence between queries we present Algorithm 1.

Algorithm 1: Check if two CQT queries are syntactically similar

```
Input: Queries $Q_1$ and $Q_2$
Output: TRUE or FALSE
1 if $Compare(Q_1.H, Q_2.H)$ AND $Compare(Q_1.A, Q_2.A)$ AND
    $Compare(Q_1.R, Q_2.R)$ then return TRUE;
2 else return FALSE;
3 Function Compare(X: Set, Y: Set) : Boolean is
4     if ($X \in \mathcal{E}_\mathcal{T}$ AND $Y \in \mathcal{E}_\mathcal{T}$ AND $X.\tau == Y.\tau$ AND $X == Y$) then return TRUE ;
5     else if ($X == Y$) then return TRUE;
6     else return FALSE;
7 end
```

Algorithm 1 takes two queries Q_1 and Q_2 as inputs. The algorithm calls a function *Compare* (lines 1) to check if the set of head variables (*H*), atomic formulas (*A*) and relations (*R*) are identical (lines 3-7). The compare function does a standard set comparison to check if two sets are of same size and elements are pairwise equivalent (line 5). To match two relations representing path expressions the *Compare* function uses grammar for Tarski's algebra presented in equation 6.1 to check if two path expressions are identical (line 4). Algorithm 1 returns TRUE if head variables, atomic formulas and relations in both the queries are identical (line 1) and FALSE otherwise (line 2).

Moreover, Algorithm 1 can also be used to check syntactic equivalence between queries expressed in formalisms such as union of conjunctive queries and union of conjunctive queries with Tarski's algebra since these formalisms are extensions of CQ and CQT under union.

6.5 Semantics of CQT and UCQT

Semantics of a query refers to finding the answer (or result set) of the query in a graph database [34, 62]. We discuss two types of semantics *(i) evaluation semantics* and *(ii) output semantics*. Evaluation semantics correspond to the underlying algorithm used by different query languages to search for the existence of patterns in a graph database. Output semantics are used to determine the result set generated by different query languages which might contain duplicates depending upon the output semantics used by different query languages. We discuss these criteria in detail in the following sections.

6.5.1 Evaluation semantics

Evaluation semantics for a CQT query that represents a graph or navigation pattern $\mathcal{G}_p$ against a graph database $\mathcal{G}_d$ corresponds to finding all the possible occurrences of $\mathcal{G}_p$ in $\mathcal{G}_d$ [200, 201]. A graph or navigation pattern might never occur in the graph database, but if the graph or navigation pattern occurs, then the pattern matching algorithm finds all sub-graphs in the graph database structurally similar to the graph or navigation pattern. We follow the same notion of matching a graph or navigation pattern against a graph database as discussed in [32, 62, 171, 173].

A graph or navigation pattern can contain *variables* (in form of node and edge variables) and *constants* such as node labels, edge labels and topological order defined over edge labels. Therefore, *match* $\mathcal{M}$ of $\mathcal{G}_p$ in $\mathcal{G}_d$ is a *homomorphism* mapping from variables and constants in a graph or navigation pattern to constants in a graph database. Formally, the mapping is defined as $\mathcal{M} : \mathcal{G}_p \rightarrow \mathcal{G}_d$, the graph or navigation pattern is a match in the graph database if the following conditions hold:

- All node and edge variables in a graph or navigation pattern are mapped to the graph database's node and edges.

- All node/edge labels in a graph or navigation pattern are mapped to node/edge labels in the graph database
- For every edge $\mathcal{E}_{\text{DE}}(n_i, e_j, n_k)$ of the graph or navigation pattern it holds that $(\mathcal{M}(n_i), \mathcal{M}(e_j), \mathcal{M}(n_k)) \in \mathcal{E}_d$ and $\mathcal{M}(n_i), \mathcal{M}(n_k) \in \mathcal{N}_d$.
- For every edge $\mathcal{E}_\tau(n_i, \tau, n_k)$ of the navigation pattern it holds that $(\mathcal{M}(n_i), \mathcal{M}(n_k)) \in [\![\tau]\!]_{\mathcal{G}_d}$ where $\mathcal{M}(n_i)$ is the start node and $\mathcal{M}(n_k)$ is the end node of the path satisfied by the expression τ and $\mathcal{M}(n_i), \mathcal{M}(n_k) \in \mathcal{N}_d$. As previously mentioned in Section 6.4.3 while evaluating path expressions we check for existence of all paths between two nodes that satisfy the path expression.

The mapping $\mathcal{M}$ is a non injective mapping [32, 73] which means that multiple variables in $\mathcal{G}_p$ can be mapped to same elements in $\mathcal{G}_d$.

Example 21 *In the graph database shown in Figure 6.4 a host `Renna` wants to find out the year since she has owned a listing named `Grafton House` and information about a user whom a friend of `Reena` knows. Query 11 is the CQT representation of this query.*

QUERY 11: Query in Example 21 represented as CQT

$$\text{CQT} = \Big\{\big(o, u \mid \exists(h, l) \quad \mathcal{E}_{\text{DE}}(h, o, l) \wedge \mathcal{E}_\tau(h, \texttt{(FRIEND\_OF.KNOWS)}, u) \wedge \eta_p(u) = \texttt{USER} \wedge \xi_p(o) = \texttt{OWNS} \wedge \eta_p(l) = \texttt{LISTING} \wedge \eta_p(h) = \texttt{HOST} \wedge h.\texttt{name} = \texttt{"Renna"} \wedge l.\texttt{name} = \texttt{"Grafton House"}\big)\Big\}$$

Table 6.2 shows the mapping from node and edge variables in Query 11 to the nodes and edges in the graph database. We can observe for instance that variable u is mapped to the node u_2 of graph database that is $\mathcal{M}(u) = u_2$. The values associated with this node can be accessed by using the dot notation u_2.`name` = `David`. The edge $(\mathcal{M}(h), \mathcal{M}(o), \mathcal{M}(l))$ is a valid edge in the graph database that is $(h_3, e_{19}, l_3) \in \mathcal{E}_d$.

The set of nodes returned by the path expression $[\![\texttt{(FRIEND\_OF.KNOWS)}]\!]_{\mathcal{G}_d}$ are $(\mathcal{M}(h), \mathcal{M}(u)) = \{(h_3, u_2), (h_1, u_1)\}$. The filter conditions based on node/edge labels and properties are used to restrict the result set for instance $\mathcal{M}(h)$.name = Renna indicates that the result set should be returned for a host named Renna. Hence the set of nodes (h_1, u_1) are not included in the final result set.

Table 6.2: Mapping between node/edge variable in Query 11 and graph database shown in Figure 6.4

Node/edge variables in navigation pattern	o	u	h	l
Nodes/edges in graph database	e_{19}	u_2	h_3	l_3

Evaluating a query expressed as the union of conjunctive queries with Tarski's algebra (UCQT) corresponds to combining the result set produced by all the conjunctive queries with Tarski's algebra (CQT) that are used to form a UCQ with the condition that all CQT must share the same tuple of head variables.

Example 22 *In the graph database shown in Figure 6.4, one can ask information about hosts who either own a listing named as* Grafton House *and are connected to people by path labelled with* KNOWS *relationship, or these hosts are known by users* Ron *and* David *and are connected to people by path labelled with* FRIEND_OF *relationship. Query 12 is the UCQT representation of this query, formed of two CQT that are used to express different navigation patterns, however, both the queries share the same output variable h.*

QUERY 12: Query in Example 22 represented as *UCQT*

$$\text{UCQT} = \Big\{\big(h \mid \exists(o,l,p)\quad \mathcal{E}_{\text{DE}}(h,o,l) \wedge \mathcal{E}_{\mathcal{T}}(h,(\texttt{KNOWS}^* \cap \texttt{di}),p) \wedge \xi_p(o) = \texttt{OWNS} \wedge \eta_p(l) = \texttt{LISTING} \wedge \eta_p(h) = \texttt{HOST} \wedge l.\text{name} = \text{``}\texttt{Grafton House}\text{''}\big)$$
$$\vee$$
$$\big(h \mid \exists(a,b,p)\quad \mathcal{E}_{\text{DE}}(a,k_1,h) \wedge \mathcal{E}_{\text{DE}}(b,k_2,h) \wedge \mathcal{E}_{\mathcal{T}}(h,(\texttt{FRIEND\_OF}^* \cap \texttt{di}),p) \wedge \eta_p(a) = \texttt{USER} \wedge \eta_p(b) = \texttt{USER} \wedge \xi_p(k_1) = \texttt{KNOWS} \wedge \xi_p(k_2) = \texttt{KNOWS} \wedge \eta_p(h) = \texttt{HOST} \wedge a.\text{name} = \text{``}\texttt{Ron}\text{''} \wedge b.\text{name} = \text{``}\texttt{David}\text{''}\big)\Big\}$$

The output produced after evaluating CQT or UCQT is a set of head variables and the corresponding values associated with the head variables represent the answer (or result set) of the query.

Example 23 *Outputs produced after evaluating Queries 11 and 12 are presented in Tables 6.3 and 6.4 respectively where we can see that the output is only displayed for the head variables o and u of Query 11, and head variable h of Query 12. We use the dot notation to display property information related to the head variables.*

Table 6.3: Result set produced by Query 11

o.since	**u.name**	**u.age**
2000	David	23

Table 6.4: Result set produced by Query 12

h.name	**h.age**
Renna	39
Shradha	29

6.5.2 Output semantics

In database query languages two types of output semantics exist *(i) set-based-* and *(ii) bag-based-semantics* [32, 202]. In set-based semantics, duplicate values are eliminated from the final result set while bag based semantics maintain the duplicate values. The importance of output semantics depends upon the type of output returned by a query. There are four types of outputs *(i) Boolean, (ii) nodes and/or edges, (iii) path* and *(iv) graph*. The boolean output is produced when there are no head variables in a query, and we only check the existence of the graph or navigation pattern in a graph database. If head variables are specified in the query, we get nodes and/or edges as output. When it is desirable to output the entire path connecting some nodes results in path as output.

Finally, when it is desirable to output, the entire graph, after evaluating the query, results in graphs as output. There is no difference between set and bag-semantics when the query's output is boolean, path and graphs [32]. The importance of set and bag-based semantics is vital when nodes and/or edges are considered outputs. For instance in case of cycles in a graph database there may exist many (possibly infinite) paths between two nodes; hence the query can return duplicate values for node and/or edges. Therefore, for CQT and UCQT we consider set-based output semantics.

Formalisms of CQT and UCQT use homomorphism based evaluation semantics for graph pattern matching, arbitrary path semantics for graph navigation and set based output semantics. On the other hand, practical query languages do not use same evaluation semantics and output semantics [32]. This means that two syntactically identical queries expressed in different query languages may not produce the same result set if underlying evaluation and output semantics differ.

6.5.3 Query equivalence

We consider the concept of query equivalence which states that given a graph schema $\mathcal{G}_s$, two queries Q_1 and Q_2 are equivalent if they produce the same result set for all databases instantiated by the graph schema $\mathcal{G}_s$ [40, 58, 203]. To prove that the result set generated by two syntactically identical queries expressed in different query languages is same, the underlying semantics must be the same. We discuss the evaluation and output semantics used by practical graph query languages and present some examples related to Cypher and PGQL.

Evaluation semantics used in practical graph query languages

Evaluation semantics for graph pattern matching: Broadly there are two types of evaluation semantics for graph pattern matching *(i) homomorphism based semantics* and *(ii) isomorphism based semantics*. Under homomorphism based semantics all possible sub-graphs in graph database that match the graph pattern are returned in the result set. This means that multiple node and/or edge variables in a graph pattern can match to same node and/or edge in the graph database. On the other hand isomorphism based semantics enforce restrictions on the result set based on conditions that either a node variable match to a single node (no-repeated-node semantics) or an edge variable only match to a single edge (no-repeated-edge semantics) or both node and/or edge variable match a single node and/or edge in a graph database (no-repeated-anything semantics) [32]. Homomorphism based semantics are less restrictive than Isomorphism based semantics.

Example 24 *A graph pattern that searches for nodes connected by* `KNOWS` *relationships is presented in Figure 6.12. This graph pattern is used to search for sub-graphs over the graph database shown in Figure 6.4. The graph pattern consists of three nodes represented by node variables x, y and z connected by two edges represented by edge variables k_1 and k_2. The central node should only have incoming edges from other nodes, and the label of the edges should be* `KNOWS`*.*

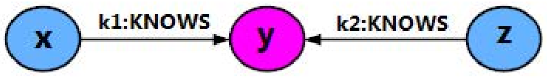


Figure 6.12: Example of a graph pattern

Example 25 *Queries 13 and 14 describe the graph pattern in Figure 6.12 expressed in Cypher and PGQL respectively. To display the associated edges, we use the inbuilt function* ID *provided by both the languages. The result set produced by both the queries is present in Table 6.5 that consists of names associated with each node and the edge IDs.*

QUERY 13: Graph pattern in Figure 6.12 expressed as a Cypher query

```
MATCH (x)-[k1:KNOWS]->(y), (y)<-[k2:KNOWS]-(z)
RETURN x.name, ID(k1), y.name, ID(k2), z.name
```

QUERY 14: Graph pattern in Figure 6.12 expressed as a PGQL query

```
SELECT x.name, ID(k1), y.name, ID(k2), z.name
MATCH (x)-[k1:KNOWS]->(y), (y)<-[k2:KNOWS]-(z)
```

As shown in Table 6.5 running the query in PGQL returns 14 rows whereas Cypher only return 6 rows. The result set returned by Cypher is shown between rows 1-6 where we can observe that in each row, single edge in the graph database is mapped to only one edge variable. This is because Cypher uses no-repeated-edge isomorphism based semantics whereas PGQL follows less restrictive homomorphism based semantics. The result set returned by a query expressed in PGQL contains all valid matches even in cases where multiple node and/or edge variables are mapped to the same node and/or edge of the graph database.

Table 6.5: Result set generated by Cypher and PGQL for graph pattern in Figure 6.12

			x.name	**ID(k1)**	**y.name**	**ID(k2)**	**z.name**
C		1	Ron	e_{25}	Shradha	e_{27}	David
Y		2	Ron	e_{26}	John	e_{37}	David
P		3	Rohan	e_{22}	Reena	e_{23}	Dave
H		4	David	e_{27}	Shradha	e_{25}	Ron
E		5	David	e_{37}	John	e_{26}	Ron
R	P	6	Dave	e_{23}	Renna	e_{22}	Rohan
	G	7	Ron	e_{25}	Shradha	e_{25}	Ron
	Q	8	Ron	e_{26}	John	e_{26}	Ron
	L	9	Rohan	e_{22}	Renna	e_{22}	Rohan
		10	Renna	e_{24}	Ron	e_{24}	Renna
		11	John	e_{28}	David	e_{28}	John
		12	David	e_{27}	Shradha	e_{27}	David
		13	David	e_{37}	John	e_{37}	David
		14	Dave	e_{23}	Renna	e_{23}	Dave

*Evaluation semantics for graph navigation:*The evaluation semantics for graph navigation queries are as follows:

- *Arbitrary path semantics* consider all paths that can be included in the result set of a graph navigation query. However, a query may return infinite paths; therefore, under these semantics, we are only interested in the existence of such paths, and actual paths are not returned [30, 115, 52].

- *Shortest path semantics* only consider the shortest paths returned after evaluating a graph navigation query. Such queries correspond to finding "*top-k*" shortest paths connecting nodes[32].
- *No-repeated-node semantics* allow paths where the same node cannot occur more than once in the result set of a graph navigation query. Such paths are commonly known as simple paths [204, 205].
- *No-repeated-edge semantics* allow paths that only have distinct edges which means that in cases where cycles exist in a graph database, infinite paths are not returned in the evaluation of a graph navigation query under these semantics [32, 51].

Example 26 *Queries 15 and 16 represent queries expressed in Cypher and PGQL respectively that are used to search for paths of arbitrary-lengths where edges are labelled by* `KNOWS` *relationship. The result set for these queries presented in Table 6.6 is only generated for the host* `Renna` *where the start and end nodes of the paths are provided as outputs.*

QUERY 15: Navigation pattern representing the expression `(KNOWS)`* expressed as a Cypher query

```
MATCH (a:HOST)-[:KNOWS*1..]->(b)
WHERE a.name = "Renna"
RETURN a.name,b.name
```

The graph database shown in Figure 6.4 contains a cycle between node h_1(host named `Shradha`) and node u_2(user named `David`) where the edges are labelled by `KNOWS` relationship. The number of paths found after evaluating an arbitrary-length

QUERY 16: Navigation pattern representing the expression (KNOWS)* expressed as a PGQL query

```
SELECT a.name,b.name
MATCH (a:HOST)-/:KNOWS+/->(b)
WHERE a.name = "Renna"
```

graph navigation query can be infinite in case cycles exist in graph database. Practical graph query languages avoid such scenarios by using different evaluation semantics. Cypher uses no-repeated-edge semantics while PGQL uses arbitrary path semantics.

In the graph database shown in Figure 6.4 there exist two paths that contain no-repeated-edges labelled by KNOWS relationship between node h_3(host named Renna) and node h_1(host named Shradha). Therefore, the result set produced by Queries 15 expressed in Cypher returns two rows where the start node is Reena and end node is Shradha as shown in Table 6.6. On the other hand PGQL uses arbitrary path semantics therefore, the number of paths are infinite. Such scenarios are avoided in PGQL by not allowing duplicate rows to be returned in the result set.

Additionally, both query languages also allow the use of shortest path semantics for finding shortest paths connecting any given nodes in a graph database.

Table 6.6: Result set generated by Cypher and PGQL for Queries 15 and 16

		a.name	**b.name**
P	**C**	Renna	Ron
G	**Y**	Renna	Shradha
Q	**P**	Renna	John
L	**H**	Renna	David
	E	Renna	Shradha
	R	Renna	John

Output semantics used in practical graph query languages

The choice of output semantics depends on different evaluation semantics used by query languages. For example, PGQL employs set based output semantics along with arbitrary path semantics for graph navigation queries. Therefore, duplicate records are eliminated from the result set, as shown in Table 6.6. Cypher employs bag based output semantics and no-repeated-edge semantics for graph navigation queries. Moreover, using bag semantics along with arbitrary path semantics are problematic [32]. For graph pattern matching both query languages use bag based output semantics.

As previously discussed the choice of output semantics also depends upon the type of output returned by the query language. Both Cypher and PGQL allow Boolean and node and/or edge as outputs. The ability to output paths is only provided by Cypher and since PGQL uses arbitrary path semantics outputting paths can be problematic as infinite paths can be returned. Cypher does provide the ability to visualize graphs, but PGQL does not provide this functionality. Furthermore, graphs are not provided as outputs in both query languages.

6.5.4 Semantic equivalence of queries

To show the semantic equivalence between two queries expressed in different graph query languages the evaluation semantics and output semantics must be the same. Graph query languages such as Cypher and PGQL do not share the same evaluation and output semantics. Furthermore, semantics are dependent on the implementation of query languages. Therefore, semantic equivalence cannot be proved between queries without making syntactic adjustments.

Table 6.7: Evaluation and output semantics used by Cypher and PGQL

	Cypher	**PGQL**
GPM	No-repeated-edge isomorphism and bag semantics (†, ⊛)	Homomorphism and bag semantics (⊛)
GN	No-repeated-edge isomorphism and bag semantics (⊛)	Arbitrary path and set semantics

GPM = Graph Pattern Matching

GN = Graph Navigation

⊛ = Set based output semantics can be enforced

† = Homomorphism based evaluation semantics can be enforced

The evaluation and output semantics for graph pattern matching and graph navigation queries used in Cypher and PGQL are summarized in Table 6.7. We can observe that set based output semantics can be enforced for graph pattern matching and graph navigation queries in Cypher and PGQL by using the `DISTINCT` keyword. For graph pattern matching PGQL uses homomorphism based evaluation semantics by default while in Cypher homomorphism based evaluation semantics can be enforced by using multiple `MATCH` clauses [32, 171]. The dissimilarity is observed for evaluation semantics in graph navigation queries as Cypher uses no-repeated-edge isomorphism based semantics while PGQL uses arbitrary path semantics and syntactic adjustments cannot be made to enforce particular evaluation semantics.

6.6 An Integrated Framework for describing graph queries

The integrated framework discussed in this section relates to RO2. We use the formalisms of CQT and UCQT to measure graph query languages' expressiveness. To formulate queries, we consider some common graph query patterns such as chains and cycles identified via RQ3 and discussed in Section 6.2.3. These graph query patterns are then turned into queries to measure the expressiveness of graph query languages.

6.6.1 Equivalence between queries expressed in different graph query languages

Queries expressed in different graph query languages are equivalent if they represent the same graph or navigation pattern and the result set produced after evaluating the query is also same. In order to show that queries express same graph or navigation pattern we first convert the queries into CQT and UCQT based representation and then use Algorithm 1 to show syntactic equivalence. To show that the result set produced by syntactically identical queries is also the same, we enforce the same evaluation and output semantics.

6.6.2 Expressing graph query patterns

Graph query languages use graph and navigation patterns to extract data from graph databases. In this section, we look at various graph query patterns discussed in the literature such as in [64, 77, 184, 74, 189, 166] that we use to formulate graph and navigation patterns to compare graph query languages. Graph query pattern are broadly classified as *chain* and *cycle* patterns. A chain is a path where the start and end node of a path are not the same. In contrast, a cycle is a path where the start and end node are the same.

Chain patterns and their extensions

Chain pattern, as shown in Figure 6.13, are the most basic patterns, and other patterns are their extensions. The extension of chain patterns include shapes such as *tree, star, star-chain, chainset,* and *forest* [77]. A tree pattern, as shown in Figure 6.14 contains a start(or root) node with no incoming edge, and other nodes in a tree have exactly one incoming edge. A star pattern, as shown in Figure 6.15 contains at most one node that has more than two incoming and/or outgoing edges. A star-chain, as shown in Figure 6.16 is a chain of star patterns. A chainset as shown in Figure 6.17 is formed by the union of set of disjoint chains, and a forest as shown in Figure 6.18 is formed by the union of set of disjoint trees.

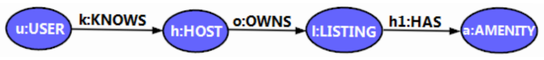


Figure 6.13: Chain pattern

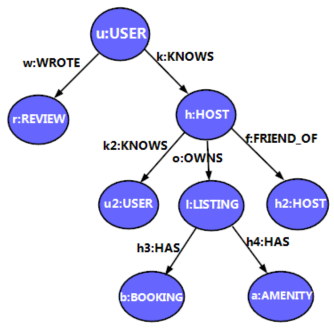


Figure 6.14: Tree pattern

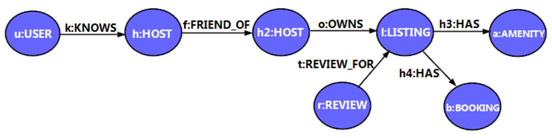


Figure 6.15: Star pattern

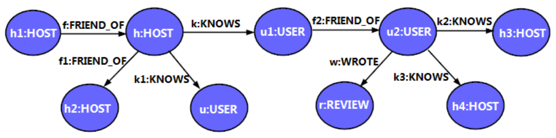


Figure 6.16: Star-chain pattern

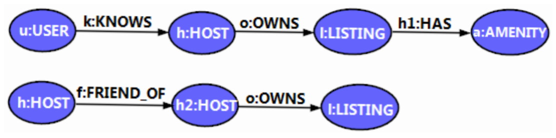


Figure 6.17: Chain set pattern

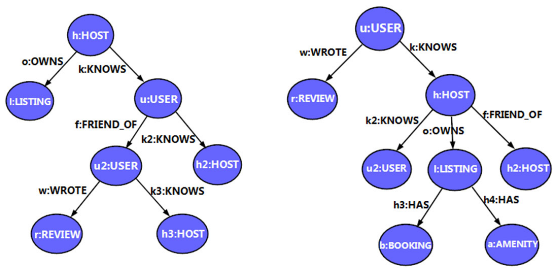


Figure 6.18: Forest pattern

Acyclic and cyclic patterns

The simplest form of acyclic patterns is *petal* that consists of a start node, an end node and at least two distinct chains connecting the start and end nodes as shown in

Figure 6.20. *Plain cycle* patterns are chains with the same start and end node, as shown in Figure 6.19. Other cyclic patterns include *flowers* and *bouquet*. A flower pattern, as shown in Figure 6.21 has one central node and three types of patterns such chains, trees and petals all three are attached to the same central node. For example, the flower pattern in Figure 6.21 contains five chains, two petals and one tree pattern connected to a central node. Flower patterns have a property that the path length of chains, petals and trees is two [166]. A bouquet pattern, as shown in Figure 6.22 is formed by the union of set of disjoint flower patterns.

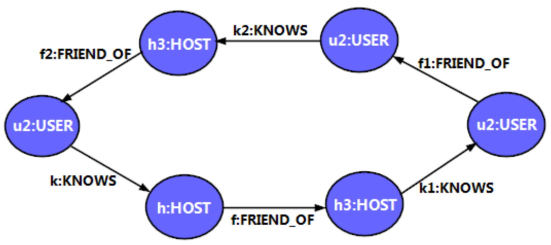


Figure 6.19: Plain cycle pattern

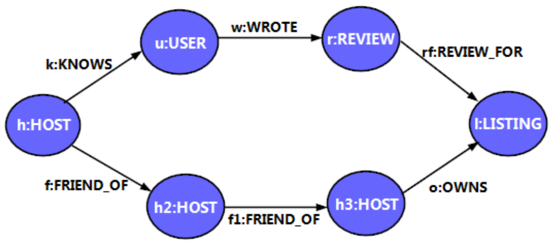


Figure 6.20: Petal pattern

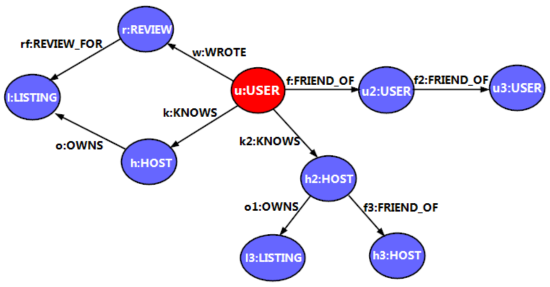


Figure 6.21: Flower pattern

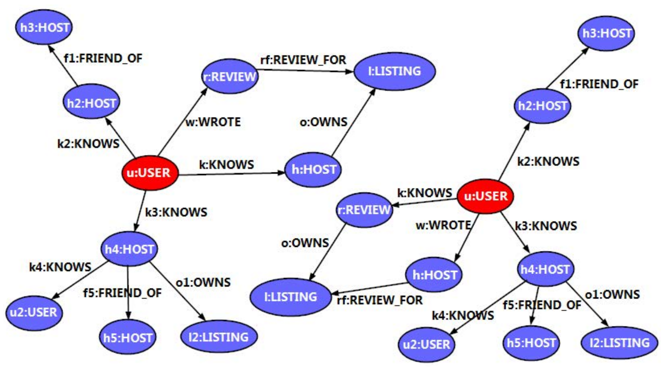


Figure 6.22: Bouquet pattern

Properties of patterns

Patterns have various properties associated with them such as *path length*, *maximum degree* and *length of cycle*. *Path length* refers to the longest path in a pattern. *Maximum degree* refers to the total number of incoming and/or outgoing edges of a node in a pattern, and *length of cycle* refers to the longest length of a chain used to form a cyclic pattern.

Table 6.8: Properties of graph query patterns

Pattern types	**Properties**	
Chain	**Path Length**	**Maximum Degree**
	1-3	2
Star	**Path Length**	**Maximum Degree**
	2-4	3-4
Tree	**Path Length**	**Maximum Degree**
	3-6	3-5
Cycle	**Length of cycle**	
	3-6	

The values associated with properties of various patterns are derived from the existing studies such as in [184, 166]. The pattern properties and associated values are summarized in Table 6.8 where we can observe that majority of *chain* patterns have path length of `1-3` and the maximum degree of inner nodes is `2`. *Star* patterns have a path length of `2-4` and maximum degree varies between `3-4`. *Tree* patterns have longest path length of `3-6` and a maximum degree of nodes ranges between `3-5`. *Cycle* patterns have length between `3-6`.

6.7 Comparing Cypher and PGQL using CQT and UCQT

The integrated framework allows us to construct a comprehensive set of graph pattern matching and navigation benchmark queries that can be used to compare the expressiveness practical graph query. We use the integrated framework to evaluate the expressiveness of two popular graph query languages Cypher and PGQL.

6.7.1 Benchmark queries for measuring expressiveness

We use graph query patterns presented in Section 6.6.2 to construct graph and navigation patterns. These patterns are then turned into queries and tested for support by Cypher and PGQL. We test if each query language can syntactically and logically express each pattern. We then run each query on the Neo4j and Oracle databases, which were deployed on identical hardware configurations. The only difference in the experimental setup is that Neo4j runs on the Windows operating system while Oracle runs on Linux. The benchmark queries for graph pattern matching consist of ten queries representing shapes such as chain, tree, star, chainset, forest, star chain, plain cycle, petal, flower and bouquet. We use the same graph patterns for graph navigation queries as we used for graph pattern matching queries. The only variation between graph pattern matching and graph navigation queries is that for graph navigation we use Kleene star operation over the graph patterns presented in Figures 6.13, 6.17, 6.14, 6.18, 6.15 , 6.16, 6.19, 6.20, 6.21 and 6.22. Additionally, we also consider chains formed out of multiple edge labels, star formed by chains of single and multiple edge labels.

6.7.2 Equivalence of queries expressed in Cypher and PGQL

Syntactic equivalence of queries

In order to show syntactic equivalence between queries we first discuss the conversion of queries expressed in Cypher and PGQL into CQT and UCQT based representation. As discussed in Section 6.4 the formalisms of CQT and UCQT are based on conjunctive queries and union of conjunctive queries. Therefore, queries expressed in CQT and UCQT also correspond to the *selection, projection, join* and *union* (SPJU) fragment of relational algebra. The *join* or more specifically *natural join* in graph and/or navigation patterns is based on the notion of compatible mappings [32, 171, 172] which states that relations/edges in graph or navigation patterns are join compatible if some of the node variables are same. For example in Query 11 relations $\mathcal{E}_{\texttt{DE}}(h, o, l)$ and $\mathcal{E}_{\mathcal{T}}(h, \texttt{FRIEND\_OF.KNOWS}, u)$ are joined based on the common node variable h. *Selection* is used to enforce some restriction on the result set in the query, for example, the condition that node and edge variables in the query must belong to certain labels and properties such as host name should be `Renna` represent selection. *Projection* is used to output the result set for example, in Query 11 the head variables u, o are used to output result of the query. Finally *union* is used to combine the result set from different CQT. Table 6.9 shows the mapping between relational algebra operations and clauses in Cypher and PGQL.

Table 6.9: Relational Algebra like operations in Cypher and PGQL

Relational Algebra Operations	Cypher	PGQL
Natural join	`MATCH`	`MATCH`
Selection	`WHERE`	`WHERE`
Projection	`RETURN`	`SELECT`
Union	`UNION`	`N/A`

Both query languages use `MATCH` clause to express a graph/navigation pattern or natural join of graph/navigation patterns where `ASCII` art is used to draw the pattern(s). For example, the navigation pattern in Query 11 can be expressed in the `MATCH` clause of Cypher and PGQL as the following pattern `(h:HOST)-[o:OWNS]->(l:LISTING), (h)- [FRIEND_OF]->()-[:KNOWS]->(u:USER).` In the pattern h, l, u are node variables while o is the edge variable. Moreover `(h:HOST)-[o:OWNS]->(l:LISTING)` represents an edge $\mathcal{E}_{\text{DE}}(h, o, l)$ while `(h)- [FRIEND_OF]->()-[:KNOWS]->(u:USER)` represents a path connecting two nodes that is $\mathcal{E}_{\mathcal{T}}(h, \texttt{FRIEND\_OF.KNOWS}, u)$. The symbol `:` signifies that nodes and/or edges have been assigned a label for example `u:USER` signifies the labelling $\eta_p(u) = \texttt{USER}$ similarly `w:WROTE` means $\xi_p(w) = \texttt{WROTE}$. The `WHERE` clause is optional in both the languages [171, 51, 52] and represents selection in Cypher and PGQL. For projection Cypher uses the `RETURN` clause while PGQL uses the SQL like `SELECT` clause.

Both languages support the use of logical `OR` in the `WHERE` clause. Even though Queries 17 and 18 are not in the disjunctive normal form (DNF). The queries can be transformed into DNF and represented as a UCQT.

QUERY 17: A navigation pattern expressed as a Cypher query

```
MATCH (u:USER)-[w:WROTE]->(r:REVIEW),
 (r)-[:REVIEW_FOR]->()-[:HAS]->(a:AMENITY)
WHERE u.age < 25 OR u.age > 30
RETURN u,w,r
```

QUERY 18: A navigation pattern expressed as a PGQL query

```
SELECT u,w,r
MATCH (u:USER)-[w:WROTE]->(r:REVIEW),
 (r)-[:REVIEW_FOR]->()-[:HAS]->(a:AMENITY)
WHERE u.age < 25 OR u.age > 30
```

However, the `UNION` clause is only present in Cypher while PGQL does not supports the clause. This means that some queries that can be expressed in Cypher cannot be expressed in PGQL. For example, Query 12 can only be expressed in Cypher because both CQTs in Query 12 are structurally different and the result set cannot be combined without the explicit use of `UNION` clause. Based on the discussion presented in this section we can conclude that every query expressed in Cypher and PGQL by using the four clauses presented in Table 6.9 can be represented in the formalisms of CQT and UCQT. Moreover, syntactic equivalence between such queries can be shown by using Algorithm 1.

Semantic equivalence of queries

As discussed in Section 6.5.4 Cypher and PGQL do not share common evaluation and output semantics. Therefore, in order to show semantic equivalence, evaluation and output semantics have to be syntactically enforced. The choice of semantics to be enforced depends upon the query languages being compared. As shown in Table 6.7 homomorphism based evaluation semantics can be enforced in Cypher for graph pattern matching queries and set based output semantics can be enforced in both the languages. Moreover, these semantics are ideal because this comparison is based on the formalisms of CQT and UCQT that use homomorphism based evaluation semantics for graph pattern matching queries and set based output semantics. For graph navigation queries we choose arbitrary path evaluation semantics since they are common between PGQL and CQT(UCQT), these semantics cannot be enforced in Cypher for graph navigation queries. Furthermore, nodes and edges are considered as outputs of queries since these are common output types in Cypher, PGQL and formalisms of CQT and UCQT.

6.7.3 Findings from benchmark queries

Graph pattern matching: As shown in Table 6.10 graph patterns such as *chain, tree, star, star chain, cycle, petal* and *flower* can be expressed in both the languages. Moreover, conjunctive queries represent the common formalism in both languages. Cypher can express graph patterns such as *chain set, bouquet* and *forest* whereas PGQL cannot express such patterns. The main reason behind this is the absence of the `UNION` clause in PGQL which makes it unable to combine result set of two or more conjunctive queries that do not represent same graph pattern. For instance, the chainset graph pattern and forest graph pattern shown in Figures 6.17 and 6.18 respectively are made up of different chains and trees. Expressing such graph patterns requires the explicit use of `UNION` clause. A shown in Table 6.10 PGQL only allows limited use of union of conjunctive queries. This is because we can still use the Boolean connector `OR` in the `WHERE` clause of PGQL and such queries can be expressed as the union of conjunctive queries. Based on the experiments presented in Table 6.10 we can observe that concerning graph pattern matching queries Cypher is more expressive than PGQL.

Table 6.10: Comparison between Cypher and PGQL for graph pattern matching queries

Exp	**Graph pattern**	**Formalism used in Cypher**	**Formalism used in PGQL**
1	Chain	CQ	CQ
2	Tree	CQ	CQ
3	Star	CQ	CQ
4	Star Chain	CQ	CQ
5	Cycle	CQ	CQ
6	Petal	CQ	CQ
7	Flower	CQ	CQ
8	Chainset	UCQ	UCQ ⋄
9	Bouquet	UCQ	UCQ ⋄
10	Forest	UCQ	UCQ ⋄

⋄ = Limited use of formalism

Graph navigation: We considered seventeen navigation patterns for analysing graph navigation queries in Cypher and PGQL. For chain shaped patterns, we considered two variations *chain formed by a single repeating edge label* and *chain formed by multiple repeating edge labels*. As shown in Table 6.11 experiment 1 can be performed by both the query languages, whereas experiment 2 can only be performed in PGQL. In Cypher Kleene star operation cannot be applied over the concatenation of two or more different edge labels; therefore, only navigation patterns representing chains formed by single edge labels can be expressed in Cypher. On the other hand navigation patterns in experiments, 1 and 2 can be expressed in PGQL where `PATH` clause is used to express a pattern and then Kleene star operation is applied over the pattern in the `MATCH` clause. In PGQL `PATH` clause is not a stand-alone clause and has to be used with the `MATCH` clause. Navigation patterns in experiments 1 and 2 can be expressed in CQT by using the $\mathsf{TA}(.,|,^-,*)$ fragment of Tarski's algebra.

Table 6.11: Comparison between Cypher and PGQL for graph navigation queries

Exp	Navigation patterns	Cypher	PGQL	Fragment of Tarski's algebra	Formalism
1	Chain with single edge label	✓	✓	$\mathsf{TA}(.,\|,^-,*)$	CQT
2	Chain with multiple edge labels	N/A	✓	$\mathsf{TA}(.,\|,^-,*)$	CQT
3	Tree	N/A	✓	$\mathsf{TA}(.,\|,^-,*,\pi)$	CQT
4	Star	N/A	✓	$\mathsf{TA}(.,\|,^-,*,\pi)$	CQT
5	Star chain	N/A	✓	$\mathsf{TA}(.,\|,^-,*,\pi)$	CQT
6	Star with chains of single edge label	✓	✓	$\mathsf{TA}(.,\|,^-,*)$	CQT
7	Star with chains of multiple edge label	N/A	✓	$\mathsf{TA}(.,\|,^-,*)$	CQT
8	Cycle	N/A	N/A	$\mathsf{TA}(.,\|,^-,*,\cap)$	CQT
9	Petal	N/A	N/A	$\mathsf{TA}(.,\|,^-,*,\cap)$	CQT
10	Flower	N/A	✓	$\mathsf{TA}(.,\|,^-,*,\cap,\pi)$	CQT
11	Chainset	N/A	N/A	$\mathsf{TA}(.,\|,^-,*)$	CQT
12	Forest	N/A	N/A	$\mathsf{TA}(.,\|,^-,*,\pi)$	CQT
13	Bouquet	N/A	N/A	$\mathsf{TA}(.,\|,^-,*,\cap,\pi)$	CQT
14	Difference	N/A	✓	$\mathsf{TA}(.,\|,^-,*,\overline{\pi})$	CQT
15	Union of chains with single edge label	✓	N/A	$\mathsf{TA}(.,\|,^-,*)$	UCQT
16	Union of Star with chains of single edge label	✓	N/A	$\mathsf{TA}(.,\|,^-,*)$	UCQT
17	Union of Trees	N/A	N/A	$\mathsf{TA}(.,\|,^-,*,\pi)$	UCQT

Tree, star and star chain shaped navigation patterns presented as experiments 3, 4 and 5 can only be expressed in PGQL whereas Cypher does not express such navigation patterns. It turns out that in PGQL tree and star patterns are expressed by using the `PATH` and `WHERE EXISTS` clauses together. The `MATCH` clause is used to apply the Kleene star over the patterns.

Example 27 *As shown in Query 19 the `PATH` clause describes a path starting from node `u:USER` and with end node `a:AMENITY`. Along the path, branches are existentially quantified using the `WHERE EXISTS` clause at nodes `h, l` and `u` respectively that describes a sub query. Finally the `MATCH` clause is used to apply the Kleene star over the entire tree structure expressed in the `PATH` clause.*

QUERY 19: Graph navigation query for a tree shaped navigation pattern expressed in PGQL

```
PATH p1 AS
 (u:USER)-[:KNOWS]->(h)-[:OWNS]->(l:LISTING)-[:HAS]->(a:AMENITY)
WHERE EXISTS (SELECT * MATCH (h)-[:FRIEND_OF]->())
  AND EXISTS (SELECT * MATCH (l)-[:HAS]->())
  AND EXISTS (SELECT * MATCH (u)-[:KNOWS]->())
SELECT DISTINCT x,y
MATCH (x)-/:p1+/->(y)
```

Navigation patterns in experiments 3, 4 and 5 can be expressed as CQT by using the $\mathsf{TA}(.,|,^{-},*,\pi)$ fragment of Tarski's algebra. We consider two more variations of star-shaped navigation patterns: stars formed by chains of single but distinct edge labels (experiment 6) and stars formed by chains of multiple edge labels (experiment 7). As shown in Table 6.11 both Cypher and PGQL can express navigation pattern in experiment 6. However, experiment 7 can only be expressed in PGQL, because Cypher does not allow the use of Kleene star over the concatenation of two or more edge labels. These navigation patterns can be expressed as CQT by using $\mathsf{TA}(.,|,^{-},*)$ fragment of Tarski's algebra.

As shown in experiments 8 and 9 both Cypher and PGQL cannot express navigation patterns to search for repeated cyclic and acyclic structures between nodes. These navigation patterns can be expressed in CQT by using the $\mathsf{TA}(.,|,^{-},*,\cap)$ fragment of Tarski's algebra. The `PATH` clause in PGQL does not allow comma-separated edges to be specified in a pattern as done in the `MATCH` clause and only chain, tree and star-shaped patterns can be expressed inside the `PATH` clause. Cyclic and acyclic navigation patterns are harder to evaluate, and hence many graph query language tends not to include them in their implementations [77]. Moreover, cyclic structures are also not that common in real-world data sets [166]. A navigation pattern representing a flower shape is presented as experiment 10 only PGQL can express such a pattern with the limitation that flower pattern must only be formed of chain and tree.

Experiments 11, 12 and 13 cannot be expressed in both the query languages. Experiment 11 requires the use of Kleene star over the union of two or more chains patterns. Similarly, experiments 12 and 13 require Kleene star over the union of two or more tree patterns and flower patterns, respectively. As shown in Table 6.11 chainset patterns can be expressed as CQT by using $\mathsf{TA}(., |,^{-}, *)$ fragment, forest pattern can be expressed as CQT by using $\mathsf{TA}(., |,^{-}, *, \pi)$ fragment and bouquet pattern can be expressed as CQT by using $\mathsf{TA}(., |,^{-}, *, \pi, \cap)$ fragment of Tarski's algebra.

Navigation patterns that search for the absence of certain pattern in a path are presented as experiment 14 in Table 6.11. Such a navigation pattern can only be expressed in PGQL by using the `WHERE NOT EXISTS` clause along with the `PATH` clause. These navigation patterns can be expressed as CQT by using the $\mathsf{TA}(., |,^{-}, *, \overline{\pi})$ fragment of Tarski's algebra.

Results presented in Table 6.11 are obtained by using the formalism of CQT. These results can be easily extended to the formalism of UCQT as discussed in Definition 15 queries expressed in UCQT combine the result set obtained from individual CQT. We consider three more navigation patterns presented in Table 6.11 as experiments 15, 16 and 17 all three navigation patterns cannot be expressed in PGQL where as experiments 15 and 16 can only be expressed in Cypher because of the presence of `UNION` clause. Navigation patterns in experiments 15 ,16 and 17 can be expressed as UCQT by using the $\mathsf{TA}(., |,^{-}, *)$ fragment and $\mathsf{TA}(., |,^{-}, *, \pi)$ fragment of Tarski's algebra respectively.

Based on the experiments presented in Table 6.11 it turns outs that PGQL is more expressive than Cypher in terms of expressing graph navigation queries. PGQL allows expressing structures such as chains, trees and stars. In terms of cyclic and acyclic structures, both the languages do not express such structures for graph navigation. Furthermore, chain shaped navigation patterns are shared between both the languages that can be expressed as CQT by using the $\mathsf{TA}(., |,^{-}, *)$ fragment of Tarski's algebra. In the following section, we further examine the extent to which chain shaped navigation patterns can be expressed in both languages.

6.8 Comparison of chain shaped queries in Cypher and PGQL

Chain shaped navigation patterns are used to search for paths that do not have branches or cycles between start and end nodes. Such navigation patterns can be expressed as CQT and UCQT by using the $\mathsf{TA}(., |, ^-, *)$ fragment of Tarski's algebra. Paths are sequences of edges where the total number of edges in the sequence represents the path's length. For example an expressions such as $\tau = a.b^-.c$ is used to search for the existence of paths that has three edges labeled as a, b^- and c respectively. For describing such expression, we employ concatenation and more edge labels can be specified in the expression for searching longer paths. Broadly there are two types of chain shaped expressions: *fixed-length* and *arbitrary-length*.

6.8.1 Fixed-length expressions

Expressions formed over elements of the set $L_{\mathcal{E}}$ involving a random but valid use of inverse, concatenation and union operations (but without the use of the Kleene star operator) results in searching for fixed-length paths in the graph database. The valid use of operations corresponds to defining expressions by using the $\mathsf{TA}(., |, ^-)$ fragment of Tarski's relation algebra. Standard results obtained from the algebra can be applied which state that applying concatenation operation over two expressions always increases the length of the path represented by the new expression[55, 74]. For example, the expression $a.b^-$ generates paths of length 2. We use the shorthand τ_n to represent expressions that generate paths of length n.

- Expressions in τ_1, such as $a, b^-, a|b$ etc., where $a, b \in L_{\mathcal{E}}$ will require searching for edges (size 1 path) in the graph database.

- Expressions in τ_i, where $i \geq 1$, such as $a.b$, $a|b^{-}.c$, $(a|b).c^{-}.(d^{-}|e)$, etc., where $a, b, c, d, e \in L_{\mathcal{E}}$, will require searching for paths containing i edges in the graph database.

The union operation can be seen as a choice, and when two expressions that represent paths of different sizes are combined over a union, the resulting expression has the same size as the highest-sized operand. In other words, the union operation does not increase the length of a path. For example, let $\tau_3 = a.b.a^{-}$ and $\tau_2 = b^{-}.a$ be two expressions formed by applying concatenation. Then applying union operation over τ_3 and τ_2 will result in an expression $\tau_3|\tau_2 = a.b.a^{-}|b^{-}.a$ where length of paths represented by the expression is never greater than 3. We present two theoretical results over the notion of Tarski's algebra representing fixed-length patterns, which are useful in constructing a rich set of chain shaped graph navigation queries.

Lemma 1 *Every expression written in the* $\mathsf{TA}(.,|,^{-})$ *fragment containing union can be rewritten as a union of union-free expressions.*

Consider an expression of the form $(a|b^{-}.a \mid a^{-}.a.a^{-}|b.b.b.b)$ such an expression can be rewritten as $\tau_1 \mid \tau_2 \mid \tau_3 \mid \tau_4$ where $\tau_1 = a$, $\tau_2 = b^{-}.a$, $\tau_3 = a^{-}.a.a^{-}$ and $\tau_4 = b.b.b.b$. Lemmas 1 have been presented in existing studies such as [73, 183, 206, 136, 128, 55].

Lemma 2 *Every union-free expression* τ_n *that represent fixed-length paths of length* n *are formed by applying concatenation operation* $n - 1$ *times.*

A proof for lemma 2 follows where we state that for all $i > 1$, $\tau_i = \{a.b : (a \in \tau_j) \wedge (b \in \tau_k) \wedge (j + k = i)\}$ where τ_j and τ_k are union free expressions. The base case involves constructing τ_2 expressions which can be done using concatenation operation once over two expression representing paths of length 1(from τ_1). Hence, we can see that to create expressions of size 2, we require applying the concatenation operation once. Next, as our inductive hypothesis, we assume that for some $i > 2$, and for all $l \leq i$ and $l \geq 2$, τ_l–expressions (of size l) are formed by applying concatenation operation $l - 1$ times.

For our inductive step, we need to prove that τ_{i+1}– expressions (of size $i + 1$) are formed by applying the concatenation operation i times. This can be shown by expanding the expression for τ_{i+1} as $\tau_{i+1} = \{a.b : (a \in \tau_j) \wedge (b \in \tau_k) \wedge (j + k = i + 1)\}$. We can see that expressions of size $i + 1$ can be formed by applying concatenation over two expressions of size j and k where $j + k = i + 1$. Since expressions of size j require $j - 1$ concatenations, and expressions of size k require $k - 1$ concatenations (as per the inductive hypothesis), the total number of concatenations required to form the expression of size $i + 1$ are $j - 1 + k - 1 + 1$. Since $j + k = i + 1$, we get the total number of steps to be equal to i which completes the proof.

Forming fixed-length expressions

We start by formulating fixed-length expressions of path length 1 which are expressions of the form a, b, a^- or b^-, or any finite union of a, b, a^-, and b^-. Navigation patterns labelled by expressions of size 2 are formed using the concatenation operation once, over expression of size 1. Similarly, expressions of size 3 can be formed by concatenating expressions of size 1 and 2. Possible combinations are shown in Table 6.12, where for an expression of size 3 of the form $\tau_1.\tau_2$, the first operand τ_1 is shown in the cells of the first column and the second operand τ_2 is shown in the cells of the first row. Expressions of path length 3 can contain a finite nesting of union operations but must contain at least one concatenation operation. Hence, expressions of path length 3 can be of the form $a.a.a, b.(a|a^-).b^-$, and $(a|b|a^-).b^-.(a|b|a^-)$.

Expressions of path lengths larger than 3 can be obtained by gradually increasing the number of concatenation operations used, along with a finite nesting of other operations

Table 6.12: Expressions of path length 3

τ_1 ↓ τ_2 →	$a.a$	$b.a^-$	$a^-.(a\|a^-)$	$b^-.(a\|b\|a^-)$	$(a\|a^-).b^-$	$(a\|b\|a^-).(a\|a^-)$
a	✓	✓	✓	✓	✓	✓
b	✓	✓	✓	✓	✓	✓
a^-	✓	✓	✓	✓	✓	✓
b^-	✓	✓	✓	✓	✓	✓
$a\|a^-$	✓	✓	✓	✓	✓	✓
$a\|b\|a^-$	✓	✓	✓	✓	✓	✓

like union while using brackets to specify the order of precedence. If a language supports expressing finite nesting of inverse over single edge label, union and concatenation, any two valid expressions to form larger expressions, then testing its support for expressions of path lengths 1, 2 and 3 is sufficient to deduce its support for all expressions of sizes larger than 3.

6.8.2 Arbitrary-length expressions

For constructing expressions of arbitrary-lengths, we apply the Kleene star operator over fixed-length expressions that is expressions formed by using the $\mathsf{TA}(.,|,^-,*)$ fragment of Tarski's relation algebra over the set of edge labels $L_\mathcal{E}$ that includes expressions of the form $(\tau_n)^*$, $(\tau_n)^*.(\tau_m)^*$, $(\tau_n)^*|(\tau_m)^*$, $((\tau_n)^*)^*$,$(\tau_n^*.\tau_m^*)^*$ and $(\tau_n^*|\tau_m^*)^*$ such that $n, m \geq 0$.

Forming arbitrary-length expressions

Kleene star operator's use results in an arbitrary-length expression that can be unrolled unto fixed-length expressions of increasing sizes. In our experiments, we test languages to support Kleene star applied over expressions of sizes 1, 2 and 3. A language that supports Kleene star over expressions of these sizes, as well as a finite nesting of inverse over single edge label, union and concatenation, will automatically support expressing Kleene star operation over expressions of sizes 4 and beyond. The complete list of fixed and arbitrary length navigation pattern are listed in Table 6.13.

Table 6.13: Expressions of fixed and arbitrary path length

Expression	**Example**
τ_1	$a, b, a\mid a^-, a\mid b\mid b^-, a\mid b\mid b^-\mid a^-$
τ_2	$a.b, b.(a\mid a^-), (a\mid b\mid b^-\mid a^-).(a\mid b)$
τ_3	$a.a.b, (a\mid b\mid b^-).(a\mid b\mid b^-\mid a^-).(a\mid b)$
$\tau_1\mid\tau_2$	$(a\mid a.b), (a\mid a^-)\mid(a\mid b\mid b^-\mid a^-).(a\mid b)$
τ_1^*	$a^*, (a\mid b)^*, (a\mid b^-)^*, (a^-\mid b^-)^*$
τ_2^*	$(a^-.(a\mid b))^*, (a.b)^*, ((a\mid b).(b\mid a^-))^*$
$(\tau_3)^*$	$((a\mid b).(a^-.a^-))^*$
$(\tau_1\mid\tau_2)^*$	$((a\mid b)\mid(a^-.a^-))^*$
$(\tau_1)^*\mid(\tau_1)^*$	$a^*\mid(a\mid b)^*$
$(\tau_1)^*\mid(\tau_2)^*$	$a^*\mid(a.(a\mid b))^*$
$(\tau_1)^*.(\tau_1)^*$	$(a^-)^*.(a\mid b)^*$
$(\tau_1)^*.(\tau_2)^*$	$(a^-)^*.(b^-.(a\mid b))^*$
$(\tau_1^*)^*$	$(a^*)^*$
$(\tau_2^*)^*$	$((a^-.(a\mid b))^*)^*$
$(\tau_1^*\mid\tau_2^*)^*$	$(a^*\mid(a.(a\mid b))^*)^*$
$(\tau_1^*.\tau_2^*)^*$	$(a^*.(a.(a\mid b))^*)^*$

6.8.3 Forming concrete queries

The chain shaped expressions from Table 6.13 can now be transformed, if supported, into Cypher and PGQL queries by expressing each expression as a concrete query. This leads to a comprehensive and objective comparison of the two languages, presented in the next section. To formulate the queries for this comparison, we used the graph schema for Airbnb dataset, presented in Figure 6.3, and populate two databases in Neo4j and Oracle on this schema with identical data from Airbnb. We then use the edge labels defined in the schema to create concrete queries based on expressions described in Table 6.13.

We reduce the total number of queries generated, but not the coverage of all query classes, using topological knowledge from the graph schema. For example, the Airbnb graph schema clearly shows that over any path in a graph database, an edge labelled by `WROTE` will always be followed by an edge labelled by `REVIEW_FOR`. Similarly, the edge label `KNOWS` will never be followed by `REVIEW_FOR`. Even though we can quickly formulate queries that require searching for the pattern `KNOWS.REVIEW_FOR`, the knowledge that this query will never yield a result set allows us to remove it from the comparison queries.

6.8.4 Comparison based on Fixed-Length Navigation patterns

Table F.1 in Appendix F shows the navigation patterns formed of fixed length expressions, expressed as queries in Cypher and PGQL. Out of a total *sixteen* fixed length expressions shown in Table F.1, Cypher can express *thirteen* navigation patterns while PGQL can only express *nine* navigation patterns. Both languages allow the use of *union* over edge labels to form fixed length expressions of path length 1, as shown in experiment `1a` and `1d`. The direction of the edges in experiments `1a` and `1d` are same. However, direction of edges in not same in experiment `1b` so for expressing such expressions we apply lemma 1 and convert the expression into a union of union free expressions. By doing so experiment `1b` can only be expressed in Cypher since it supports the `UNION` clause.

Navigation patterns representing paths of length 2 are presented as experiments `2a-2f`, where we can see that experiments `2c` and `2e` cannot be expressed in both the languages. We can apply lemma 1 to such expressions but in such a case the resulting expressions will not represents paths of length 2. For example, applying lemma 1 on expression `(OWNS|KNOWS⁻).(OWNS|WROTE)` results in three expressions where two expressions `OWNS` and `WROTE` represent paths of length 1 while the expression `KNOWS⁻.OWNS` represent paths of length 2. In other words, not all paths will be of length 2 where as in experiments `2a-2f` we are only interested in expressions that represent paths of length 2. The use of *inverse* and *concatenation* operations nested together is supported by both the languages, as shown by experiments `2a`. Experiments

3a-3c represent paths of length 3 where we can see that both the languages cannot express experiment 3b, this is for the same reason as per why we cannot express experiments 2c and 2e. The use of *inverse* and *union* is permissible in both the languages with the condition that the two edge labels involved in the *union* operation must be of same direction as shown in experiments 2d, 2f, 3a and 3c.

The navigation patterns in experiments 4a-4c contain expressions that represent paths of length 1 or 2 or both. This is because the *union* operation is applied over expressions of path length 1 and 2. As shown in Table F.1 only Cypher can express such navigation patterns. We can apply lemma 1 over these expressions and express them in Cypher by using the UNION clause. On the other hand PGQL cannot express any navigation patterns in experiments 4a-4c. Based on the experiments presented in Table F.1 we can see that in terms of fixed length chain structured navigation patterns Cypher is more expressive than PGQL. A primary reason for this is the absence of UNION clause is PGQL. Furthermore, *union* operation over edge labels is supported only in two cases: when the *inverse* operation has not been applied at all, or when the *inverse* operation has been applied over all the edge labels.

6.8.5 Comparison based on Arbitrary-Length Navigation Patterns

Table F.2 in Appendix F shows navigation patterns formed of arbitrary length expressions, expressed as queries in Cypher and PGQL. Out of a total of twenty seven queries Cypher can only express six queries where as PGQL can express thirteen queries. In Cypher Kleene star operation is supported when it is applied over expressions representing paths of length 1 as shown in experiments 5a, 5c and 5d. The expression in experiment 5a searches for path of lengths 0,3,6,... where edges in the paths are labeled by any combination of the labels WROTE, OWNS and HAS. These experiments can also be performed in PGQL. Experiment 5c cannot be performed in both the languages this is because the use of *union* operation with edges of opposite direction is not permitted in both the languages.

Experiments `6a-7c` involve applying Kleene star over fixed-length expressions representing paths of length 2 and 3. Out of 9 experiments 6 can be expressed in PGQL where as Cypher cannot express any of the experiments. A primary reason for this is due to the availability of `PATH` clause in PGQL to express fixed-length expressions and then the `MATCH` clause to apply Kleene star over expressions defined in the `PATH` clause. Experiments `6c, 6e, 7b` and `10b` cannot be expressed in both the languages because it involves the use of Kleene star over expressions formed by *union* of edge labels representing opposite direction edges. Experiments `8a, 8b` and `8c` cannot be expressed in both the languages. In these experiments Kleene star operation is applied over expressions representing paths of different length which is achieved by applying *union* operation over expressions of lengths 1 and 2. Since PGQL also supports the use of multiple `PATH` clauses so we tried to express experiment `8a` in PGQL as shown in Query 20.

QUERY 20: Query showing the use of multiple `PATH` clauses in PGQL

```
PATH p1 AS ()-[:WROTE]->()
PATH p2 AS ()-[:REVIEW_FOR]->()-[:HAS]->()
SELECT x.name, y.name
MATCH (x)-/:p1|p2*/->(y)
```

PGQL syntactically does not considers this query as incorrect however, the result set produced by this query only satisfies the fixed length expression in the path variable `p1`. This is a limitation of PGQL because it only permits application of Kleene star over only one path variable. Experiments `9a` and `9b` can be expressed in both the languages as Cypher allows the use of Kleene star operation over expressions representing paths of length 1 and PGQL allows the use of `PATH` and `MATCH` clause to express such patterns. However, experiments `10a` and `10c` cannot be expressed in Cypher while PGQL can express such expressions by using the `PATH` clause. Experiment `11` can be expressed only in Cypher and not in PGQL this is because Cypher allows the use of lemma 1 and such an expression can be easily expressed in Cypher by using the `UNION` clause.

Experiment `12` cannot be expressed in both the languages as PGQL does not support the `UNION` clause whereas Cypher does not allow the use of Kleene star over expressions of path length greater than 1. Experiments `13,14,15` and `16` represent expressions formed by applying Kleene star over expressions of arbitrary lengths, both Cypher and PGQL are unable to express such patterns. Overall, for graph navigation queries PGQL is more expressive than Cypher, a primary reason for this the presence of `PATH` clause that enables the query language to express a variety of navigation patterns.

6.9 Conclusion

This article proposes an extension of conjunctive queries and union of conjunctive queries with Tarski's algebra. We have proposed novel formalisms of CQT and UCQT that provide a formal basis to compare, integrate and model practical graph query languages. In order to objectively compare practical graph query languages, a framework is proposed that integrates the extended formalisms with common graph query patterns to generate a comprehensive set of benchmark queries. This process is the basis of a comparative study of two practical graph query languages Cypher and PGQL. Our analysis shows that Cypher is more expressive than PGQL for graph pattern matching queries due to the presence of explicit `UNION` clause. For graph navigation queries PGQL is more expressive than Cypher due the presence of the `PATH` clause. In PGQL the use of `PATH` clause along with the `MATCH` clause enables the application of Kleene star over complex structures such as chains, trees, stars and star chains. Cypher on the other hand does not provide such functionality and has limited expressiveness concerning graph navigation queries. Our study also shows that with respect to graph navigation queries, cyclic and acyclic graph query patterns cannot be expressed in both the languages.

Such graph query patterns are important because they may have use in specialised graph databases used in fields like chemistry, biology and astronomy. This study's shape-based analysis can help identify common and exclusive characteristics for other currently available practical graph query languages such as SPARQL, GSQL, SQL/PG and Gremlin. Furthermore, the extended formalism and the integrated framework can be utilised to model future graph query languages; therefore, they serve as a basis for upcoming standards like IEC/ISO 39075.

6.9.1 Limitations

We have considered Tarski's algebra for comparing the graph navigation features of query languages. However, formalisms such as Tarski's algebra are purely navigational and do not support comparisons of data values in path expressions [34]. We intend to include formalisms used in query languages such as GXPath, *regular expressions with memory (*`REM`*), walk logic* and *register logic* [65, 34, 73, 165, 72] along with Tarski's algebra for graph navigation in future studies. Furthermore, we have also not considered more expressive formalisms such as ECRPQ that enable path comparisons and context-free paths in our study. This is because features provided by such formalisms are not yet present in all practical query languages. Moreover, returning paths can be problematic if cycles exist in graph databases as infinite paths can be returned if arbitrary path semantics are used. Another important concept of the algorithmic complexity associated with graph query languages and the extended formalism is not considered in this study, and we see it as future work.

We have only considered operations such as natural join, selection, projection and union. Other operations such as *difference* is computationally more expensive [32]. As a result difference clause is not implemented yet in Cypher and PGQL but similar functionality can be simulated by using the `NOT EXISTS` clause along with `WHERE` clause. Cypher also allows the use of `NOT` keyword to simulate difference operation with a restriction that the pattern specified in the `WHERE NOT` clause must represent an eulerian path [171]. Cypher and PGQL do not have such restrictions in the `WHERE NOT EXISTS` clause.

The use of `EXISTS` and `NOT EXISTS` clause represents semi join and anti semi join respectively, that correspond to the *semi join algebra* and its equivalent formalism *guarded fragments* (GF) of first order logic [207, 63, 208, 209]. We do not consider the study of guarded fragment of first order logic in this study. Furthermore, we have also not considered other relational algebra operations such as *outer join* and *aggregate* functions. Outer join clause `OPTIONAL MATCH` is only present in Cypher while aggregate functions are used after the result set is returned by a pattern matching algorithm.

6.9.2 Future work

The use of conjunctive queries based formalism for graph pattern matching reveals some similarities between query languages for graph and relational databases. Concerning difference operation, graph query languages such as Cypher and PGQL implement this operation by using semi-join algebra and equivalent guarded fragment of first-order logic. For graph navigation queries authors in [74, 55, 14] also suggest similarities between Tarski's algebra and semi-join algebra. Semi join algebra can be particularly useful in query optimisation. Hence, we consider the study of graph query languages based on semi-join algebra as future work.

We also intend to work on graph schema driven template query generation tool and the formalisms identified in this research will assist in creating the tool that can be used to test the expressiveness of other query languages such as SPARQL, Gremlin, SQL/PG and GSQL. The automatic generation of query language adapters is another interesting future direction of this work, which will help the community's shared goal of high interoperability between available graph database technologies.

Chapter 7

Introduction to Manuscript 3

Contemporary graph databases are either schema-less or schema-optional to support frequent changes in the structure of data found in domains requiring high flexibility. However, the lack of structure impacts on data transformation and loading operations from heterogeneous sources into graph databases. We present a formal algebra FLASc for specifying and generating graph schema for labeled property graph databases. We formally define FLASc and demonstrate the use of FLASc generated graph schemas to systematically transform and load data-sets related to domains of cyber physical systems, big data analytics and tourism. Findings from three disparate case studies show that FLASc-generated schemas assist in enforcing integrity constraints that reduces the chance of data corruption, hence assuring data consistency and integrity. The two case studies related to cyber physical systems and tourism have been adopted from our two conference publications presented as Chapters A, B, 3 and 4. The case study related to big data analytics has been adopted from [86].

Chapter 8

FLASc: A Formal Algebra for Labeled Property Graph Schema (Manuscript 3)

8.1 Introduction

Labeled property graph database henceforth graph database are storage systems that allow modeling of real-world entities as nodes and relationships between entities as edges [196]. Nodes and edges in a graph database have associated labels. Data is stored inside nodes and edges as properties that exist in the form of key-value pairs [32, 23]. Graph databases are efficient in storing and managing highly interconnected data-sets related to domains such as transportation networks, social media, bioinformatics, chemistry and astronomy [23, 133, 32, 31, 5]. Graph databases suit big data applications as they provide a better alternative for modeling and handling complex information [7, 8]. Graph databases are more efficient than relational databases for extracting information from highly connected data-sets. Specifically, querying graph databases does not require the expensive join operation [14]. Furthermore, unlike relational databases, the same graph database can be used for online transaction processing (OLTP) as well as online analytical processing (OLAP) tasks [144, 210, 211]. The interconnections between data represent the underlying meaning of a graph data-set. Therefore, maintaining data consistency and integrity is vital in graph databases [23, 130].

Obtaining a database that is sound and consistent requires embracing good database modeling principles [80]. In contrast to relational databases, modeling principles for graph databases are ad-hoc and not well-grounded [144]. Contemporary graph databases lack mechanisms to ensure data consistency and integrity, especially when the data being stored comes from multiple heterogeneous sources [81]. A primary reason is that graph databases are either schema-less or schema-optional [81]. A schema represents the overall structure of a data-set and assists in understanding data semantics [44]. Furthermore, schemas aid in defining integrity constraints that are sets of rules for ensuring consistency and integrity in the database that conforms to the schema [212, 135]. The lack of schema and integrity constraints poses significant challenges in ensuring data consistency and integrity [213], in performing advanced analytics and achieving data interoperability [214], and for data integration, query optimization and processing [215].

Traditional database modeling consists of three stages *conceptual, logical* and *physical modeling* [80]. In graph databases, the conceptual modeling stage represents gathering requirements of a given problem domain that are then used for defining entities and relationships between them. The logical modeling stage represents the enforcement of integrity constraints, including mandatory, optional and unique properties associated with entities and relationships defined in the conceptual modeling stage. The physical modeling stage represents the realization of graph schema formulated at the conceptual and logical modeling stage into database creation scripts.

An open problem in graph database design is that practitioners do not have proper guidelines for designing the conceptual models [44, 80] that can facilitate systematic transformation and loading of data from heterogeneous sources into graph databases. Conceptual modeling stage is not used in the majority of graph database solutions [216, 217]. Graph databases lack abstraction tools [23] and most current research is primarily focused on logical and physical modeling [81, 45]. These observations lead us to the following research questions:

RQ1. What are the key strengths and limitations of existing approaches used for modeling graph databases?

RQ2. What mechanisms can be designed to formulate conceptual and logical graph schemas for labeled property graph databases?

RQ3. In order to ensure data consistency, how can the graph schema generated by RQ2 be used to systematically import data from heterogeneous sources into a labeled property graph database?

- RQ3.1 How can the Extract-Transfrom-Load design pattern be extended in order to support loading data-sets for heterogeneous sources into graph database?

We answered these research questions using a mixed-methods research methodology [218]. Firstly, for addressing RQ1 a literature review was carried out to identify existing evidence and gaps in the literature related to the research question. We addressed RQ2 by proposing an algebra FLASc which is based on conceptual graphs introduced by Sowa [50, 48, 49]. The two operators of `JOIN` and `DETACH` provided by FLASc serve as mechanisms for formulating conceptual graph schemas which are further extended to logical graph schemas. For addressing RQ3 and RQ3.1, we illustrate the integration of FLASc with the well known Extract-Transform-Load (ETL) design pattern. The graph schemas generated by FLASc can be used to enforce integrity constraints and assist in the systematic generation of database creation scripts hence ensuring data consistency. To demonstrate the utility of our approach we consider three distinct case studies related to industrial cyber-physical systems [16], big data analytics [86, 219] and tourism [87, 15]. We generate graph schemas for the heterogeneous data-sets provided in the three case studies and produce database creation scripts in $Cypher$ using the FLASc integrated ETL design pattern. The critical contributions of this work include:

1. We formulate FLASc a formal algebra for constructing a labeled property graph schema that can capture data semantics of any given problem domain. We define operators of FLASc that assist in constructing a graph schema.

2. We demonstrate the use of graph schemas formulated via FLASc to enforce integrity constraints that ensure data consistency in contemporary labeled property graph databases such as Neo4j.

3. We illustrate how FLASc can be integrated with the Extract-Transform-Load process for loading data-sets from heterogeneous sources into Neo4j.

Two case studies related to tourism and cyber physical systems, presented in Sections 8.5.2 and 8.5.4, have been adopted from previously published research [16], [220] and [15] respectively. The formalism for labeled property graph schemas presented in [15] and [220] is foundational for designing our algebra FLASc. The work presented in this research paper empowers users of FLASc to design robust graph schemas for labeled property graph databases.

The rest of the chapter is organized as follows. Section 8.2 presents background information and related work. The gaps identified in Section 8.2 are used to build FLASc which is presented in Section 8.3. In Section 8.4 we illustrate how the conceptual and logical graph schema formulated using FLASc can be used to enforce several integrity constraints in Neo4j graph database. In Section 8.5 we present the integration of FLASc with ETL design pattern and experimentally demonstrate its use for data transformation and loading of heterogeneous data-sets into Neo4j graph database. Finally, in Section 8.6 we summarize the key contributions and future directions of this work.

8.2 Background and Related Work

This section enables us to address RQ1. We present a brief survey of the existing approaches that have been proposed for modeling graph databases.

8.2.1 Graph database design and modeling

Graph databases use graphs consisting of nodes and edges as elementary data structures for modeling any problem domain [133, 32, 23]. All graph databases use slight variations of the basic graph data structure. For example, graph databases proposed in academia such as GOOD [12], Gram [90], GraphDB [94], GDM [113], [96] and [221] use directed labeled graph data structures. Graph database such as hyper log [222, 97] use hyper node and hyper edge based graphs. Commercial graph databases such as Resource Description Framework (RDF) by W3C [114] use directed labeled graphs while Neo4j [51], Oracle [52] use directed, labeled and attributed graphs which are also known as property graphs [27]. There are three main stages of modeling a graph database: *conceptual, logical* and *physical.*

Conceptual modeling

Conceptual modeling represents the initial stage of modeling a graph database. In this stage, knowledge is collected in the form of requirements and specifications related to a problem domain. Using graphs for representing knowledge was first proposed by Sowa [50, 48, 47, 49]. Subsequent works [130, 131, 132] also propose the use of graphs to represent knowledge at the conceptual modeling stage. Graphs provide a natural and intuitive interface for understanding the semantics of data [50, 80]. Knowing the semantics of data is vital for understanding the overall structure of the database [44] that aids in creating, modifying and retrieving data. Schemas created at the conceptual modeling stage provide a level of abstraction that aids in the natural modeling of data [133]. Conceptual graph schemas are used to define entities that belong to the database and relationships between those entities [80]. Moreover, determining nodes, edges, and the direction of edges are vital for conceptual modeling [39].

Logical modeling

Logical modeling is used to define integrity constraints on entities and relations of conceptual graph schema. Integrity constraints serve as mechanisms to ensure data consistency and integrity. They are broadly classified into two categories: *graph entity integrity* and *semantic constraints* [134]. Graph entity integrity constraints are related to basic database design principles. These include constraints such as node/edge property uniqueness [23, 45, 133, 134, 46], label uniqueness [23, 44, 133, 134, 45], property data type [44, 46] and mandatory property constraints [135, 44]. Enforcing semantic constraints require knowledge of the problem domain captured in the conceptual graph schema. These constraints are used to guarantee the conformity of graph database with domain specific rules and require intervention from end users. These include edge pattern [46, 134, 135, 81, 45], graph pattern [46, 134, 133, 135] and path pattern constraints [46]. Other constraints discussed in literature include type checking [23, 133, 135], node/edge property value constraints [81], cardinality constraints [44, 46, 133, 134, 81, 84, 83] and functional dependencies [23, 44, 133, 137, 138, 139].

Physical modeling

Physical modeling represents the realization of the graph schema designed during conceptual and logical modeling into actual database [140]. There are two approaches discussed in literature for physical modeling: *integrated* and *layered* [83]. In the integrated approach, changes are made in the source code of the database system. Database creation scripts are created and directly deployed on the database platform. In the layered approach, APIs specific to the database platform are used to create an additional layer that communicates with the database. This consist of wrappers written in programming languages such as Java, Python that contains database creation scripts and logic to enforce the integrity constraints.

Integration of logical and physical modeling

There exist many studies to support the integration of logical and physical modeling aspects of graph databases. For instance, [134] follow a layered approach and propose the construction of a wrapper that can be used to enforce integrity constraints, including graph and path pattern constraints over Neo4j graph database. An integrated approach to extend the source code of OrientDB to support the enforcement of integrity constraints, including uniqueness, key, cardinality, and edge degree constraints, has been studied in [81]. Similarly, the extension of Cypher query language to support additional integrity constraints such as uniqueness, node property, edges pattern, and mandatory properties is presented in [45, 82]. A layered approach to demonstrate the enforcement of uniqueness integrity constraint on two different graph databases Neo4j and Apache Tinkerpop, is proposed in [83]. The use of integrated and layered approach together to perform graph database manipulation operations on Neo4j graph database is proposed in [46]. Authors in [141] propose the model-driven engineering based approach for converting and loading of UML diagrams into tinkerpop blueprints[1].

[1]`https://github.com/tinkerpop/blueprints`

8.2.2 Gaps in Current Literature

Several studies have been proposed in the last decade that address the problem of modeling graph databases. These studies mainly focus on the integration of logical and physical modeling aspects. A primary reason of this due to graph data models such as resource description framework (RDF) [17], labeled property graphs [27] and creation of query languages such as SPARQL [163], Cypher (Neo4j) [51], Gremlin (Apache) [111], PGQL (Oracle) [52] and GSQL (TigerGraph) [223] to support data modeling and retrieval. More recently, projects such as ISO/IEC 39075[2], openCypher [164] and Linked Data Benchmark Council (LDBC) [4] have been proposed for developing a standard query language for the labeled property graph data model. Most of these studies focus on extending the existing query languages to support logical and physical modeling while conceptual modeling is done in an ad-hoc manner. Authors in [134, 145, 142] present a formal approach for logical modeling of graph databases. However, physical modeling in these research papers are not discussed in detail [84] and application of the proposed formalisms on real-world data-sets are considered future work.

To obtain a robust graph database that captures semantics of the problem domain conceptual modeling stage is vital. A sound conceptual graph schema ensures that logical and physical modeling stages are also robust [143]. The graph data modeling approaches proposed so far do not provide the means to create robust conceptual graph schemas.

Authors in [144, 145, 141] propose the use of existing visual modeling tools such as entity relationships diagrams (ERD) and unified modeling language (UML) for creating conceptual and logical graph schemas. The schemas generated by such visual models are based on node-labeled graphs [15] where only the nodes can have properties associated with them. On the other hand FLASc directly supports LPG schemas that have labels

[2]`https://www.iso.org/standard/76120.html`

and properties associated with nodes and edges [15, 220]. Modeling tools such as ERD and UML are generic and while they can be used to model LPG schema, they do not capture subtleties like edge labels and attributes without carefully considered extensions. As an example, edge-related information has to be stored as additional nodes in an ERD. Both UML and ERD are semi-formal modeling tools whereas FLASc provides a formal basis for LPG schemas. This opens up the opportunity to define a FLASc-driven schema-generation language based on formal languages such as conjuntive queries and first order logic. However, extensions of FLASc are not in the scope of this research paper.

Therefore, we present FLASc a simple yet sturdy formal tool that assists in the formulation of robust conceptual graph schemas which is an advancement over existing studies in graph database modeling. The majority of integrity constraints presented in the existing studies can be specified in graph schemas generated by FLASc. Furthermore, syntax and semantics of FLASc presented in this study assist in its implementation at the physical modeling stage. FLASc assists in the integration of conceptual, logical and physical modeling stages which currently is lacking in graph database research.

8.3 FLASc: Formal Algebra for conceptual and logical graph Schema

This section addresses RQ2, we present the formal algebra FLASc that assists in formulating conceptual and logical graph schemas for labeled property graph databases. We use the concepts from Sowa's conceptual graphs identified in Section 8.2.1 to propose the operations of FLASc. We use a formal approach for constructing FLASc which assures the robustness of its design [224, 225]. FLASc has sound mathematical basis that enables a user to precisely define: *(i)* connections between entities of a graph database (intensional information) and *(ii)* properties associated with entities and relations in a graph database (extensional information) [47, 48, 49, 50].

We consider a data-set from Airbnb [15, 220] as our first case study related to the tourism domain that assists in illustrating various definitions and concepts of FLASc. This data-set consists of three CSV files that contain information related to property listings, reviews and calendar data. This data-set is highly interconnected, making it a prime candidate for graph database design and implementation.

8.3.1 Basic terminology

Definition 16 (Directed Multigraph) *A directed multigraph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ is a tuple where $\mathcal{N}$ is a set of nodes and $\mathcal{E}$ is a set of edges. Two associated functions, $s : \mathcal{E} \rightarrow \mathcal{N}$ and $t : \mathcal{E} \rightarrow \mathcal{N}$, map each edge to its source and target nodes, respectively.*

We use the shorthand $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ to represent a directed multigraph.

Each edge in a *directed* multigraph has unique source and target nodes. Edges with same source and target nodes are allowed (hence the term *multi*graph. We use the short hand $n_i \rightarrow n_j$ to represent an edge e_k where $s(e_k) = n_i$ and $t(e_k) = n_j$.

Graph can contain labels over nodes and edges. Given a set of node labels $L_{\mathcal{N}}$ and a set of edge labels $L_{\mathcal{E}}$ such that $L_{\mathcal{N}} \cap L_{\mathcal{E}} = \varnothing$. A labeling is simply a map $f : S_1 \rightarrow S_2$ such that for every element $a \in S_1$, there is a unique element $f(a) \in S_2$. We can define an *edge*- labeled graph as follows.

Definition 17 (Edge-Labeled Graph) *A graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \xi)$ is called an edge-labeled graph if there exists a labeling $\xi : \mathcal{E} \rightarrow L_{\mathcal{E}}$ which maps all edges to labels in a set of edge labels $L_{\mathcal{E}}$. We use the short-hand $e_k = n_i \xrightarrow{l} n_j$ for any $e_k \in \mathcal{E}$ and $\xi(e_k) = l$.*

Similarly, we can define a node labeled graph.

Definition 18 (Node-Labeled Graph) *A graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \eta)$ is called a node-labeled graph if there exists a labeling $\eta : \mathcal{N} \to L_{\mathcal{N}}$ which maps all nodes to labels in a set of node labels $L_{\mathcal{N}}$ for any $n_i \in \mathcal{N}$ and $l \in L_{\mathcal{N}}$ if l is mapped to n_i then $\eta(n_i) = l$.*

8.3.2 Conceptual graph schema

A conceptual graph schema is used to capture intensional information. Conceptual modeling is easier for the user to understand and contribute. Therefore, a conceptual graph schema must be closer to the semantics of natural languages like English. It must reflect real-world entities, and relations that are not directly represented by the conceptual graph schema must be accessible to infer [48, 132]. As discussed in [15] to define relationships, we use the `(subject,predicate,object)` format from semantics web [151] where the subject can be a noun, the predicate can be a verb, and an object can also be a noun.

Definition 19 (Conceptual Graph Schema) *Given a set of node labels $L_{\mathcal{N}}$ and a set of edge labels $L_{\mathcal{E}}$, conceptual graph schema $\mathcal{G}_s$ is a tuple $(\mathcal{N}_s, \mathcal{E}_s, \eta_s, \xi_s, L_{\mathcal{N}}, L_{\mathcal{E}})$ where,*

- *$\mathcal{N}_s$ is a finite set of nodes and $\mathcal{E}_s$ is a finite set of edges of the graph schema.*
- *$(\mathcal{N}_s, \mathcal{E}_s)$ is a directed multigraph.*
- *$\eta_s : \mathcal{N}_s \to L_{\mathcal{N}}$ is a node labeling function and $\xi_s : \mathcal{E}_s \to L_{\mathcal{E}}$ is an edge labeling function.*

We use the shorthand notation $\mathcal{G}_s = (\mathcal{N}_s, \mathcal{E}_s, \eta_s, \xi_s)$ to represent the conceptual graph schema.

Example 28 *The conceptual graph schema generated for Airbnb case study as discussed in [15] is presented in Figure 8.1. The graph schema consists of six labels including `review`, `user`, `host` and `listing` and four edge labels* *`wrote`, `review_for`, `has` and `owns`. In the Airbnb data-set [87] a person using Airbnb service can write a review for a listing that was recently visited by him or her. A conceptual graph schema in such a scenario consists of entities such as `user` and `review`. Relationships can be of the form `(users,wrote,review)` where `users` is the subject, `wrote` is the verb and `review` is the object.*

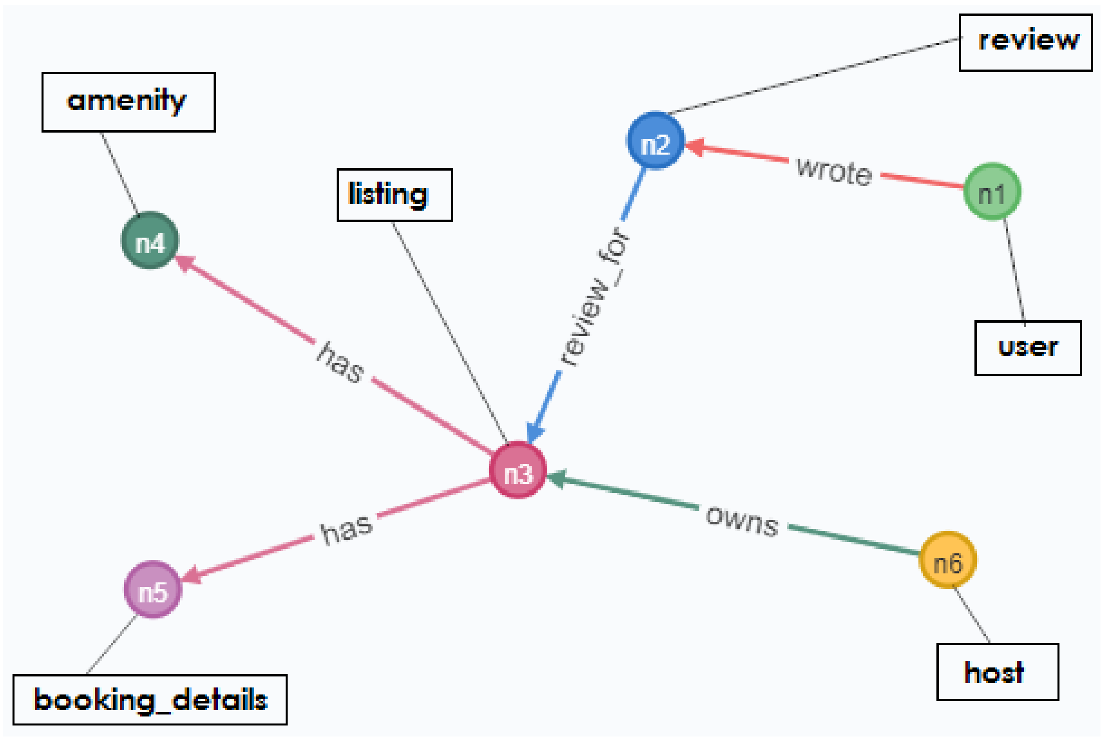


Figure 8.1: Conceptual graph schema generated for Airbnb case study

Basic conceptual graph schema

Basic conceptual graph schemas are restricted form of conceptual graph schemas. They serve as building blocks for formulating conceptual graph schemas. Formally basic conceptual graph schemas are defined as follows.

Definition 20 (Basic Conceptual Graph Schema) *Given sets of node and edge labels $L_{\mathcal{N}}$ and $L_{\mathcal{E}}$, a basic conceptual graph schema $\mathcal{G}_b$ is a tuple $(\mathcal{N}_b, \mathcal{E}_b, \eta_b, \xi_b)$ where*

- *$\mathcal{N}_b = \{n_i, n_j\}$ is a set of two nodes.*
- *$\mathcal{E}_b = \{e_k\} \cup \varnothing$ can either be a singleton set or an empty set.*
- *$(\mathcal{N}_b, \mathcal{E}_b)$ is a restricted from of directed multigraph supporting only one directed edge between two nodes.*
- *$\eta_b : \mathcal{N}_b \to L_{\mathcal{N}}$ is a node labeling function and $\xi_b : \mathcal{E}_b \to L_{\mathcal{E}}$ is an edge labeling function.*

Example 29 *The Airbnb data-set consists of several basic conceptual graph schemas including $\mathcal{G}_{b1} = \big(\{n_1, n_2\}, \{n_1 \xrightarrow{\texttt{wrote}} n_2\}, \eta_1, \xi_1\big)$ such that $\eta_{b1}(n_1) = \texttt{user}$, $\eta_1(n_2) = \texttt{review}$ and $\xi_1(n_1 \xrightarrow{\texttt{wrote}} n_2) = \texttt{wrote}$. Similarly $\mathcal{G}_{b2} = \big(\{n_2, n_3\}, \{n_2 \xrightarrow{\texttt{review\_for}} n_3\}, \eta_2, \xi_2\big)$ such that $\eta_2(n_2) = \texttt{review}$, $\eta_2(n_3) = \texttt{listing}$ and $\xi_2(n_2 \xrightarrow{review\_for} n_3)$. The basic conceptual graph schema is used to represent the intensional information that a review was written by a user and review was written for a listing.*

Basic conceptual graph schemas serve as a starting point for a database designer and assist in conceptual modeling. A basic conceptual graph schema can contain nodes that are not connected to one another by an edge. A designer can create separate basic conceptual graph schemas for each requirement and/or use case. We define FLASc to create conceptual graph schemas from basic conceptual graph schemas.

Syntax and semantics of FLASc

An algebra consists of sets, constants that belong to the sets and some functions or operators that are used to manipulate data stored inside the sets [169]. Our algebra FLASc is defined as follows:

Definition 21 (FLASc) *An algebra defined over a finite set of basic conceptual graph schemas $\mathcal{G}_B$, is a tuple $\langle \mathcal{G}_B, \mathcal{G}, \mathcal{F} \rangle$ where:*

- *$\mathcal{G}$ is the set of all conceptual graph schemas over $\mathcal{G}_B$, with $\mathcal{G}_B \subset \mathcal{G}$.*
- *$\mathcal{F}$ is a set containing three operators:*
  1. *`JOIN`: $\mathcal{G} \times \mathcal{G} \to \mathcal{G}$ is a binary operator such that if $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}$ then `JOIN` $(\mathcal{G}_1, \mathcal{G}_2)$ is a conceptual graph schema formed by the union of two conceptual graph schemas. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1)$ where $L_{\mathcal{N}_1}$ is a set of node labels and $L_{\mathcal{E}_1}$ is a set of edge labels associated with $\mathcal{G}_1$. Let $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2)$ where $L_{\mathcal{N}_2}$ is a set of node labels and $L_{\mathcal{E}_2}$ is a set of edge labels associated with $\mathcal{G}_2$. Then `JOIN`$(\mathcal{G}_1, \mathcal{G}_2) = \big((\mathcal{N}_1 \cup \mathcal{N}_2), (\mathcal{E}_1 \cup \mathcal{E}_2), \eta_3, \xi_3\big)$ such that $\eta_3 : (\mathcal{N}_1 \cup \mathcal{N}_2) \to (L_{\mathcal{N}_1} \cup L_{\mathcal{N}_2})$ and $\xi_3 : (\mathcal{E}_1 \cup \mathcal{E}_2) \to (L_{\mathcal{E}_1} \cup L_{\mathcal{E}_2})$. The resulting conceptual graph obtained from `JOIN` $(\mathcal{G}_1, \mathcal{G}_2)$ also has two associated functions $s : (\mathcal{E}_1 \cup \mathcal{E}_2) \to (\mathcal{N}_1 \cup \mathcal{N}_2)$ and $t : (\mathcal{E}_1 \cup \mathcal{E}_2) \to (\mathcal{N}_1 \cup \mathcal{N}_2)$.*
  2. *`DETACH`: $\mathcal{G} \times \mathcal{G} \to \mathcal{G}$ is a binary operator such that if $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}$ then `DETACH` $(\mathcal{G}_1, \mathcal{G}_2)$ is a conceptual graph schema formed by applying ring sum over the edge sets of $\mathcal{G}_1$ and $\mathcal{G}_2$. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1)$ where $L_{\mathcal{N}_1}$ is a set of node labels and $L_{\mathcal{E}_1}$ is a set of edge labels associated with $\mathcal{G}_1$. Let $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2)$ where $L_{\mathcal{N}_2}$ is a set of node labels and $L_{\mathcal{E}_2}$ is a set of edge labels associated with $\mathcal{G}_2$. The resultant conceptual graph schema consists of all the nodes present in graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ that is $\mathcal{N}_1 \cup \mathcal{N}_2$. While the ring sum operator is only applied over the edge sets of two graphs that is $(\mathcal{E}_1 \oplus \mathcal{E}_2) = (\mathcal{E}_1 \cup \mathcal{E}_2) - (\mathcal{E}_1 \cap \mathcal{E}_2)$. `DETACH`$(\mathcal{G}_1, \mathcal{G}_2) = \big((\mathcal{N}_1 \cup \mathcal{N}_2), (\mathcal{E}_1 \oplus \mathcal{E}_2), \eta_3, \xi_3\big)$ such that $\eta_3 : (\mathcal{N}_1 \cup \mathcal{N}_2) \to (L_{\mathcal{N}_1} \cup L_{\mathcal{N}_2})$ and $\xi_3 : (\mathcal{E}_1 \oplus \mathcal{E}_2) \to (L_{\mathcal{E}_1} \oplus L_{\mathcal{E}_2})$. The resulting conceptual graph obtained from `DETACH` $(\mathcal{G}_1, \mathcal{G}_2)$ also has two associated functions $s : (\mathcal{E}_1 \oplus \mathcal{E}_2) \to (\mathcal{N}_1 \cup \mathcal{N}_2)$ and $t : (\mathcal{E}_1 \oplus \mathcal{E}_2) \to (\mathcal{N}_1 \cup \mathcal{N}_2)$.*
  3. *`DELETE_NODE`: $\mathcal{G} \times \mathcal{G} \to \mathcal{G}$ is a binary operator such that if $\mathcal{G}_1, \mathcal{G}_d \in \mathcal{G}$ then `DELETE_NODE`$(\mathcal{G}_1, \mathcal{G}_d)$ is a conceptual graph schema formed by applying ring sum over the node sets of $\mathcal{G}_1$ and $\mathcal{G}_d$. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1)$ where*

$L_{\mathcal{N}_1}$ is a set of node labels and $L_{\mathcal{E}_1}$ is a set of edge labels associated with $\mathcal{G}_1$. Let $\mathcal{G}_d = (\mathcal{N}_d, \emptyset, \eta_d)$ is a node labeled graph where $L_{\mathcal{N}_d}$ is a set of node labels associated with $\mathcal{G}_d$. Then the resultant conceptual graph schema consist of nodes that are formed by applying the ring sum over the node sets of two graphs that is $(\mathcal{N}_1 \oplus \mathcal{N}_d) = (\mathcal{N}_1 \cup \mathcal{N}_d) - (\mathcal{N}_1 \cap \mathcal{N}_d)$. The set of edges in the conceptual graph schema `DELETE_NODE`$(\mathcal{G}_1, \mathcal{G}_d)$ is equal to the set of edges in $\mathcal{G}_1$ that is $(\mathcal{E}_1 \cup \emptyset)$. The set of nodes in conceptual graph schema `DELETE_NODE`$(\mathcal{G}_1, \mathcal{G}_d)$ belongs to the set $(\mathcal{N}_1 \oplus \mathcal{N}_d)$ such that $\exists n_i \in \mathcal{N}_1$ where $\forall e \in \mathcal{E}_1, s(e) \neq n_i, t(e) \neq n_i$. Furthermore, $\forall n_d \in \mathcal{N}_d, \eta_1(n_i) = \eta_d(n_d)$. The resulting conceptual graph schema `DELETE_NODE`$(\mathcal{G}_1, \mathcal{G}_d)$ $= \big((\mathcal{N}_1 \oplus \mathcal{N}_d), (\mathcal{E}_1 \cup \emptyset), \eta_2, \xi_2\big)$ such that $\eta_2 : (\mathcal{N}_1 \oplus \mathcal{N}_d) \rightarrow (L_{\mathcal{N}_1} \oplus L_{\mathcal{N}_d})$ and $\xi_2 : \mathcal{E}_1 \rightarrow L_{\mathcal{E}_1}$. The conceptual graph schema also consists of two associated functions where $s : \mathcal{E}_1 \rightarrow (\mathcal{N}_1 \oplus \mathcal{N}_d)$ and $t : \mathcal{E}_1 \rightarrow (\mathcal{N}_1 \oplus \mathcal{N}_d)$.

FLASc provides `JOIN`, `DETACH` and `DELETE_NODE` operators over basic conceptual graph schemas to formulate composite conceptual graph schemas. We can now discuss the semantics of these three operators and provide some examples.

`JOIN` is used to combine together two or more conceptual graph schemas. We follow the similar notion of join compatible mapping as discussed in [32, 171, 172]. Two conceptual graph schemas are join compatible if they share common nodes. That is $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1)$ and $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2)$ are join compatible if $\exists e_i \in \mathcal{E}_1$ and $\exists e_j \in \mathcal{E}_2$ such that *either $s(e_i) = t(e_j)$ or $t(e_i) = s(e_j)$ or $s(e_i) = s(e_j)$ or $t(e_i) = t(e_j)$*. Furthermore, if $s(e_i)$ *or* $t(e_i) = n_i$ and $s(e_j)$ *or* $t(e_j) = n_j$ then $\eta_1(n_i) = \eta_2(n_j)$.

Example 30 *The basic conceptual graph schemas presented in Example 29 are join compatible because both graphs share a common node n_2 that have the node label `review`.*

Figure 8.2 shows that applying the `JOIN` operator over basic conceptual graph *schemas $\mathcal{G}_{b1} = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1)$ and $\mathcal{G}_{b2} = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2)$ creates a conceptual graph schema $\mathcal{G}_{b3}$ = `JOIN` $(\mathcal{G}_{b1}, \mathcal{G}_{b2})$. Graphs $\mathcal{G}_{b1}$ and $\mathcal{G}_{b2}$ are join compatible because the target node of edge $e_1 \in \mathcal{E}_1$ that is $t(e_1)$ and source node of edge $e_2 \in \mathcal{E}_2$ that is $s(e_2)$ are same. Moreover the node labels associated with these two nodes are also same that is $\eta_1(t(e_1)) = \eta_2(s(e_2))$ = `review`.*

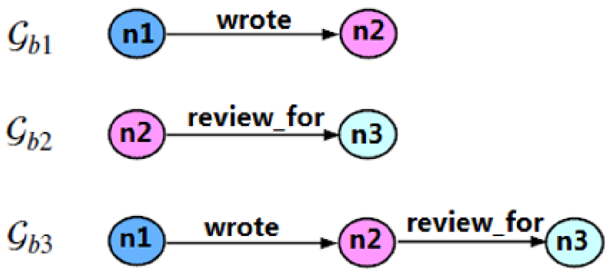


Figure 8.2: The application of `JOIN` operator to connect two conceptual graph schemas

Two join compatible conceptual graphs share common nodes. This assists in connecting smaller graphs. When two conceptual graph schemas are not join compatible, then application of the `JOIN` operator creates a union of two disconnected conceptual graph schemas.

`DETACH` is used to delete edges from a conceptual graph schema. This operator is useful if a database designer wishes to delete existing relationships in a conceptual graph schema. The graph produced after applying a `DETACH` operator over two conceptual graph schemas contain nodes from both the graphs. While edges of the new conceptual graph schema are calculated by applying the ring sum operator over the edges of conceptual graph schemas that provided as input to the `DETACH` operator. Applying the `DETACH` operator over two conceptual graph schemas $\mathcal{G}_{b1} = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1)$ and $\mathcal{G}_{b2} = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2)$ creates a conceptual graph schema $\mathcal{G}_{b3}$ = `DETACH` $(\mathcal{G}_{b1}, \mathcal{G}_{b2})$. If one graph is a sub-graph of another conceptual graph schema then applying `DETACH` operator over such graph represents set difference of the edge set. An edge can only be deleted using `DETACH` if $(\mathcal{E}_1 \cap \mathcal{E}_2) \neq \varnothing$ which means that both conceptual graph schema must share some common edges. Furthermore, the labels associated with these edges must be same that is, $\exists e_1 \in \mathcal{E}_1$ and $\exists e_2 \in \mathcal{E}_2$ such that $\xi_1(e_1) = \xi_2(e_2)$. The application of `DETACH` removes existing edges from a conceptual graph schema. The resulting conceptual graph schemas after the application of `DETACH` may contain disconnected nodes.

Example 31 *Edges can be deleted from a conceptual graph schema by using `DETACH`. As shown in Figure 8.3 applying `DETACH` between conceptual graph schemas $\mathcal{G}_{b1}$ and $\mathcal{G}_{b3}$ results in conceptual graph schema $\mathcal{G}_{b4}$ that only contains an edge between node n_2 and n_3. That is $\mathcal{G}_{b4}$ = `DETACH` $(\mathcal{G}_{b1}, \mathcal{G}_{b3})$ such that $\eta(n_1) = \texttt{user}, \eta(n_2) = \texttt{review}$ and $\eta(n_3) = \texttt{listing}$. Furthermore, node n_1 is not the source and target of any edge in the conceptual graph schema.*

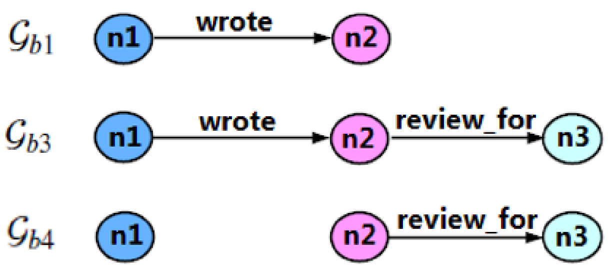


Figure 8.3: The application of `DETACH` operator to delete an edge from a conceptual graph schemas

`DELETE_NODE` is used to delete disconnected nodes in a conceptual graph schema. This operator is useful if a database designer wishes to delete existing nodes that are not connected to any other nodes in a conceptual graph schema. That is nodes that are neither the source nor the target of any edge in a conceptual graph schema. A node can only be deleted by using `DELETE_NODE` if $\mathcal{N}_1 \cap \mathcal{N}_2 \neq \emptyset$. This means that both graph must share common nodes. Furthermore, $\exists n_i \in \mathcal{N}_1$ and $\exists n_j \in \mathcal{N}_2$ such that $\eta_1(n_i) = \eta_2(n_j)$ which means that both nodes must have same node label.

Example 32 *Disconnected nodes can be deleted from a conceptual graph schema by using the* `DELETE_NODE`*. As shown in Figure 8.4 applying the* `DELETE_NODE` *operator between conceptual graph schemas $\mathcal{G}_{b4}$ and $\mathcal{G}_d$ results in a conceptual graph schema $\mathcal{G}_{b7}$ that only consists of nodes n_2, n_3 and an edge connecting nodes n_2 and n_3. The resulting graph does not contain any disconnected node. That is $\mathcal{G}_{b7}$ = $\texttt{DELETE\_NODE}(\mathcal{G}_{b4}, \mathcal{G}_d)$ such that $\eta(n_2)$ = `review` and $\eta(n_3)$ = `listing`. The graph $\mathcal{G}_d$ only consists of a node n_1 such that $\eta(n_1)$ = `user` and this node has been removed from the conceptual graph schema $\mathcal{G}_{b4}$.*

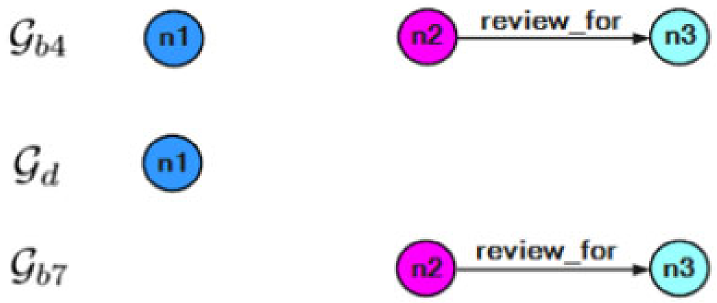


Figure 8.4: The application of `DELETE_NODE` operator to delete a node from a conceptual graph schemas

Using `JOIN` and `DETACH` together become helpful if the label and/or direction of edges in a conceptual graph schema have to be altered or changed. These operators, when used together, enables a designer to alter intensional information stored in a conceptual graph schema.

Example 33 *For instance if a designer wishes to alter the label and direction of an edge between node n_1 labeled as* `user` *and node n_2 labeled as* `review` *in the conceptual graph schema $\mathcal{G}_{b3}$ presented in Example 30. As shown in Figure 8.5 a designer can apply* `DETACH` *between graphs $\mathcal{G}_{b1}$ and $\mathcal{G}_{b3}$ which results in graph $\mathcal{G}_{b4}$ = $\texttt{DETACH}(\mathcal{G}_{b3}, \mathcal{G}_{b1})$. The designer can now define a basic conceptual graph schema $\mathcal{G}_{b5}$ where $\eta(n_1)$ = `user` and $\eta(n_2)$ = `review`. Applying the* `JOIN` *operator between* *graphs $\mathcal{G}_{b4}$ and $\mathcal{G}_{b5}$ results in conceptual graph schema $\mathcal{G}_{b6} = \mathtt{JOIN}(\mathcal{G}_{b4}, \mathcal{G}_{b5})$ as shown in Figure 8.5.*

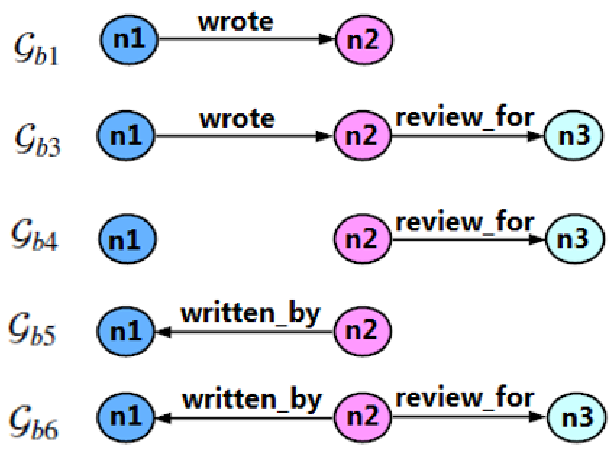


Figure 8.5: The application of `JOIN` and `DETACH` operators to alter an existing edge

8.3.3 Logical graph schema

A logical graph schema is used to capture extensional information of the entities and relations stored in a graph database. A logical graph schema is formed by enforcing integrity constraints on conceptual graph schema. Label uniqueness constraints are automatically enforced in the logical graph schema since the node, and edge labels used in conceptual graph schema are unique. For defining property-based constraints, we first define properties that can exist in graph databases. Properties in graph databases exist as key-value pairs where property values are atomic entities and have an associated data type. Logical graph schema stores properties as key-type format. Properties can be mandatory as well as optional for instance, properties such as ids must be unique. This information must be stored in a logical graph schema.

Let $\mathcal{K}$ be a set of infinite keys (e.g., id, name, age, etc.) and T be a finite set of data types (e.g., String, Integer, etc.) We define a set of properties $P \subseteq (\mathcal{K} \times T)$. The property set is of two types *(i) mandatory property set* (P_m) and *(ii) optional property* *set* (P_o) such that $P = P_m \cup P_o$. Mandatory property set can have some properties that have unique values associated with them. Let $\mathtt{U}$ be a set of Boolean values, we define a uniqueness function $\mathcal{U} : P_m \rightarrow \mathtt{U}$ that maps elements from mandatory property set to `TRUE` or `FALSE` signifying that some values associated with a mandatory property must be unique.

Edges in a graph schema also have semantic information such as cardinality associated with them which refers to total number of edges that can exist between any two given nodes of a graph database. Cardinality of an edge represents a range where the minimum value of cardinality refers to minimum number of edges that must exist between any two nodes of a graph databases. Similarly, maximum value of cardinality refers to maximum number of edges that can exist between any two nodes in a graph database.

Let $\mathtt{MIN} \in \mathbb{W}$ represent a minimum cardinality set which belongs to a set of whole numbers. Let $\mathtt{MAX} \in \mathbb{N}$ represents a maximum cardinality set which belongs to a set of natural numbers. We define a set of cardinalities as $\mathtt{C} \subseteq (\mathtt{MIN} \times \mathtt{MAX})$ with a condition that if $\mathtt{min} \in \mathtt{MIN}$ and $\mathtt{max} \in \mathtt{MAX}$ then $\mathtt{min} \leq \mathtt{max}$. This means that minimum cardinality can never be greater than maximum cardinality. The minimum cardinality belongs to a set of whole numbers which means that minimum cardinality can be zero. On the other hand maximum cardinality belongs to a set of natural numbers therefore, the smallest value that can be associated with maximum cardinality is 1. Furthermore, in such as scenario minimum cardinality can be either 0 or 1.

A logical graph schema extends the conceptual graph schema discussed in Definition 19 by labeling the nodes and edges with mandatory and optional properties. Moreover, in a logical graph schema edges are labeled with cardinality values. Formally, a logical graph schema is defined as follows:

Definition 22 (Logical Graph Schema) *A logical graph schema $\mathcal{G}_l$ is a tuple*

$(\mathcal{N}_s, \mathcal{E}_s, P_m, P_o, \mathtt{C}_s, \eta_s, \xi_s, \Delta_m, \Delta_o, \zeta_s)$ *where,*

- *$(\mathcal{N}_s, \mathcal{E}_s, \eta_s, \xi_s)$ is a conceptual graph schema as presented in Definition 19.*
- *$\Delta_m : (\mathcal{N}_s \cup \mathcal{E}_s) \rightarrow \mathcal{P}^+(P_m)$ is a mandatory property labeling function that maps all nodes and edges to the non empty subset of the mandatory property set where $\mathcal{P}^+(P_m)$ represents the powerset of mandatory property set excluding the empty set.*
- *$\Delta_o : (\mathcal{N}_s \cup \mathcal{E}_s) \rightarrow \mathcal{P}(P_o)$ is an optional property labeling function that maps all nodes and edges to the powerset, represented as $\mathcal{P}(P_o)$, of the optional property set.*
- *$\zeta_s : \mathcal{E}_s \rightarrow \mathtt{C}_s$ is a cardinality labeling function that maps all edges to a set of cardinalities such that $\forall e \in \mathcal{E}_s$, the cardinality function $\zeta_s(e) = (\mathtt{min}, \mathtt{max})$ returns a minimum and maximum value pair such that $\mathtt{min} \le \mathtt{max}$, $\mathtt{min} \in \mathtt{MIN}$ and $\mathtt{max} \in \mathtt{MAX}$.*

Example 34 *By using Definition 22 the logical graph schema generated for Airbnb case study is presented in Figure 8.6. The logical graph schema's topology is the same as the conceptual graph schema presented in Figure 8.1.*

Based in Definition 22 we can observe that a logical graph schema extends the conceptual graph schema by defining the property labeling functions over the nodes and edges of conceptual graph schema. Therefore, the intensional information captured in the conceptual graph schema is maintained in the logical graph schema. Additionally, the logical graph schema consists of extensional information as unique, mandatory and optional properties [226]. Furthermore, the data type associated with each property is also captured in the logical graph schema.

Example 35 *Figure 8.6 shows the properties associated with nodes and edges of the logical graph schema. For instance, the node labeled as* `host` *consists of a mandatory and an optional property. The mandatory property* `host_id` *is of data type Integer and must be unique. The value associated with the Boolean flag being* `TRUE` *signifies the uniqueness constraint. The optional property* `name` *is of data type String and does not contain the uniqueness constraint. As discussed in Definition 22 edges of the logical graph schema contain information about the cardinality. For instance, the edge between node labeled as* `host` *and* `listing` *is labeled as* `owns` *and the cardinality associated in* `(1,n)`*. This means that a host can own multiple listings and a listing can be associated with a single host. In the cardinality* `n` *represents a place holder for a natural number that can be calculated while creating the database creation script.*

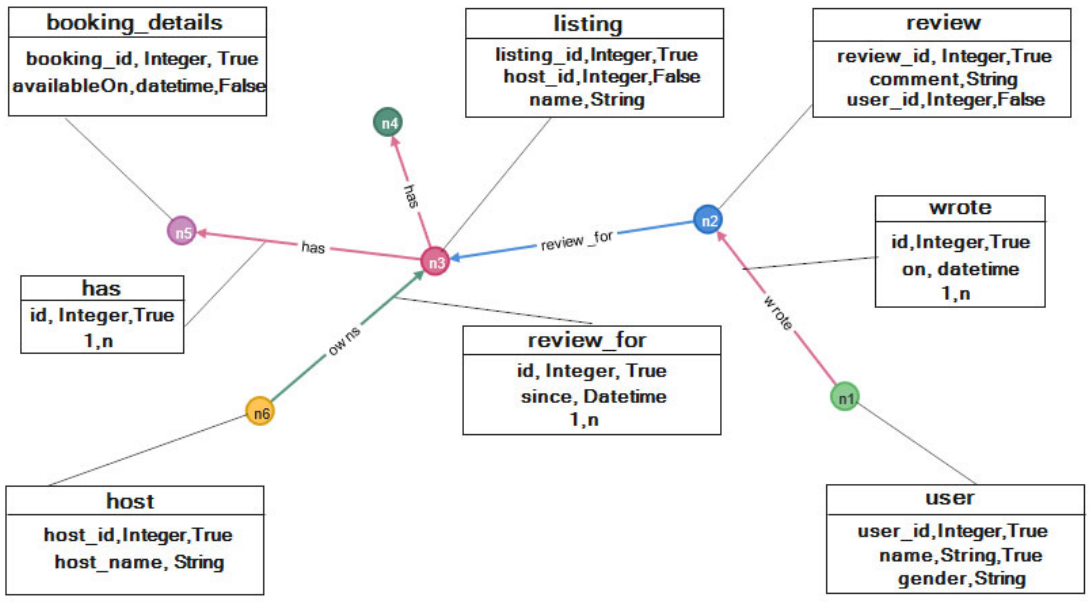


Figure 8.6: Logical graph schema generated for Airbnb case study

In our approach, the combination of conceptual and logical graph schema modeling stages represent the four steps of database design as suggested by Chen [227]. Information such as identify of entity set, relationship set and organization of data into entities and relationships is covered in conceptual graph schema modeling stage [226]. In the logical graph schema modeling stage semantic information such as cardinality of edges and properties associated with nodes and edges are defined [226].

FLASc operations for designing logical graph schemas

The three operations, `JOIN`, `DETACH` and `DELETE_NODE` can also be used for designing and manipulating the logical graph schema. As mentioned in Definition 22 a logical graph schema is an extension of conceptual graph schema. Therefore, node and edge labeling functions as well as source and target function are valid in a logical graph schema. The semantics associated with these functions are also same. A logical graph schema consists of additional functions such as mandatory and optional property labeling and edge cardinality functions. The use of FLASc operators namely `JOIN`, `DETACH` and `DELETE_NODE` is constrained due the additional labeling functions at the logical graph schema modeling stage. We now discuss the application of FLASc operators for logical graph schema modeling:

`JOIN`: The application of `JOIN` on two given logical graph schemas works in the similar manner as for source, target, node and edge labeling functions as presented in Definition 21. The additional mappings are required for property and cardinality labeling functions which are discussed as follows:

Definition 23 (`JOIN` on Logical Graph Schema) *Given two logical graph schemas* $\mathcal{G}_{l1} = (\mathcal{N}_{s1}, \mathcal{E}_{s1}, P_{m1}, P_{o1}, \mathtt{C}_{s1}, \eta_{s1}, \xi_{s1}, \Delta_{m1}, \Delta_{o1}, \zeta_{s1})$ *and* $\mathcal{G}_{l2} = (\mathcal{N}_{s2}, \mathcal{E}_{s2}, P_{m2}, P_{o2}, \mathtt{C}_{s2}, \eta_{s2}, \xi_{s2}, \Delta_{m2}, \Delta_{o2}, \zeta_{s2})$ *then* $\mathcal{G}_{l3} = \mathit{JOIN}(\mathcal{G}_{l1}, \mathcal{G}_{l2}) = \big((\mathcal{N}_{s1} \cup \mathcal{N}_{s2}), (\mathcal{E}_{s1} \cup \mathcal{E}_{s2}), (P_{m1} \cup P_{m2}), (P_{o1} \cup P_{o2}), (\mathtt{C}_{s1} \cup \mathtt{C}_{s2}), \eta_{s3}, \xi_{s3}, \Delta_{m3}, \Delta_{o3}, \zeta_{s3}\big)$ *where:*

- $\big((\mathcal{N}_{s1} \cup \mathcal{N}_{s2}), (\mathcal{E}_{s1} \cup \mathcal{E}_{s2}), \eta_{s3}, \xi_{s3}\big)$ *is a conceptual graph schema where node and edge labeling functions* η_{s3} *and* ξ_{s3} *are defined in the same way as in Definition 21.*

- $\Delta_{m3} : (\mathcal{N}_{s1} \cup \mathcal{N}_{s2} \cup \mathcal{E}_{s1} \cup \mathcal{E}_{s2}) \rightarrow \mathcal{P}^{+}(P_{m1} \cup P_{m2})$ *the mandatory property labeling functions that maps all the nodes and edges to the powerset (excluding the empty set) of the mandatory property sets of both the logical graph schemas.*
- $\Delta_{o3} : (\mathcal{N}_{s1} \cup \mathcal{N}_{s2} \cup \mathcal{E}_{s1} \cup \mathcal{E}_{s2}) \rightarrow \mathcal{P}(P_{o1} \cup P_{o2})$ *the optional property labeling function that maps all the nodes and edges to the powerset (including the empty set) of the optional property sets of both the logical graph schemas.*
- $\zeta_{s3} : (\mathcal{E}_{s1} \cup \mathcal{E}_{s2}) \rightarrow (\mathtt{C}_{s1} \cup \mathtt{C}_{s2})$ *is a cardinality labeling function which maps all edges to the cardinality sets of both the logical graph schemas.*

The notion of two logical graph schemas being join compatible is same as discussed for conceptual graph schemas as discussed in Section 8.3.2. With respect to the properties two logical graph schemas are join compatible if nodes have same mandatory and optional properties that is, $\exists n_1 \in \mathcal{N}_{s1}$ and $\exists n_2 \in \mathcal{N}_{s2}$ such that $\Delta_{m1}(n_1) = \Delta_{m2}(n_2)$ and $\Delta_{o1}(n_1) = \Delta_{o2}(n_2)$. In such a scenario we say that nodes n_1 and n_2 of two logical graph schemas are join compatible.

`DETACH`: The `DETACH` operator can be utilized by a database designer to delete an existing edge from a logical graph schema. Deleting an existing edge from a logical graph schema requires checking that the two conceptual graphs share some common edge with same labels as discussed in Section 8.3.2. Additionally, deleting edges in logical graph schemas also requires that the edge properties and cardinalities must be same. In order to formalize the notion of `DETACH` operator at the logical schema level we further divide the set of mandatory and optional properties into node and edge properties. Let $\mathtt{NP}_m$ and $\mathtt{EP}_m$ be two sets containing mandatory properties specific to nodes and edge respectively such that $P_m = \mathtt{NP}_m \cup \mathtt{EP}_m$. Similarly, let $\mathtt{NP}_o$ and $\mathtt{EP}_o$ be two sets containing optional properties specific to nodes and edge respectively then $P_o = \mathtt{NP}_o \cup \mathtt{EP}_o$

Definition 24 (`DETACH` on Logical Graph Schema) *Given two logical graph schemas* $\mathcal{G}_{l1} = (\mathcal{N}_{s1}, \mathcal{E}_{s1}, (\mathtt{NP}_{m1} \cup \mathtt{EP}_{m1}), (\mathtt{NP}_{o1} \cup \mathtt{EP}_{o1}), \mathtt{C}_{s1}, \eta_{s1}, \xi_{s1}, \Delta_{m1}, \Delta_{o1}, \zeta_{s1})$ *and* $\mathcal{G}_{l2} = (\mathcal{N}_{s2}, \mathcal{E}_{s2}, (\mathtt{NP}_{m2} \cup \mathtt{EP}_{m2}), (\mathtt{NP}_{o2} \cup \mathtt{EP}_{o2}), \mathtt{C}_{s2}, \eta_{s2}, \xi_{s2}, \Delta_{m2}, \Delta_{o2}, \zeta_{s2})$ *then* $\mathcal{G}_{l3} = \mathtt{DETACH}(\mathcal{G}_{l1}, \mathcal{G}_{l2}) = \big((\mathcal{N}_{s1} \cup \mathcal{N}_{s2}), (\mathcal{E}_{s1} \oplus \mathcal{E}_{s2}), (\mathtt{NP}_{m1} \cup \mathtt{NP}_{m2} \cup (\mathtt{EP}_{m1} \oplus \mathtt{EP}_{m2})), (\mathtt{NP}_{o1} \cup \mathtt{NP}_{o2} \cup (\mathtt{EP}_{o1} \oplus \mathtt{EP}_{o2})), (\mathtt{C}_{s1} \oplus \mathtt{C}_{s2}), \eta_{s3}, \xi_{s3}, \Delta_{m3}, \Delta_{o3}, \zeta_{s3}\big)$ *where:*

- $\big((\mathcal{N}_{s1} \cup \mathcal{N}_{s2}), (\mathcal{E}_{s1} \oplus \mathcal{E}_{s2}), \eta_{s3}, \xi_{s3}\big)$ *is a conceptual graph schema where node and edge labeling functions* η_{s3} *and* ξ_{s3} *are defined in the same way as in Definition 21.*

- $\Delta_{m3} : (\mathcal{N}_{s1} \cup \mathcal{N}_{s2} \cup (\mathcal{E}_{s1} \oplus \mathcal{E}_{s2})) \rightarrow \mathcal{P}^{+}((\mathtt{NP}_{m1} \cup \mathtt{NP}_{m2}) \cup (\mathtt{EP}_{m1} \oplus \mathtt{EP}_{m2}))$ *the mandatory property labeling functions that maps the nodes and edges to the powerset (excluding the empty set) of the mandatory node and edge property sets.*

- $\Delta_{o3} : (\mathcal{N}_{s1} \cup \mathcal{N}_{s2} \cup (\mathcal{E}_{s1} \oplus \mathcal{E}_{s2})) \rightarrow \mathcal{P}((\mathtt{NP}_{o1} \cup \mathtt{NP}_{o2}) \cup (\mathtt{EP}_{o1} \oplus \mathtt{EP}_{o2}))$ *the optional property labeling functions that maps the nodes and edges to the powerset (including the empty set) of the optional node and edge property sets.*

- $\zeta_{s3} : (\mathcal{E}_{s1} \oplus \mathcal{E}_{s2}) \rightarrow (\mathtt{C}_{s1} \oplus \mathtt{C}_{s2})$ *is a cardinality labeling function which maps edges to the cradinality sets.*

In order to delete existing edges by using the `DETACH` operator there must exist some edges that are common between two logical graph schemas that is $(\mathcal{E}_{s1} \cap \mathcal{E}_{s2}) \neq \varnothing$. This means that labels for both edges must be the same. Additionally, the properties and cardinalities associated with the common edges must be same as well that is $\exists e_1 \in \mathcal{E}_{s1}$ and $\exists e_2 \in \mathcal{E}_{s2}$ such that $\Delta_{m1}(e_1) = \Delta_{m2}(e_2), \Delta_{o1}(e_1) = \Delta_{o2}(e_2)$ and $\zeta_{s1}(e_1) = \zeta_{s2}(e_2)$.

`DELETE_NODE`**:** The `DELETE_NODE` operator can be utilized by a database designer to delete disconnected nodes from a logical graph schema. As discussed in Section 8.3.2 in order to delete an existing disconnected node the two logical graph schemas must contain common nodes. As mentioned in Definition 21 the node labeling must be same. Additionally the mandatory and optional properties must be the same as well.

Definition 25 (`DELETE_NODE` on Logical Graph Schema) *Given two logical graph schemas* $\mathcal{G}_{l1} = (\mathcal{N}_{s1}, \mathcal{E}_{s1}, (\mathtt{NP}_{m1} \cup \mathtt{EP}_{m1}), (\mathtt{NP}_{o1} \cup \mathtt{EP}_{o1}), \mathtt{C}_{s1}, \eta_{s1}, \xi_{s1}, \Delta_{m1}, \Delta_{o1}, \zeta_{s1})$ *and* $\mathcal{G}_{l2} = (\mathcal{N}_{s2}, \varnothing, (\mathtt{NP}_{m2} \cup \varnothing), (\mathtt{NP}_{o2} \cup \varnothing), \varnothing, \eta_{s2}, \varnothing, \Delta_{m2}, \Delta_{o2})$ *is a node labeled property graph then* $\mathcal{G}_{l3} = \mathit{DELETE\_NODE}(\mathcal{G}_{l1}, \mathcal{G}_{l2}) = \big((\mathcal{N}_{s1} \oplus \mathcal{N}_{s2}), (\mathcal{E}_{s1} \cup \varnothing), ((\mathtt{NP}_{m1} \oplus \mathtt{NP}_{m2}) \cup (\mathtt{EP}_{m1} \cup \varnothing)), ((\mathtt{NP}_{o1} \oplus \mathtt{NP}_{o2}) \cup (\mathtt{EP}_{o1} \cup \varnothing)), (\mathtt{C}_{s1} \cup \varnothing), \eta_{s3}, \xi_{s3}, \Delta_{m3}, \Delta_{o3}, \zeta_{s3}\big)$ *where:*

- $\big((\mathcal{N}_{s1} \oplus \mathcal{N}_{s2}), (\mathcal{E}_{s1} \cup \varnothing), \eta_{s3}, \xi_{s3}\big)$ *is a conceptual graph schema where node and edge labeling functions* η_{s3} *and* ξ_{s3} *are defined in the same way as in Definition 21.*

- $\Delta_{m3} : \big((\mathcal{N}_{s1} \oplus \mathcal{N}_{s2}) \cup (\mathcal{E}_{s1} \cup \varnothing)\big) \to \mathcal{P}^{+}((\mathtt{NP}_{m1} \oplus \mathtt{NP}_{m2}) \cup (\mathtt{EP}_{m1} \cup \varnothing))$ *the mandatory property labeling functions that maps the nodes and edges to the powerset (excluding the empty set) of the mandatory property sets of both the logical graph schemas.*

- $\Delta_{o3} : \big((\mathcal{N}_{s1} \oplus \mathcal{N}_{s2}) \cup (\mathcal{E}_{s1} \cup \varnothing)\big) \to \mathcal{P}((\mathtt{NP}_{o1} \oplus \mathtt{NP}_{o2}) \cup (\mathtt{EP}_{o1} \cup \varnothing))$ *the optional property labeling function that maps the nodes and edges to the powerset (including the empty set) of the optional property sets of both the logical graph schemas.*

- $\zeta_{s3} : (\mathcal{E}_{s1} \cup \varnothing) \to (\mathtt{C}_{s1} \cup \varnothing)$ *is a cardinality labeling function which maps all edges to the cardinality sets. Since the logical graph schema* $\mathcal{G}_{s2}$ *is a node labeled property graph therefore, the function* ζ_{s3} *maps all the edges from logical graph schema* $\mathcal{G}_{s1}$ *to cardinality set* $\mathtt{C}_{s1}$ *of the logical graph schema* $\mathcal{G}_{s1}$*.*

In order to delete existing nodes by using the `DELETE_NODE` operator there must exist some nodes that are common between two logical graph schemas that is $(\mathcal{N}_{s1} \cap$

$\mathcal{N}_{s2} \neq \varnothing$). This means that labels for both nodes must be the same. Additionally, the mandatory and optional properties associated with the common nodes must be same as well that is $\exists n_1 \in \mathcal{N}_{s1}$ and $\exists n_2 \in \mathcal{N}_{s2}$ such that $\Delta_{m1}(n_1) = \Delta_{m2}(n_2)$ and $\Delta_{o1}(n_1) = \Delta_{o2}(n_2)$.

Axiomatic specifications of FLASc operations

The axiomatic specifications of any algebra enable us to check its completeness [169]. In order to show the axiomatic specification we use infix notation for the operators in FLASc. As such we use the ($\bowtie$) notation for the `JOIN` operator, ($\diamond$) notation for the `DETACH` operator and (∇) notation for the `DELETE_NODE` operator.

The axiomatic specification of FLASc operators is presented in Table 8.1. For defining the identity axiom, we define an identity graph $I_{\mathcal{G}} = (\varnothing, \varnothing)$ which means that the identity graph does not contain any nodes and edges. We can observe that `JOIN`, `DETACH` and `DELETE_NODE` operations follow associativity, commutativity, idempotent and identity axioms.

Table 8.1: Axiomatic specifications of operators in FLASc

Axioms	JOIN	DETACH	DELETE_NODE
Associativity	$\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G}\ [(\mathcal{G}_1 \bowtie \mathcal{G}_2) \bowtie \mathcal{G}_3 = \mathcal{G}_1 \bowtie (\mathcal{G}_2 \bowtie \mathcal{G}_3)]$	$\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G}\ [(\mathcal{G}_1 \diamond \mathcal{G}_2) \diamond \mathcal{G}_3 = \mathcal{G}_1 \diamond (\mathcal{G}_2 \diamond \mathcal{G}_3)]$	$\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G}\ [(\mathcal{G}_1 \nabla \mathcal{G}_2) \nabla \mathcal{G}_3 = \mathcal{G}_1 \nabla (\mathcal{G}_2 \nabla \mathcal{G}_3)]$
Commutativity	$\forall \mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}\ [\mathcal{G}_1 \bowtie \mathcal{G}_2 = \mathcal{G}_2 \bowtie \mathcal{G}_1]$	$\forall \mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}\ [\mathcal{G}_1 \diamond \mathcal{G}_2 = \mathcal{G}_2 \diamond \mathcal{G}_1]$	$\forall \mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}\ [\mathcal{G}_1 \nabla \mathcal{G}_2 = \mathcal{G}_2 \nabla \mathcal{G}_1]$
Identity	$\forall \mathcal{G}_1 \in \mathcal{G}\ [\mathcal{G}_1 \bowtie I_{\mathcal{G}} = \mathcal{G}_1]$	$\forall \mathcal{G}_1 \in \mathcal{G}\ [\mathcal{G}_1 \diamond I_{\mathcal{G}} = \mathcal{G}_1]$	$\forall \mathcal{G}_1 \in \mathcal{G}\ [\mathcal{G}_1 \nabla I_{\mathcal{G}} = \mathcal{G}_1]$
Idempotent	$\forall \mathcal{G}_1 \in \mathcal{G}\ [\mathcal{G}_1 \bowtie \mathcal{G}_1 = \mathcal{G}_1]$	$\forall \mathcal{G}_1 \in \mathcal{G}\ [\mathcal{G}_1 \diamond \mathcal{G}_1 = \mathcal{G}_1]$	$\forall \mathcal{G}_1 \in \mathcal{G}\ [\mathcal{G}_1 \nabla \mathcal{G}_1 = \mathcal{G}_1]$

$\bowtie$ = `JOIN` operator
$\diamond$ = `DETACH` operator
∇ = `DELETE_NODE` operator

The distributive axioms for the `JOIN`, `DETACH` and `DELETE_NODE` operators is presented in Table 8.2. The axiomatic specification of FLASc operators enable us to use FLASc for generating new graph schemas from existing logical and conceptual graph schemas.

Table 8.2: Distributive axiom of FLASc operators

FLASc operators	Axiomatic Specification
`JOIN` and `DETACH`	$\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G}\ [\mathcal{G}_1 \bowtie (\mathcal{G}_2 \diamond \mathcal{G}_3) = (\mathcal{G}_1 \bowtie \mathcal{G}_2) \diamond (\mathcal{G}_1 \bowtie \mathcal{G}_3)]$
`JOIN` and `DELETE_NODE`	$\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G}\ [\mathcal{G}_1 \bowtie (\mathcal{G}_2 \nabla \mathcal{G}_3) = (\mathcal{G}_1 \bowtie \mathcal{G}_2) \nabla (\mathcal{G}_1 \bowtie \mathcal{G}_3)]$
`DETACH` and `DELETE_NODE`	$\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G}\ [\mathcal{G}_1 \diamond (\mathcal{G}_2 \nabla \mathcal{G}_3) = (\mathcal{G}_1 \diamond \mathcal{G}_2) \nabla (\mathcal{G}_1 \diamond \mathcal{G}_3)]$

$\bowtie$ = `JOIN` operator

$\diamond$ = `DETACH` operator

∇ = `DELETE_NODE` operator

The integrity constraints that can be enforced by a logical graph schema presented in Definition 22 include graph entity integrity constraints such as property uniqueness, label uniqueness, property data type and mandatory property constraints. The enforcement of these constraints and semantics constraints such as edge pattern, graph pattern, and path pattern constraints can be done at the physical modeling stage by using database-specific query languages. Following the graph schema to generate database creation scripts at the physical modeling stage ensures data consistency.

Schema Instance consistency

The schema instance consistency is used to ensure that the labeled property graph database constructed at the physical modeling stage adheres to the logical graph schema generated by using FLASc. A labeled property graph database uses a graph structure for storing and managing data, allowing the modeling of real world entities as nodes and edges [196, 15]. Nodes are used to store data and relationships or interactions between nodes are stored as edges [32, 16]. Nodes and edges in a graph database can have properties associated with them. Let P_d be a set of properties of a graph database such that each $p_d \in P_d$ is a key-value pair where each value has a data type. To accommodate the existence of mandatory and optional properties the set of properties can be further written as $P_d = P_{dm} \cup P_{do}$. Formally a labeled property graph database is defined as follows:

Definition 26 (`Labeled Property Graph Database`) *A labeled property graph database $\mathcal{G}_d$ is a tuple $(\mathcal{N}_d, \mathcal{E}_d, P_{dm}, P_{do}, \eta_d, \xi_d, \Delta_{dm}, \Delta_{do})$ where,*

- *$\mathcal{N}_d$ is a finite set of nodes and $\mathcal{E}_d$ is a finite set of edges of the graph database*
- *($\mathcal{N}_d$, $\mathcal{E}_d$) a directed multigraph.*
- *P_{dm} and P_{do} are mandatory and optional property sets associated with the graph database.*
- *$\eta_d : \mathcal{N}_d \to L_{\mathcal{N}}$ is a node labeling function which maps all nodes to labels in the set of node labels $L_{\mathcal{N}}$.*
- *$\xi_d : \mathcal{E}_d \to L_{\mathcal{E}}$ is an edge labeling function which maps all edges to labels in the set of edge labels $L_{\mathcal{E}}$.*
- *$\Delta_{dm} : (\mathcal{N}_d \cup \mathcal{E}_d) \to \mathcal{P}^+(P_{dm})$ is a property labeling function which maps all nodes and/or edges to all subsets (excluding the empty set) of the mandatory property set P_{dm}.*
- *$\Delta_{do} : (\mathcal{N}_d \cup \mathcal{E}_d) \to \mathcal{P}(P_{do})$ is a property labeling function which maps all nodes and/or edges to all subsets (including the empty set) of the optional property set P_{do}.*

The notion of schema instance consistency implies that a labeled property graph database adheres the structural restrictions established by a labeled property graph schema [27]. Such a notion can be formally defined as follows:

Definition 27 (`Schema Instance Consistency`) *Given a labeled property graph database $\mathcal{G}_d = (\mathcal{N}_d, \mathcal{E}_d, P_{dm}, P_{do}, \eta_d, \xi_d, \Delta_{dm}, \Delta_{do})$ as defined in Definition 26 and a labeled property graph schema $\mathcal{G}_l = (\mathcal{N}_s, \mathcal{E}_s, P_{sm}, P_{so}, \mathtt{C}_s, \eta_s, \xi_s, \Delta_{sm}, \Delta_{so}, \zeta_s)$ as defined in Definition 22. We say that $\mathcal{G}_d$ is consistent with $\mathcal{G}_l$ when:*

- *For each node $n \in \mathcal{N}_d$, there must exist a corresponding node in graph schema where $n' \in \mathcal{N}_s$ such that $\eta_d(n) = \eta_s(n')$.*
- *For each edge $e_i \in \mathcal{G}_d$ there must exist a corresponding edge in graph schema that is $e'_i \in \mathcal{G}_l$ such that $\eta_d(s(e_i)) = \eta_s(s(e'_i))$, $\eta_d(t(e_i)) = \eta_s(t(e'_i))$ and $\xi_d(e_i) = \xi_s(e'_i)$.*
- *For each $n_i \in \mathcal{N}_d$ (or $e_i \in \mathcal{E}_d$), there exists $n'_i \in \mathcal{N}_s$ (or $e'_i \in \mathcal{E}_s$) such that $\Delta_{dm}(n_i) = \Delta_{sm}(n'_i)$ (or $\Delta_{dm}(e_i) = \Delta_{sm}(e'_i)$). The data type of value stored in node (or edge) of graph database is same as the data type of node (or edge) in the graph schema.*
- *For each $n_i \in \mathcal{N}_d$ (or $e_i \in \mathcal{E}_d$), there exists $n'_i \in \mathcal{N}_{so}$ (or $e'_i \in \mathcal{E}_{so}$) such that $\Delta_{do}(n_i) = \Delta_{so}(n'_i)$ (or $\Delta_{do}(e_i) = \Delta_{so}(e'_i)$). The data type of value stored in node (or edge) of graph database is same as the data type of node (or edge) in the graph schema.*
- *The total number of edges of a certain label generated in the labeled property graph database must be between the minimum and maximum cardinality values associated with edges of same label in the graph schema.*

Cardinality can be enforced programatically at the physical modeling stage by using the logical graph schema generated by FLASc. Similarly, the adherence to node and edge labeling, property (optional and mandatory) labeling can be enforced at the physical modeling stage. The logical graph schema is independent of the underlying implementations. Moreover, the graph schema can be used in both integrated and layered physical modeling approaches. To support our claim in the following two sections, we experimentally demonstrate the use of graph schema to transform and load data-sets by using both approaches for physical modeling graph databases.

8.4 Using FLASc to enforce integrity constraints

In this section, we demonstrate the use of graph schema generated by FLASc for enforcing integrity constraints, which are essential for ensuring data consistency in graph databases. We illustrate the manual integration of conceptual, logical and physical modeling stages. We design the database creation scripts using the logical graph schema generated by FLASc for Airbnb data-set as shown in Figure 8.6. We do not make any changes to the source code of Neo4j; however, the formulation of database creation scripts in Cypher is driven by the logical graph schema. We then execute these scripts directly over the Neo4j graph database.

As discussed in [15] Airbnb data-set consists of three CSV files containing information related to listings, review and calendar data. The listing file contains information, such as hosts that own the listings, amenities provided in the listings, location of the listing etc. The reviews file contains information related to the users who have stayed in the listings and provided feedback in reviews. The calendar file contains information related to booking details such as pricing and occupancy. These files contain multiple lines (rows) of data, where each row contains a comma-separated list of values. For instance, a CSV file containing information related to listings from Airbnb's data is shown in Table 8.3.

Table 8.3: Sample data from listing.csv in the Airbnb data-set

Host Name	**Listing ID**	**Listing Name**	**Room Type**	**Street**	**Host ID**
Manju	9835	Beautiful Room & House	Private room	Bulleen, VIC, Australia	33057
Lindsay	10803	Room in Cool Deco Apartment in Brunswick East	Private room	Brunswick East, VIC, Australia	38901
Eleni	15246	Large private room-close to city	Private room	Thornbury, VIC, Australia	59786
Eleni	68482	Charming house inner Melbourne	Entire home/apt	Thornbury, VIC, Australia	59786

8.4.1 Manual generation of database creation scripts

The logical graph schema generated by FLASc for Airbnb data-set contains intensional and extensional information that assists a database designer for enforcing integrity constraints in the database scripts.

Enforcement of graph entity integrity constraints

Graph entity integrity constraints are used to enforce restrictions on properties associated with nodes and edges in a graph database. The extensional information captured in the logical graph schema as discussed in Definition 22 is used to enforce such constraints. We discuss the enforcement of graph entity integrity constraints for transforming and loading Airbnb data-set into Neo4j graph database by using Cypher query language.

Node property uniqueness constraint: The sample listing file as shown in Table 8.3 has `Listing ID` associated with each listing. Furthermore, in the logical graph schema shown in Figure 8.6 `listing_id` field the uniqueness flag is set to be `True` which means that the `listing_id` must be unique. Therefore, before creating the listing nodes in the Neo4j graph database, the uniqueness constraint must be established to reduce data corruption chances. This is achieved by running the following query in Cypher.

QUERY 21: Cypher query to enforce node property uniqueness constraint

```
CREATE CONSTRAINT unique_listing_id IF NOT EXISTS ON
 (list:listing)
ASSERT list.listing_id IS UNIQUE
```

The uniqueness constraint specified in Query 21 ensures that multiple nodes with same `listing_id` are not created in the Neo4j graph database. The use of `IF NOT EXISTS` clause is used to ensure that the constraint is enforced at most once. The next constraints to be enforced are the mandatory node and edge property constraints.

Mandatory node property constraint: The sample listing file also contains information about the `host_id` and in the logical graph schema as shown in Figure 8.6, the `host_id` is a mandatory field. Therefore, additional constraints must be enforced on the listing nodes. This can be achieved by running the following query in Cypher.

QUERY 22: Cypher query to enforce mandatory node property constraint

```
CREATE CONSTRAINT listing_host_id IF NOT EXISTS ON
 (list:listing)
ASSERT EXISTS list.host_id
```

The node property existence constraint specified in Query 22 ensures that listing nodes must always have a value assigned to the property `host_id` the `ASSERT EXISTS` clause is used to enforce such a condition.

Mandatory edge property constraint: The mandatory property constraints can also be specified on the edges that have to be created in the graph database. The logical graph schema as discussed in Definition 22 helps in enforcing this constraint in two ways; first, it provides details about the edge labels. Second, it also provides details about mandatory, unique and optional properties associated with the edges. For example, as shown in Figure 8.6 the edge labeled as `owns` has a mandatory property `since` which can be enforced by running the following Cypher query.

QUERY 23: Cypher query to enforce mandatory edge property constraint

```
CREATE CONSTRAINT owns_edge_id IF NOT EXISTS ON
 ()-[owns:OWNS]->()
ASSERT EXISTS owns.id
```

The mandatory edge property constraint shown in Query 23 is used to ensure that their is always a value assigned to `id` of every edge labeled as `OWNS` in the graph database.

Node key constraint: This constraint can be applied over a set of node properties. This constraint combines the functionality provided by uniqueness and mandatory property constraints. For example, the node labeled as `host` has two mandatory and unique properties `user_id` and `name`. This constraint can be enforced in the Neo4j graph database by using the following Cypher query.

QUERY 24: *Cypher* query to enforce node key property constraint

```
CREATE CONSTRAINT ON (u:user)
ASSERT u.user_id, u.name IS NODE KEY
```

As shown in Query 24 the use of `IS NODE KEY` keywords along with the `ASSERT` clause is used to enforce that the properties `user_id` and `name` are unique and must have a value associated with them in the graph database.

Property data type constraint: Logical graph schema is used to enforce property data type constraint over the node and edge properties. As discussed in Definition 22 a logical graph schema contains properties that have a data type associated with them. Therefore, database creation scripts are designed by utilizing this information. For instance, in the logical graph schema shown in Figure 8.6 `listing_id` and `host_id` are of Integer data type the Cypher query to enforce this constraint is as follows:

QUERY 25: Cypher query to enforce property data type constraint

```
LOAD CSV WITH HEADERS FROM
 "http://data.insideairbnb.com/australia/
vic/melbourne/2021-01-10/visualisations/listings.csv" AS row
WITH DISTINCT row.id AS listing_id, row.host_id AS host_id
MERGE(list:listing{listing_id:toInteger(listing_id),
 host_id:toInteger(host_id)})
```

The property data type constraint presented in Query 25 is enforced by using the inbuilt `toInteger()` function provided by Cypher query language. The use of this function is due the specification in logical graph schema that the data type associated with `listing_id` and `host_id` must of Integer type. In Query 25 the use of Cypher's `MERGE` clause represents the creation of two nodes that is a listing node and a host node. This also illustrates the combination of conceptual and logical modeling stages where a basic conceptual graph schema containing two disconnected nodes as discussed in Definition 20 is further labeled with node properties further representing the use of node labeling function (η) as discussed in Definition 22.

Other graph entity integrity constraints such as *node and edge label uniqueness* are by default maintained by the logical graph schema generated using FLASc. By definition 22 a node/edge can only have one label associated with it. On the other hand, Neo4j allows a node to be associated with more than one label. FLASc does not support this for the sake of simplicity. As mentioned in [42], these features are not present in all graph database systems and tend to make the definitions of graph schema and graph databases complex. Constraints such as *edge property uniqueness* can be specified in FLASc however, such constraints cannot be enforced in Neo4j.

Enforcement of semantic integrity constraints

Semantic integrity constraints are used to enforce a topological restriction on the graph database. The intensional information captured in the graph schema during the conceptual modeling stage becomes useful to enforce semantic integrity constraints.

Edge pattern constraint: To enforce edge pattern constraint the topological information stored in the logical graph schema is used while creating the database creation scripts. For instance, Query 26 is used to create edges between nodes of label `host` and `listing`. Each edge created by using Query 26 is labeled as `owns` and represents a valid edge in the logical graph schema shown in Figure 8.6.

QUERY 26: Cypher query to enforce edge pattern constraint

```
LOAD CSV WITH HEADERS FROM
 "http://data.insideairbnb.com/australia/
vic/melbourne/2021-01-10/visualisations/listings.csv" AS row
MATCH (l:listing),(h:host)
WHERE l.listing_id = toInteger(row.id)
AND h.host_id = toInteger(row.host_id)
MERGE (h)-[:owns{since:datetime(row.last_review)}]->(l)
```

The MATCH clause in Query 26 is used to obtain already existing listing and host node from the graph database. The WHERE clause at is used to define some constraints to filter results based on the values obtained from the csv files. The MERGE clause in Query 26 represents the creation of a basic conceptual graph schema containing two nodes and an edge connecting them as discussed in Definition 20. The edge of the basic conceptual graph schema is further labeled with edge properties further representing the use of edge labeling function (ξ) as discussed in Definition 22.

Graph pattern constraint: Enforcing graph pattern constraints require knowledge about the topology of the data-set, which is captured by logical graph schema. These constraints check for the existence of certain graph structure in the database before any new node or edge can be created. Graph pattern constraint in Cypher is presented as Query 27 which ensures that `listing` nodes that have been `reviewed` by a `user` are attached to `booking_detail` nodes by edges that are labeled as `has`.

QUERY 27: Cypher query to enforce graph pattern constraint

```
:auto USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
 "http://data.insideairbnb.com/australia/
vic/melbourne/2021-01-10/visualisations/calendar.csv" AS row
MATCH (u:user)-[:wrote]->(r:review),
 (r)-[:review_for]->(l:listing)
WHERE l.listing_id = toInteger(row.listing_id)
MERGE (l)-[:has{id:toInteger(row.id)}]->(b:booking_detail)
```

In Query 27 the MATCH clause is used to check if graph pattern exists or not. This graph pattern [32] is built by using the intensional information in the logical graph schema presented in Figure 8.6 that assists in formulating valid graph patterns for enforcing such constraints. The MATCH clause in this query connects two graph patterns which are join compatible [220]. The MERGE clause is used to combine the graph obtained from the MATCH clause with a logical graph schema specified in the MERGE clause. This represents the use of JOIN operator. The two logical graph schemas are join compatible since they share the node l labeled as `listing`. Query 27 also illustrates the use of `USING PERIODIC COMMIT` clause, which is used to handle the large amount of data being processed.

Path pattern constraint: These constraints check for the existence of certain paths in a graph database before a new node or edge can be created. Query languages for graph databases use the formalism of conjunctive two-way regular path queries `(C2RPQs)` and nested regular expressions `(NREs)` to express and then search for path patterns [124, 115, 125, 64, 62, 33, 66, 174, 30]. In these formalisms regular expressions defined over the edge labels of the graph database are used to describe path patterns [32]. The intensional information captured in logical graph schema assists in creating valid path patterns. Query 28 illustrates the enforcement of path pattern constraint in Cypher. Very similar to Query 27 the use of `MERGE` clause in the query represents the use of `JOIN` operator to combine the graph obtained from the `MATCH` clause at line 3 with the logical graph schema specified in the `MERGE` clause.

QUERY 28: *Cypher* query to enforce path pattern constraint

```
:auto USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
 "http://data.insideairbnb.com/australia/
vic/melbourne /2021-01-10/visualisations/calendar.csv" AS
 row
MATCH (u:user)-[:wrote]->()-[review_for]->l
WHERE l.listing_id = toInteger(row.listing_id)
MERGE(l)-[:HAS{id:toInteger(row.id)}]->
(a:amenityamenity_id:toInteger(row.amenity_id))
```

In Query 28 the path pattern constraint is specified in the `MATCH` clause, which represents the regular expression `(wrote.review_for)` formed by applying concatenation operator over the edge labels `wrote, review_for` and `has`. Other regular expressions operators such as union and Kleene star can also be used to form more expressions. However, Cypher only provides limited support for regular expressions as the Kleene star operator's use over the concatenation of two more edge labels is not allowed in Cypher [32, 220]. Further modifications can be done to the query language by using formalism such as Tarski's algebra instead of regular expressions for increasing their expressiveness [220].

Other Constraints such as schema instance consistency are ensured since the generation of database creation scripts is driven by the logical graph schema. Constraints such as functional dependencies are not easy to enforce in graph databases [23]; however, in order to enforce functional dependencies while modeling graph databases, a designer can follow the approach proposed in [144]. This approach states that every non-key property must only provide information about the associated nodes and edges. Constraints such as edge identify uniqueness and cardinality constraints cannot be directly enforced in Neo4j. However, enforcing such constraints can be done by writing a wrapper in programming languages such as Java, Python that can be used to ensure that edge ids must be unique.

The logical graph schema generated by FLASc enables us to enforce several practical integrity constraints. FLASc assists in the generation of robust conceptual and logical graph schemas. FLASc can be integrated with the existing Extract-Transform-load process for ensuring data consistency when data from heterogeneous sources is being loaded into a graph database such as Neo4j. The manual approach presented in this section has limitations. Firstly this approach requires a database designer to possess knowledge of graph database query language such as Cypher. Secondly, creating the database creation scripts manually can be cumbersome and error-prone, making the process less maintainable, scalable and manageable. Finally, Cypher does not support loading data from heterogeneous sources into the Neo4j graph database. Therefore, to mitigate such limitations in the next section, we present our layered approach.

8.5 A layered approach for data transformation and loading using FLASc

Graph databases are schema-less or schema optional; therefore, maintaining data consistency and integrity is not easy. A graph database can be easily altered unless the database's underlying source code is not amended to support the enforcement of all integrity constraints. Hence in this section, we propose a *layered approach* that incorporates the development of an additional wrapper to ensure data consistency. While following the layered approach, we use the APIs provided by Neo4j to access the graph database. We illustrate how FLASc can be used to assist the transformation and loading of data from heterogeneous sources into graph databases hence addresses RQ3 and RQ3.1.

8.5.1 Schema driven layered approach

Overview: The overall physical view of our layered approach is presented in Figure 8.7, that consists of three main components *(i) FLASc which serves as a graph schema generator*, *(ii) an importing subsystem* and *(iii) a graph database such as Neo4j*.

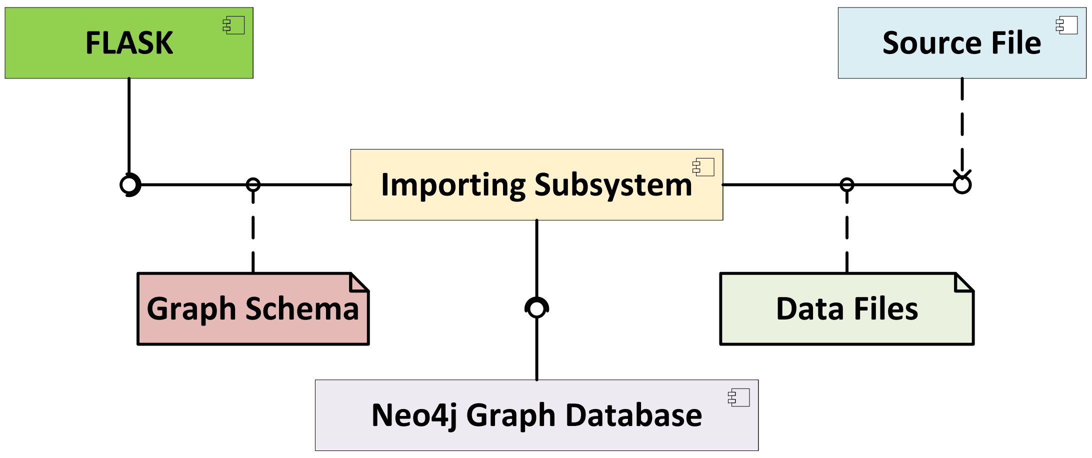


Figure 8.7: Physical view of Schema driven layered approach

The importing subsystem takes source files and a graph schema generated by FLASc as inputs. The subsystem then creates database creation scripts in $Cypher$ by following the intensional and extensional information captured in the graph schema. The subsystem then interacts with the Neo4j graph database by using the APIs and executes the database creation scripts on the graph database.

Importing subsystem design: The importing subsystem is based on the Extract-Transform-Load (ETL) design pattern. As shown in Figure 8.8 the *Extract* stage is used to fetch data from a source and consolidated it into a repository. The *transform* stage is used to apply appropriate transformation rules over the repository data. The transform stage uses the graph schema generated by FLASc to apply the transformation rules and create the database creation scripts. The *load* stage is finally used to execute the scripts on the database. In the load stage, database is accessed by using the specific API calls.

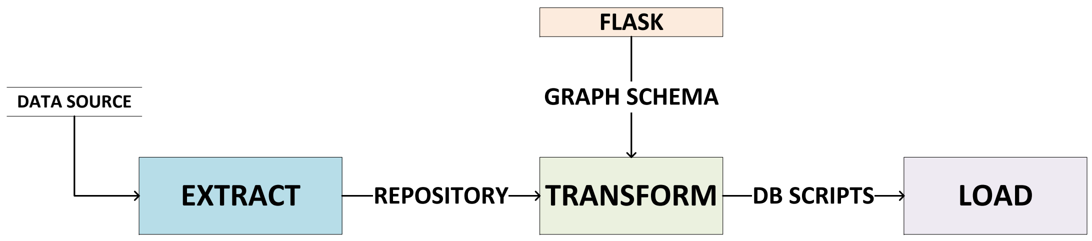


Figure 8.8: Process view of Schema driven layered approach

Technology stack: The subsystem is developed as a Java Maven project where the front end is designed using Java Swing library[3]. The subsystem uses Neo4j libraries for establishing a connection with the Neo4j graph database. Maven is used for handling API specific external dependencies. Neo4j's Cypher language is used for querying and creating the database.

[3]The source code is available for download at `removedforblindreview.github.com`

8.5.2 Airbnb case study

Transforming and loading data in CSV format is straight forward in Neo4j and $Cypher$. Furthermore, the Airbnb data-set exists in the form of denormalized relational tables as such connection between nodes can be established based on primary key foreign key relationships. As shown in Query 27, the clause `LOAD CSV WITH HEADERS FROM` represents the *extract* stage. In Query 27 the data is being fetched from the Airbnb website as shown in line 2-3. The data is stored in a repository represented by the "row" variable in the query. The *transform* stage in Query 27 is represented in lines 4-6 where the `MATCH` clause is used to search for the existence of a pattern, `WHERE` clause is used to restrict the result set based on some conditions and finally the `MERGE` clause is used to create the edge between node labeled as `listing` and `booking_details`. The transform stage is also responsible for ensuring that the integrity constraints are enforced, which is done by using the graph schema. In a layered approach, the *load* stage is responsible for creating a connection with the Neo4j graph database by making appropriate API calls. The additional wrapper written in Java is used to execute the entire query on Neo4j finally.

The main advantage of using the layered approach is that additional logic can be written to ensure data consistency. For instance, $Cypher$ does not provide inbuilt mechanisms to enforce the uniqueness constraints on edges. A layered approach is beneficial in such scenarios as additional logic can be written in programming languages to generate unique values for a particular edge property. The layered approach's advantage is evident when data in formats other than CSV are to be loaded into the Neo4j graph database. To illustrate this, we present the use of our layered approach to transform and load data-set related to big data analytics case study.

8.5.3 `BiDaML` case study

Implementing large-scale big data projects requires ongoing collaborations and monitoring by multiple stakeholders who have differing concerns. `BiDaML` (Big Data Analytics Modelling Languages) [86] is a domain-specific language for planning, specifying, monitoring and designing big data analytics projects. `BiDaML` suite presents different graph-based diagrams with highly interrelated data. The `BiDaML` diagrams considered in this case study consists of five diagrams brainstorming, process, technique, data, and deployment that provide different levels of abstractions. These diagrams are generated for National Bowel Cancer Screening Program (NBCSP) in Australia [228].

The `BiDaML` suite currently lacks the necessary automation and tooling required to allow individual users to view customised information specific to their needs and preferences within these diagrams. Importing data-sets from highly structured tools, such as the current HTML based implementation of `BiDaML` diagrams into graph databases such as Neo4j, is a challenge. This is due to the reason that Neo4j does not provides clauses for importing HTML data. We illustrate the use of our schema driven approach for transforming and loading `BiDaML` diagrams into Neo4j.

`BiDaML` diagrams data-set

The `BiDaML` data-set consists of five diagrams generated by the `BiDaML` suite. *Brainstorming diagram* provides an overview of a data analytics project and all the tasks and sub-tasks involved in designing the solution at a very high level. Users can include comments and extra information for the other stakeholders. *Process diagram* specifies the analytics process, which includes sequencing the tasks identified in the brainstorming diagram and relating these tasks to participants or stakeholders. *Technique diagrams* show how tasks from the brainstorming/process diagrams are elaborated further by applying specific techniques. *Data diagrams* document the data and artefacts produced in each of the above diagrams at a low level, i.e. the technical AI-based layer. They also define the outputs associated with different tasks like output information, reports, results, visualisations, and outcomes. And finally, *deployment diagrams* depicts the run-time configuration, i.e. the system hardware, the software installed on it, and the middle-ware connecting different machines for development related tasks.

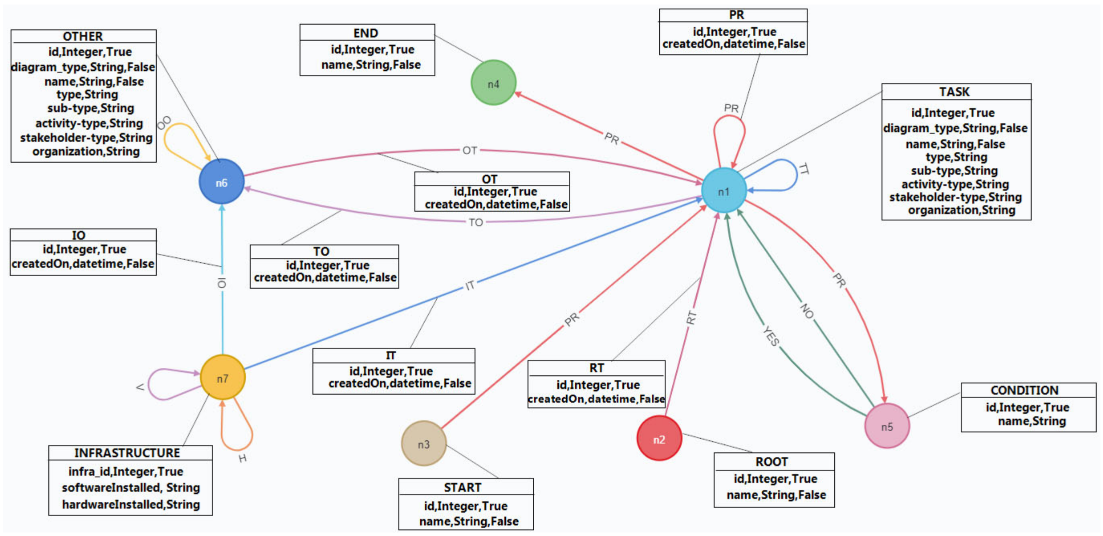


Figure 8.9: Logical graph schema for `BiDaML` diagrams

The graph schema generated by using FLASc for `BiDaML` diagrams is presented in Figure 8.9 where the node labeled as `TASK` allows edges that are available in different diagrams, including outgoing edges to other tasks allowed in brainstorming, process and technique diagrams. These edges are distinguished from each other via additional edge labels. For instance, edges between task nodes in brainstorming diagrams are labeled as `TT`. Edges between task nodes in process diagrams are labeled by `PR`. The schema also allows other node labels like `ROOT` in brainstorming diagrams, `START`, `END` and `CONDITION` in process diagrams and `INFRASTRUCTURE` node labels in deployment diagrams. In `BiDaML`, technique and data diagrams can have techniques and data artefacts that are used as nodes in deployment diagrams. For simplicity of the graph schema, we classify techniques, artefacts, etc., as nodes of label `OTHER`.

As shown in Figure 8.9 graph schema also captures the extensional information such as mandatory, unique and optional properties related to nodes and edges of BiDaML diagrams. For example, the node labeled as TASK has nine associated properties where *id,diagram_type* and *name* are mandatory properties. The *id* property must be unique and properties including *type,activity_type* and *organization* are optional.

Importing subsystem for BiDaML diagrams data-set

To transform and load BiDaML diagram data-set into Neo4j we still use the same ETL design pattern with slight modification to each stage. As shown in Figure 8.10 data files in HTML format are passed to the *Extract* stage that consists of two processes: *Parse-HTML* and *Data builder*. The HTML file contains information about nodes and edges of BiDaML graphs using *map* tags as well as additional properties such as *id, name, type, sub-type, activity-type, stakeholder, comments* and *organization*. *Parse-HTML* process reads the entire HTML file by using the JSoup library [?] and creates a repository containing all the nodes and edges, which is then passed on to the *Data Builder* for further processing.

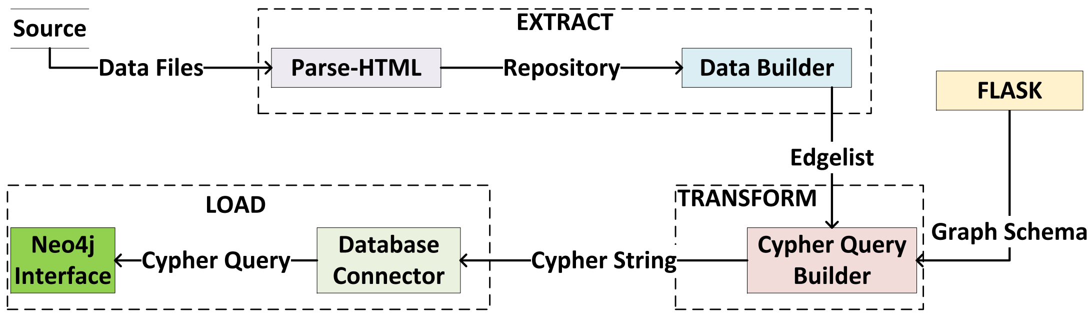


Figure 8.10: ETL stages shown as Data flow diagram to upload BiDaML diagram data-set into Neo4j

The *Data builder* process first removes duplicate elements in the repository. The builder then converts the repository into a list of edges (and nodes) that need to be stored in the graph database. In the *Transform* stage, the *Cypher Query Builder* takes the edge list from the extract stage and graph schema generated using FLASc as inputs to generate $Cypher$ queries for loading data into Neo4j. This stage also ensures that appropriate integrity constraints captured in the graph schema are enforced.

The final *load* stage consists of a *Database Connector* process and a Neo4j graph database interface. The *Database Connector* process establishes a connection with the Neo4j graph database using the Neo4j interface. A session is created between the subsystem and the Neo4j database. The Cypher query constructed in the transform stage is packaged into a create query and then executed. This process also ensures that nodes are not duplicated, especially if some of the imported nodes were already present in the database.

The time at which each node or edge is created during the ETL operations or during subsequent editing of the diagrams, is stored as a time stamp attribute within each updated element. Additional information, such as clustering of tasks in brainstorming diagrams and mapping tasks to specific stakeholders, is all stored as attributes of the corresponding nodes.

8.5.4 P2660.1 case study

Designing robust Industrial Cyber-Physical Systems (ICPS) largely depends upon identifying industrial agents, that provide complex and harmonious control mechanisms at the software level. These industrial agents practices are used to develop more extensive and feature-rich ICPS. IEEE Standardization projects such as P2660.1 aim at identifying industrial agent practices that can suit the requirements of future ICPS. A key challenge with this project is the identification of industrial agent practices based on some user-defined criteria. This case study is based on a tool (`IASelect`) developed for IEEE standardization project P2660.1 [85] that assists in selecting best fit industrial agent practices for ICPS [16].

P2660.1 data-set

The P2660.1 data-set consists of two practices *OnDevice* and *Hybrid*. Each practice is of two types *Tightly-coupled* and *loosely-coupled*. Practices have an associated set of qualities, which make these practices suitable to use in specific contexts. Hence, selecting the best-fit practices requires identifying the associated qualities. P2660.1 working group identifies four kinds of qualities *Domain, Function, Maintenance* and *Performance efficiency*. Each quality has an associated type; for instance, *Domain* has three associated types, including *Factory Automation, Building Automation* and *Energy*. Similarly, quality *Function* has three associated types *Monitoring, Control* and *Simulation*. The P2660.1 data-set exists in the form of an adjacency matrix where an ICPS expert assigns a score to a combination of practice and associated quality.

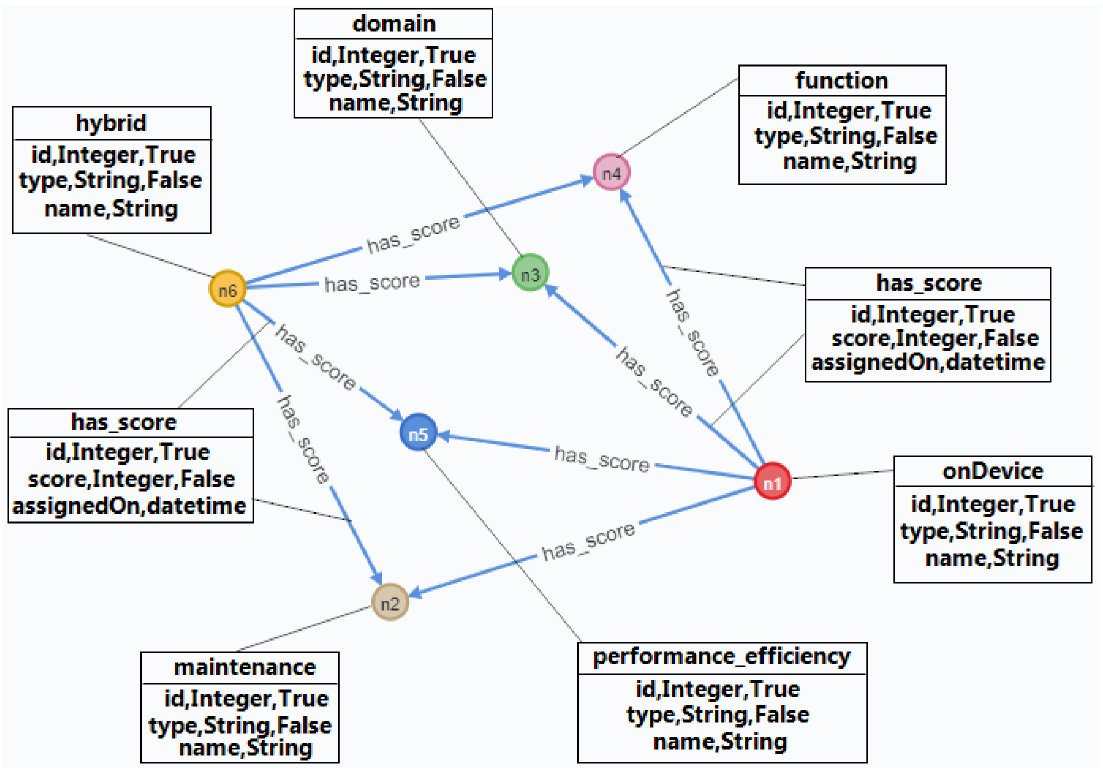


Figure 8.11: Logical graph schema for P2660.1 data-set

The graph schema generated by using FLASc for P2660.1 data-set is presented in

Figure 8.11 which consist of two practice nodes and four quality nodes. Each practice node is connected to a quality node by an edge labeled as `has_score`. This signifies that every practice to be stored in the graph database must connect with a quality, which represents the intensional information associated with the data-set. The extensional information is captured by node and edge properties. All nodes and edges have an associated property *id* which is a mandatory property, is of Integer data type and value associated with this property must be unique. Property such as *type* is mandatory but may not be unique. All edges have a unique and mandatory property *id*. The *score* property is mandatory but is not unique, and this is because the same score value can be assigned to different practice-quality pair by an ICPS expert. All edges contain an optional property *assignedOn* with an associated data type date-time.

Importing subsystem for P2660.1 data-set

To transform and load the P2660.1 data-set into Neo4j, we use the ETL design pattern with slight modifications. As shown in Figure 8.12 data in XLS file format containing an adjacency matrix is passed to the *Extract* stage that consists of two processes *Parse-AM* and *Data builder*.

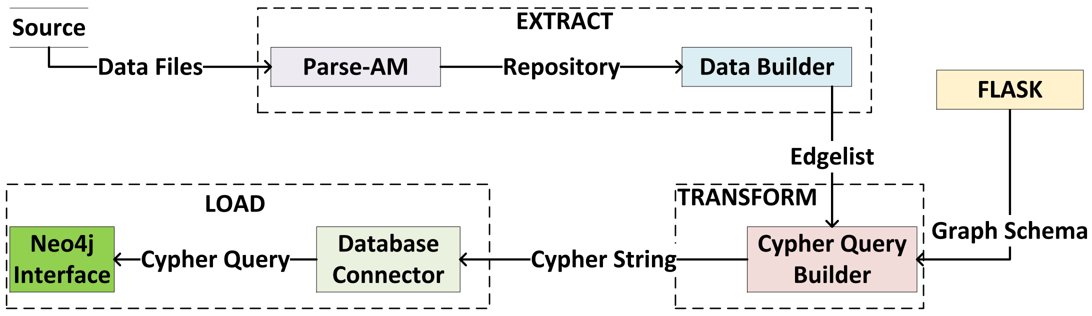


Figure 8.12: ETL stages shown as Data flow diagram to upload P2660.1 data-set into Neo4j

The *Parse-AM* process is used to reads the entire XLS file by using the Apache POI library [229] and converts it into a repository. The other process required to transform and load the P2660.1 data-set into Neo4j are similar to the processes used in the BiDaML diagram case study presented in Section 8.5.3.

8.5.5 Lessons learned from the case studies

The formal basis for FLASc and its integration with the ETL design pattern suggests that the data from heterogeneous sources can be transformed and loaded into several graph database by using our approach. We consider three case studies related to cyber-physical systems, big data analytics and tourism as presented in Sections 8.5.2, 8.5.3 and 8.5.4 respectively. The only factor that differs in loading these three diverse data-sets is the Extract phase's parse process.

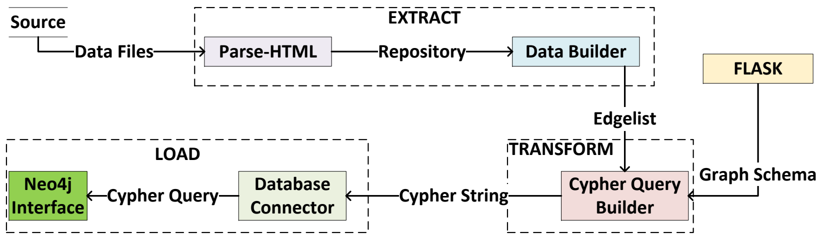


Figure 8.13: ETL stages shown as Data flow diagram to upload BiDaML diagram data-set into Neo4j

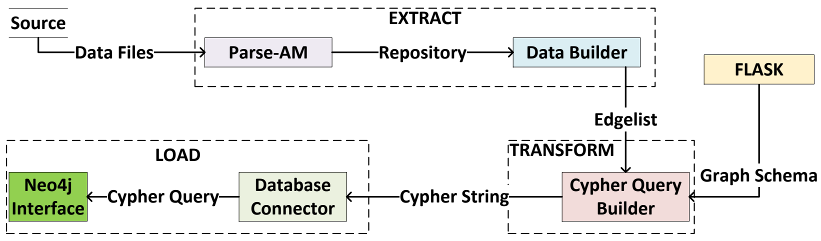


Figure 8.14: ETL stages shown as Data flow diagram to upload P2660.1 data-set into Neo4j

As shown in Figures 9.5 and 9.6 the parse process uses different APIs for reading data from heterogeneous sources. All other stages for loading data into the Neo4j graph database remain the same. Similarly, suppose data has to be transformed and loaded into a database other than Neo4j. In that case, only the Load stage needs to be altered so that APIs specific to the database platform can be utilized. The transform stage in all the scenarios as mentioned above remains the same and consistent. This demonstrates the generalizability of our approach, since by using the FLASc integrated ETL design pattern can be used to load data-sets from heterogeneous sources into a graph database. Furthermore, our approach is not limited to a specific data-set format and a particular graph database.

The use of FLASc for loading data-sets from heterogeneous sources becomes more evident when using the layered approach. As shown in Table 8.4 only a limited number of integrity constraints can be enforced in a layered approach without using FLASc. As shown in Table 8.3 structured data-sets such as provided in the Airbnb case study exist in the form of CSV files and contain intensional information as primary and foreign keys. However, semi-structured data provided in `BiDaML` and P2660.1 data-sets require predefined structural information for systematic transformation and loading. The intensional information is facilitated by using FLASc hence ensuring data consistency and integrity while using the layered approach.

8.6 Discussion, Conclusion and Future work

In this research, we present a formal algebra FLASc for generating robust graph schema for labeled property graph databases. We illustrate the integration of FLASc with the Extract-Transform-Load design pattern that assists in systematic transformation and loading of data-sets from heterogeneous sources into graph databases such as Neo4j. Graph schemas generated by FLASc assist in specifying integrity constraints in the database creation scripts, ensuring data consistency and integrity.

Our approach presents the integration of conceptual, logical and physical modeling stages for graph databases. FLASc enables users to capture requirements of any given problem domain as basic conceptual graph schemas. The `JOIN` and `DETACH` operators provided by FLASc can then be used to construct robust conceptual graph schemas from basic conceptual graph schemas. Properties associated with nodes and edges of graph schema are specified at the logical modeling stage. Finally, in the physical modeling stage, the enforcement of integrity constraints and design of database creation scripts are driven by FLASc generated graph schemas.

The integration of FLASc with the Extract-Transform-Load design pattern illustrates the practical application of our approach. This is demonstrated by using three diverse case studies related to cyber-physical systems, big data analytics and tourism. The intensional and extensional information captured in the graph schema assists in the *transform* stage of the data loading process. This information can be used to enforce several integrity constraints on the data-sets being loaded into a graph database.

Table 8.4: Coverage of integrity constraints

	Integrity constraints	**Integrated FLASc**	**Layered FLASc**	**Layered without FLASc**
Graph entity	Node Property Uniqueness	✓	✓	✓
	Node/Edge Label Uniqueness	✓	✓	×
	Edge property uniqueness	×	✓	×
	Mandatory Node property	✓	✓	✓
	Mandatory Edge property	✓	✓	✓
	Property data type	✓	✓	×
Semantic	Edge pattern	✓	✓	×
	Graph pattern	✓	✓	×
	Path pattern	✓	✓	×
Others	Type checking	✓	✓	×
	Edge Cardinality	×	✓	×
	Relationship Type	×	×	×

GN = Graph Navigation
GPM = Graph Pattern Matching

As shown in Table 8.4, FLASc facilitates the enforcement of several integrity constraints. We can observe that FLASc generated graph schemas are useful in enforcing semantic constraints because such constraints require knowledge of relationships between entities in data-sets. Semantic constraints such as edge, graph and path pattern constraints cannot be enforced without knowledge about relationships in the data-set. As shown in Table 8.4 graph entity integrity constraints such as edge property uniqueness constraint cannot be enforced in the integrated approach due to the limitations in the Neo4j graph database. Furthermore, FLASc generated logical graph schema also enable a database designer to specify cardinality constraints on the edges of a graph schema. However, due to the limitations in Neo4j graph database cardinality constraints cannot be enforced in the integrated approach. Such challenges can be mitigated in the layered approach by writing additional logic in programming languages such as Java, Python for specifying edge uniqueness and cardinality constraints.

The use of FLASc for loading data from heterogeneous sources becomes more evident while using the layered approach. As shown in Table 8.4 only a limited number of integrity constraints can be enforced in a layered approach without using FLASc. The support for integrity constraints such as node property uniqueness, mandatory node and edge property constraints are by default provided by Neo4j. Other constraints cannot be enforced without the intensional and extensional information captured in the graph schemas generated by FLASc. In general, the support for integrity constraints depends on the capabilities provided by the underlying graph database in the absence of a robustly defined graph schema.

The integration of FLASc and ETL design pattern suggests the generazibility of our approach, as data from heterogeneous sources can be transformed and loaded into different labeled property graph databases. As shown in Figures 8.10 and 8.12, while using our approach the only stage that differs in loading data-sets from heterogeneous sources is the parse process in *extract* stage. The parse process requires the use of specific APIs for reading different data formats. All other stages for loading data into the Neo4j graph database remain the same. Similarly, in scenarios where data has to be transformed and loaded into graph databases other than Neo4j, the *load* stage needs to be altered so that APIs specific to the database platform can be utilized, *transform* stage in all scenarios remain the same.

8.6.1 Limitations

As shown in Table 8.4 graph schemas generated by FLASc provide the ability to enforce several useful integrity constraints. However, other constraints relationship types is not covered in our approach. Relationship types represent the nature of relationships such as inheritance, association, composition and realisation, between nodes of a graph database. The enforcement of such constraints is not supported by FLASc in its current state. Furthermore, FLASc cannot be compared with other conceptual modeling tools such as entity-relationship diagrams (ERD) and unified modeling language (UML) diagrams as these tools support the specification of relationship types.

The main motive of FLASc is to assist in the design of robust conceptual graph schemas so that the soundness of logical and physical graph schemas can be ensured. FLASc generated conceptual graph schemas can preciously capture the intensional information. Relationship types are edge related properties [27]; hence can be classified as extensional information. These properties can be easily captured in the logical graph schema. For instance, by altering Definition 22, the logical graph schema can be enriched to support extensional information such as relationship types.

8.6.2 Conclusion and Future work

The scope of our study is limited to the Neo4j graph database. Therefore, the performance evaluation of using our approach for transforming and loading data-sets into other graph databases is not discussed. We consider this as future work where FLASc can be utilised for evaluating the coverage of integrity constraints offered by other graph databases provided by vendors such as Oracle [52], Apache Tinkerpop [53] and TigerGraph [223]. We intend to work on extending FLASc to support other integrity constraints such as cardinality constraints, relationship types and functional dependencies. The support of such constraints can enable FLASc to represent visual models expressed in languages such as Entity relationship diagram (ERD), Unified Modeling Language (UML) and System Modeling Language (SysML).

Moreover, using the FLASc extended ETL design pattern, visual models expressed as ERD, UML or SysML diagrams related to software development projects can be imported into graph databases. Storing software development visual models in graph databases provides the additional advantages of tractability and efficient database manageability, such as automatically identifying inconsistencies across all project diagrams.

In its current state our formal algebra FLASc supports the creation of robustly defined graph schemas that captures the intensional and extensional information. A natural extension to this work is the proposal of a formal schema creation language. We intend to combine our novel query language proposed in [220] with FLASc to propose a graph schema creation language. In [220] we propose the novel formalims of conjunctive queries and union of conjunctive queries extended with Tarksi's algebra (`CQT/UCQT`) for extracting data stored in a graph database. This language can be further combined with FLASc for creating a novel graph schema creation language. A main advantage of such an approach is the ability to use restricted form of first-order logic (conjunctive queries) while defining a graph schema which also makes our approach compatible with object role modeling language proposed in [230]. This will further assist in the industry wide initiative of standardizing query language for graph databases.

Chapter 9

Discussion and Conclusions

9.1 Introduction

Graph databases are ideal for handling highly interconnected data-sets. By using graph databases, real-world entities and relationships between them can be modeled as graphs. Data stored in graph databases can be inferred more efficiently when compared to relational databases [231, 232]. However, there exist several challenges that have obstructed the industry-wide adoption of graph databases. The literature review presented in Chapter 2 enables us to identify that theoretical language formalisms used by existing benchmarks for comparing graph query languages are not expressive enough. We also identified that existing graph database modeling approaches do not provide mechanisms to construct robust graph schemas.

In Chapters 5 and 6 we propose the novel formalisms of CQT and UCQT that are formed by extending conjunctive queries and union of conjunctive queries with Tarski's algebra. We present the construction of an integrated framework by considering common graph query patterns and use novel formalisms to construct benchmark queries. These benchmark queries are then used to compare the expressiveness of two practical graph query languages Cypher and PGQL objectively. The Airbnb case study presented in Chapters 3 and 4 is used as a common data-set for formulating the comparison queries. Our study serves as a formal basis for comparing and integrating contemporary graph query languages and assists projects such as ISO/IEC 39075.

In Chapters 7 and 8 we present FLASc a formal algebra that assists in formulating conceptual and logical graph schemas for labeled property graph databases. We illustrate the integration of FLASc with the well-known Extract-Transform-Load design pattern. We demonstrate the use of our approach to ensure data consistency while systematically transforming and loading data-sets from heterogeneous sources into a graph database such as Neo4j. These data-sets have been adopted from cases studies presented in Chapters A, B, 3 and 4. Overall, our approach displays the integration of conceptual, logical, and physical data modeling stages currently absent in existing graph database technologies.

The formalism related to graph schema and graph database as presented in Chapters 3 and 4 serves as a basis for constructing the integrated framework for query language comparison proposed in Chapters 5 and 6. Furthermore, the schema driven data loading and analytics approach presented in Chapter 4 enabled us to realized the importance of graph schema. Hence the design of formal algebra FLASc presented in Chapters 7 and 8 is driven by the findings from Chapter 4.

9.2 Major findings of this work

Chapters 5, 6, 7, and 8 represent the major contributions and work done for the completion of this thesis. The findings and insights obtained from these works are presented in the following sections.

9.2.1 Practical and Comprehensive Formalisms for Modeling Contemporary Graph Query Languages (Chapter 6)

There are three main research objectives of manuscript 1 presented in Chapter 6. The first research objective is related to the proposal of a novel theoretical language formalism that can be used for evaluating the expressiveness of practical graph query languages. We have discussed the syntax and semantics of CQT and UCQT in Sections 6.4 and 6.5 respectively. For addressing the second research objective in Section 6.6 we present the construction of an integrated framework by using the novel formalisms of CQT and UCQT. Finally, the third research objective demonstrates the use of an integrated framework to compare practical graph query languages Cypher and PGQL as presented in Sections 6.7 and 6.8 respectively. Major contributions and findings obtained after addressing the research objectives are summarized as follows.

Integrated framework for comparing graph query languages

The integrated framework presented in Section 6.6 assists in performing an in-depth examination of existing practical query languages. Furthermore, the two novel formalisms of CQT and UCQT presented in Sections 6.4 and 6.5 enable us to express graph navigation queries formed of cyclic and acyclic graph query patterns as shown in Figures 9.1, 9.2, 9.3 and 9.4.

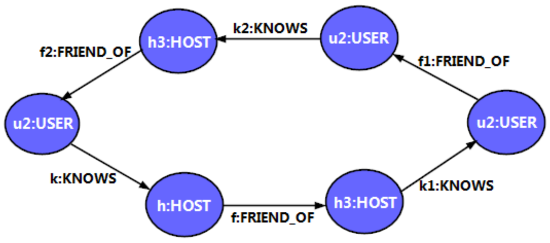


Figure 9.1: Plain cycle pattern

The graph navigation queries formed by using the cyclic and acyclic graph query

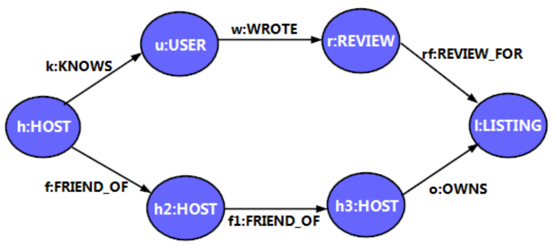


Figure 9.2: Petal pattern

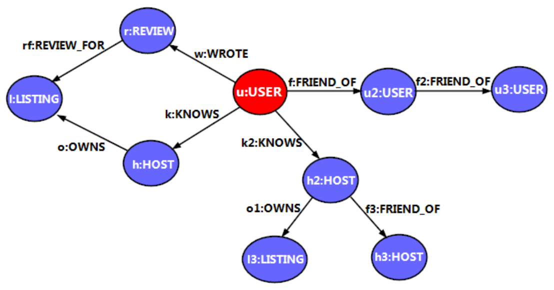


Figure 9.3: Flower pattern

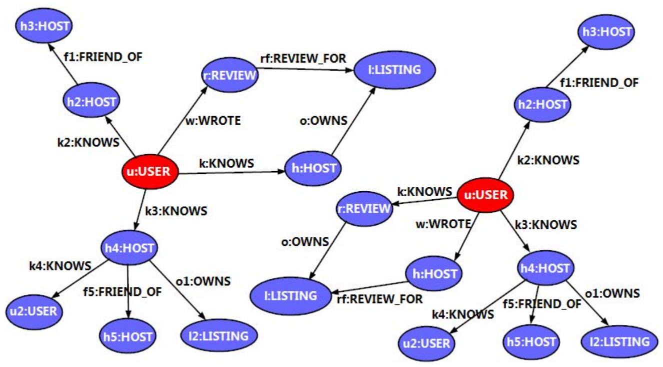


Figure 9.4: Bouquet pattern

patterns cannot be expressed in existing theoretical language formalisms. The analysis based on chain shaped graph query patterns presented in Section 6.7 reveals limitations of both Cypher and PGQL. For instance, Cypher does not allow the use of Kleene star over the concatenation of two or more edge labels; this finding is consistent with [32]. Both Cypher and PGQL support the use of union operator over edge labels only in two cases: when the inverse operator has not been applied at all or when the inverse operator has been applied over all the edge labels.

The integrated framework utilizes the graph schema of a problem domain to formulate meaningful benchmark queries that can be used to evaluate the expressiveness of different graph database query languages objectively. This can be achieved by generating database instances on different graph databases by using the same graph schema. For instance, comparative study between Cypher and PGQL presented in Sections 6.6 and 6.7 of Chapter 6 uses the same graph schema presented in Figure 6.3 to instantiate graph databases in Neo4j and Oracle. The graph schema is then used to formulate benchmark queries that are executed on both the systems.

The integrated framework can serve as a tool for creating and evaluating future graph query languages and can be used by the projects such as LDBC Task force [233] and ISO/IEC 39075. The novel formalisms of CQT and UCQT are more expressive than existing formalisms. The novel formalisms can be used to formulate chain and cycle shaped graph query patterns for expressing graph pattern matching and graph navigation queries as discussed in Section 6.6 of Chpater 6. Authors [188, 55, 35] suggest that chain and cycle shaped graph query patterns are integral for expressing more complex graph query patterns. Therefore, by using the formalisms of CQT and UCQT, more expressive benchmarks queries can be created for evaluating the expressiveness of graph query languages.

A comprehensive and objective comparison of Cypher and PGQL

We present a detailed comparison of two contemporary graph query languages Cypher and PGQL. As discussed in Section 6.7.3 our analysis shows that Cypher is more expressive than PGQL for graph pattern matching queries due to the presence of explicit `UNION` clause. For graph navigation queries PGQL is more expressive than Cypher due the presence of the `PATH` clause. In PGQL the use of `PATH` clause along with the `MATCH` clause enables the application of Kleene star over complex structures such as chains, trees, stars and star chains. Cypher on the other hand, does not provide such functionality and has limited expressiveness concerning graph navigation queries. Cypher is certainly more useful for searching sub-graphs in a graph databases. PGQL on the other hand is useful for navigating through the graph database. For graph navigation Cypher uses no-repeated edge isomorphism based semantics therefore, Cypher cannot output all existing paths in a graph database. The ability to output paths is only provided by Cypher and since PGQL uses arbitrary path semantics outputting paths can be problematic as infinite paths can be returned.

Our comparative study also shows that concerning graph navigation queries, cyclic and acyclic graph query patterns cannot be expressed in both Cypher and PGQL. As mentioned in [166, 77, 54] occurrence of cyclic and acyclic graph query patterns are not common in real-world data-sets. Therefore, practical graph query languages do not provide the ability to express such patterns. However, searching for such graph query patterns can be vital in graph database application fields such as bioinformatics and chemistry where for instance, a user might be interested in searching for the existence of long polymer chains of arbitrary-length that are formed of repeating acyclic or cyclic structures [31, 6].

Syntactic and semantic equivalence: As discussed in Section 6.4.5 formalisms of CQT and UCQT can be used to show syntactic equivalence of queries. However, as discussed in Section 6.5.4 query equivalence cannot be shown unless the evaluation and output semantics used by graph database engines are the same. Graph query languages use different evaluation and output semantics; therefore, to support interoperability between different graph query languages, vendors must provide means to enforce different evaluation semantics. As shown in Table 6.7 Cypher supports the enforcement of homomorphism based semantics for graph pattern matching queries. Similarly, PGQL supports the enforcement of isomorphism based semantics for graph pattern matching queries. A major limitation is observed for graph navigation queries where both Cypher and PGQL do not support the enforcement of similar evaluation semantics.

A primary reason for this because the result set of a graph navigation query may contain infinite paths if cycles exist in a graph database and appropriate evaluation semantics are not followed. Our this finding is also consistent with findings reported in [32, 25].

9.2.2 FLASc: A Formal Algebra for Labeled Property Graph Schema (Chapter 8)

There are two research questions addressed in Manuscript 2 presented in Chapter 8. Answering the first research enables us to propose our formal algebra FLASc. We present the syntax and semantics of FLASc in Section 8.3. Answering the second research question enables us to propose the integration of FLASc with the Extract-Transform-Load design pattern as presented in Sections 8.4 and 8.5. FLASc integrated ETL design pattern assists in the systematic import of data-sets from heterogeneous sources into graph database by ensuring data consistency and integrity. Major contributions and findings are as follows.

An algebra for generating robust conceptual graph schemas

The algebra FLASc is based on conceptual graphs presented by Sowa [47, 48, 49, 50]. In Section 8.3 we illustrate the use of FLASc for creating well-formed conceptual graph schemas from basic conceptual graph schemas. By using FLASc a user can define basic conceptual graph schemas, which are based on requirements of the problem domain. FLASc can formulate well-formed conceptual graph schemas from basic conceptual graph schemas by using the operators for `JOIN` and `DETACH` as discussed in Section **??**. These two operators of FLASc ensure that the problem domain's intentional information is well captured in the conceptual graph schema. The extensional information in the form of mandatory, optional, and unique properties are encoded in the conceptual graph schema at the logical modeling stage as presented in Section **??**. The generation of database creation scripts in the physical modeling stage is driven by the conceptual and logical graph schemas formulated by using FLASc as presented in Section **??**. The formal approach provided by FLASc for formulating conceptual graph schemas ensures their robustness which is an advancement over existing tools. Overall, our approaches present the integration of conceptual, logical, and physical modeling stages, which is lacking in contemporary graph databases. The graph schemas formulated using FLASc can be used for ensuring data consistency and integrity in graph databases. Furthermore, a formal approach used in FLASc can be used in studying and designing future data definition languages for contemporary graph databases.

Systematic import of data-sets into graph databases

Enforcement of robust integrity constraints using FLASc: The conceptual and logical graph schema helps enforce several applicable integrity constraints. Schemas formulated by FLASc ensure that both graph entity integrity and semantic integrity constraints can be enforced as presented in Section **??**. The use of graph schemas is evident while enforcing semantic integrity constraints. This is because enforcing such integrity constraints requires the knowledge of relationships between entities in data-sets; this finding is consistent with [134, 135]. Semantic constraints such as edge, graph and path pattern constraints cannot be enforced without knowledge about data-sets relationships. The majority of integrity constraints discussed in studies such as [81, 46, 44] can be enforced using conceptual and logical graph schemas formulated by FLASc.

Integration of FLASc and ETL design pattern: As presented in Section **??** the integration of FLASc with the Extract-Transform-Load design pattern illustrates the practical application of our approach. We use the intensional and extensional information captured in the graph schemas to assist in the transform stage of the design pattern as shown in Figure 8.8. Using the graph schemas formulated by FLASc database creation scripts can be created that contain the integrity constraints hence ensuring data consistency and integrity. Furthermore, the topology of database scripts is driven by the graph schema. The use of ETL design pattern facilitates the utility of our approach for loading data-sets into other graph databases, including Oracle [52] and TigerGraph [223] that follow the labeled property graph data model.

Transforming and loading data-sets from heterogeneous sources: The formal basis for FLASc and its integration with the ETL design pattern suggests that the data from heterogeneous sources can be transformed and loaded into several graph database by using our approach. We consider three case studies related to cyber-physical systems, big data analytics and tourism as presented in Sections **??**, **??** and **??** respectively. The only factor that differs in loading these three diverse data-sets is the Extract phase's parse process.

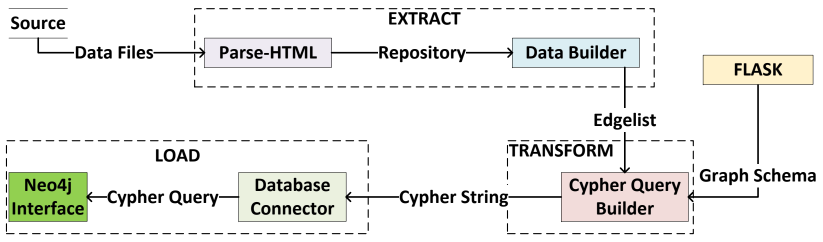


Figure 9.5: ETL stages shown as Data flow diagram to upload `BiDaML` diagram data-set into Neo4j

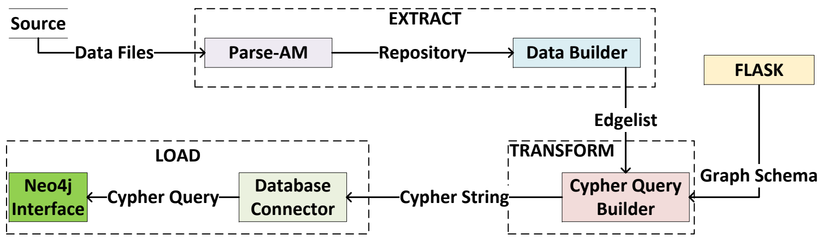


Figure 9.6: ETL stages shown as Data flow diagram to upload P2660.1 data-set into Neo4j

As shown in Figures 9.5 and 9.6 the parse process uses different APIs for reading data from heterogeneous sources. All other stages for loading data into the Neo4j graph database remain the same. Similarly, suppose data has to be transformed and loaded into a database other than Neo4j. In that case, only the Load stage needs to be altered so that APIs specific to the database platform can be utilized. The transform stage in all the scenarios as mentioned above remains the same and consistent. This demonstrates the generalizability of our approach, since by using the FLASc integrated ETL design pattern can be used to load data-sets from heterogeneous sources into a graph database. Furthermore, our approach is not limited to a specific data-set format and a particular graph database.

As presented in Section 8.6 the use of FLASc for loading data-sets from heterogeneous sources becomes more evident when using the layered approach. As shown in Table 8.4 only a limited number of integrity constraints can be enforced in a layered approach without using FLASc. As shown in Table 8.3 structured data-sets such as provided in the Airbnb case study exist in the form of CSV files and contain intensional information as primary and foreign keys. However, semi-structured data provided in `BiDaML` and P2660.1 data-sets require predefined structural information for systematic transformation and loading. The intensional information is facilitated by using FLASc hence ensuring data consistency and integrity while using the layered approach.

9.2.3 Practical and theoretical implications of this research

The work presented in this thesis serves as a bridge between existing theoretical literature related to graph database and the contemporary graph database technologies provided by the industry. On one hand there are many studies such as [62, 33, 76, 30, 72, 73, 71, 65, 36, 37, 34] that focus on theoretical language formalisms for graph databases. On the other hand there exist several practical graph query languages proposed by industry. Our research presented in Chapter 6 provides an explicit mapping between existing theoretical language formalisms and practical query languages for graph databases. Our study also enables us to identify common characteristics shared by theoretical language formalisms and practical graph query languages. Furthermore, our finding that existing formalisms are not expressive enough lead us to propose more expressive formalisms of CQT and UCQT. As presented in Section 6.6 these formalisms can be used to create and evaluate future query languages for graph databases.

Similarly, there exist several studies that only focus on logical and physical modeling aspects of graph databases. As mentioned in Section 8.2 conceptual modeling is an important data modeling aspect that assists in capturing data semantics of any problem domain. However, existing graph databases opt to be either schema-less or schema optional furthermore, conceptual modeling is conducted in an ad-hoc manner. Conceptual modeling has been widely studied in the literature as conceptual graphs [47, 48, 49, 50]. In our research presented in Chapter 8 we propose a formal algebra FLASc that can be used to generate robust conceptual and logical graph schemas for labeled property graph databases. The design of FLASc is motivated from conceptual graph and our approach illustrates the integration of conceptual, logical and physical modeling stages for graph databases. Moreover, the integration of FLASc with Extract-Transform-Load design pattern increases the efficiency of transforming and loading data-sets from heterogeneous sources into graph databases.

Our work also shows a strong correlation between graph schemas and the design of benchmark queries. Graph schemas provide the context of the problem domain; furthermore, by utilizing the intensional and extensional information stored in a graph schemas meaningful benchmark queries are generated. Using graph schemas for generating benchmark queries reduces the total number of queries generated, but not the coverage of all query classes. Hence, our work assists in generating more efficient benchmark queries that can be used to evaluate different query languages.

9.3 Limitations of this work

9.3.1 Practical and Comprehensive Formalisms for Modeling Contemporary Graph Query Languages (Chapter 6)

Comparison of data values in paths

We have considered Tarski's algebra for comparing the graph navigation features of query languages. However, formalisms such as Tarski's algebra are purely navigational and do not support comparisons of data values in path expressions [34]. We intend to include formalisms used in query languages such as GXPath, *regular expressions with memory (*`REM`*), walk logic* and *register logic* [65, 34, 73, 165, 72] along with Tarski's algebra for graph navigation in future studies. Furthermore, we have also not considered more expressive formalisms such as ECRPQs that enable path comparisons and context-free paths in our study. This is because features provided by such formalisms are not yet present in all practical query languages. Moreover, returning paths can be problematic if cycles exist in graph databases, as infinite paths can be returned if arbitrary path semantics are used. Another important concept of the algorithmic complexity associated with graph query languages and the extended formalism is not considered in this study, and we see it as future work.

Coverage of operation in CQT and UCQT

We have only considered operations such as natural join, selection, projection and union. Other operations such as *difference* is computationally more expensive [32]. As a result difference clause is not implemented yet in Cypher and PGQL but similar functionality can be simulated by using the `NOT EXISTS` clause along with `WHERE` clause. Cypher also allows the use of `NOT` keyword to simulate difference operation with a restriction that the pattern specified in the `WHERE NOT` clause must represent an eulerian path [171]. Cypher and PGQL do not have such restrictions in the `WHERE NOT EXISTS` clause.

The use of `EXISTS` and `NOT EXISTS` clause represents semi join and anti semi join respectively, that correspond to the *semi join algebra* and its equivalent formalism *guarded fragments* (GF) of first order logic [207, 63, 208, 209]. We do not consider the study of guarded fragment of first order logic in this study. Furthermore, we have also not considered other relational algebra operations such as *outer join* and *aggregate* functions. Outer join clause `OPTIONAL MATCH` is only present in Cypher while aggregate functions are used after the result set is returned by a pattern matching algorithm.

9.3.2 FLASc: A Formal Algebra for Labeled Property Graph Schema (Chapter 8)

Integrity constraints

FLASc provides the ability to enforce several useful graph entity integrity and semantic constraints; however, other constraints such as cardinality constraints, relationship types and functional dependencies are left out. Cardinality constraints are vital for ensuring the minimum and maximum number of edges between any two nodes of a graph database. Furthermore, expressiveness of graph schema formulated by FLASc cannot be compared with other conceptual modeling tools such as Entity-relationship diagrams (ERD) [234] and unified modeling language (UML) [235] diagrams. This is because tools such as ERDs and UML also support the specification of relationship types such as inheritance, association and composition. The enforcement of such constraints is not supported by FLASc in its current state.

However, the motive of FLASc is to assist in the design of robust conceptual graph schemas with the aim to preciously capture the intensional information. The number of allowed edges between nodes and the type of relationships are properties related to the entities of the graph database; hence, they can be classified as extensional information. These properties can be easily captured at the logical modeling stage. For instance, by altering Definition 22 presented in Chapter 8 the logical graph schema can be enriched to support extensional information such as cardinality and relationship types.

Diversity of graph databases

The scope of our study is limited to the Neo4j graph database. Therefore, the performance evaluation of using our approach for transforming and loading data-sets into other graph databases is not discussed. We consider this as future work where FLASc can be utilised for evaluating the coverage of integrity constraints offered by other graph databases provided by vendors such as Oracle [52], Apache Tinkerpop [53], and TigerGraph [223].

9.4 Conclusion

This thesis is primarily focused on achieving the following two research objectives:

RO1: Extend the existing theoretical language formalisms to propose novel formalisms that can be used to build benchmark queries for comparing the expressiveness of graph query languages.

RO2: Enhance the existing graph data modeling approaches and propose novel methods for constructing graph schemas so that data consistency and integrity can be ensured in labeled property graph databases.

For achieving RO1 we present an extension of conjunctive queries and union of conjunctive queries with Tarski's algebra in Chapter 6. We have proposed novel formalisms of CQT and UCQT that provide a formal basis to compare, integrate and model practical graph query languages. In order to objectively compare practical graph query languages, a framework is proposed that integrates the extended formalisms with common graph query patterns to generate a comprehensive set of benchmark queries.

This process is the basis of a comparative study of two practical graph query languages Cypher and PGQL. Our analysis shows that Cypher is more expressive than PGQL for graph pattern matching queries due to the presence of explicit `UNION` clause. For graph navigation queries PGQL is more expressive than Cypher due the presence of the `PATH` clause. In PGQL the use of `PATH` clause along with the `MATCH` clause enables the application of Kleene star over complex structures such as chains, trees, stars and star chains. Cypher on the other hand does not provide such functionality and has limited expressiveness concerning graph navigation queries.

Our study also shows that with respect to graph navigation queries, cyclic and acyclic graph query patterns cannot be expressed in both the languages. Such graph query patterns are important because they may have use in specialised graph databases used in fields like chemistry, biology and astronomy. This study's shape-based analysis can help identify common and exclusive characteristics for other currently available practical graph query languages such as SPARQL, GSQL, SQL/PG and Gremlin. Furthermore, the extended formalism and the integrated framework can be utilised to model future graph query languages; therefore, they serve as a basis for upcoming standards like IEC/ISO 39075.

For achieving RO2, we in Chapter 8 present a formal algebra FLASc for generating robust graph schema for labeled property graph databases. We illustrate the integration of FLASc with the Extract-Transform-Load design pattern that assists in systematic transformation and loading of data-sets from heterogeneous sources into graph databases such as Neo4j. Graph schemas generated by FLASc assist in specifying integrity constraints in the database creation scripts, ensuring data consistency and integrity.

Our approach presents the integration of conceptual, logical and physical modeling stages for graph databases. FLASc enables users to capture requirements of any given problem domain as basic conceptual graph schemas. The `JOIN` and `DETACH` operators provided by FLASc can then be used to construct robust conceptual graph schemas from basic conceptual graph schemas. Properties associated with nodes and edges of graph schema are specified at the logical modeling stage. Finally, in the physical modeling stage, the enforcement of integrity constraints and design of database creation scripts are driven by FLASc generated graph schemas.

The integration of FLASc with the Extract-Transform-Load design pattern illustrates the practical application of our approach. This is demonstrated by using three diverse case studies related to cyber-physical systems, big data analytics and tourism. The intensional and extensional information captured in the graph schema assists in the *transform* stage of the data loading process. This information can be used to enforce several integrity constraints on the data-sets being loaded into a graph database.

9.5 Future directions

As presented in Chapter 6 the use of conjunctive queries based formalism for graph pattern matching reveal similarities between query languages for graph and relational databases. Concerning difference operation, graph query languages such as Cypher and PGQL implement this operation by using semi-join algebra and equivalent guarded fragment of first-order logic. For graph navigation queries, authors in [74, 55, 14] also suggest similarities between Tarski's algebra and semi-join algebra. Semi join algebra can be particularly useful in query optimisation. Hence, we consider the study of graph query languages based on semi-join algebra as future work. Furthermore, insights from the work presented in this thesis can be used to build tools for supporting interoperability between relational database query language SQL and graph database query languages such as Cypher, PGQL, GSQL and SQL/PG. Initial ideas for developing such a tool called as **FLUX** have already been published and presented as Appendix C and D.

We intend to work on extending FLASc to support other integrity constraints such as cardinality constraints, relationship types and functional dependencies. The support of such constraints can enable FLASc to represent visual models expressed in languages such as Entity relationship diagram (ERD), Unified Modeling Language (UML) and System Modeling Language (SysML). Using the FLASc extended ETL design pattern, visual models expressed as ERD, UML or SysML diagrams related to software development projects can be imported into graph databases. Storing software development visual models in graph databases provides the additional advantages of tractability and efficient database manageability, such as automatically identifying inconsistencies across all project diagrams.

Another important future direction is the integration of FLASc with the novel formalisms of CQT and UCQT. Due to the lack of a standard query languages for graph databases, these formalisms can be useful in studying and comparing the enforcement of graph entity and semantic integrity constraints provided by existing graph languages. For instance, the `MATCH` clause in queries 26, 27 and 28 presented in Section 8.4 of Chapter 8 represent graph and navigation patterns discussed in Section 6.4 of Chapter 6. These queries can be expressed by using the novel formalism of CQT and UCQT. The intensional and extensional information captured by FLASc generated graph schemas can be formally expressed in the theoretical language formalisms of CQT and UCQT. Therefore, integrating FLASc with CQT and UCQT assists in formalising data definition languages for graph databases.

We intend to work on a graph schema driven template query generation tool. We can observe from our work presented in Chapter 6 that graph schemas assist in creating benchmark queries. The schema provides the context for creating the benchmark queries in the integrated framework. Therefore, the graph schema driven template query generation tool requires the merger of FLASc integrated ETL design pattern presented in Chapter 8 with CQT and UCQT based integrated framework presented in Chapter 6. The formalisms identified in this research will assist in creating the tool that can be used to test the expressiveness of other graph query languages such as SPARQL, Gremlin, SQL/PG and GSQL. The automatic generation of query language adapters is another interesting future direction of this work, which will help the community's shared goal of high interoperability between available graph database technologies.

References

[1] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. H. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda *et al.*, "A core for future graph query languages," 2018.

[2] P. Leitão, S. Karnouskos, L. Ribeiro, P. Moutis, J. Barbosa, and T. I. Strasser, "Integration patterns for interfacing software agents with industrial automation systems," in *Proceedings of the 44th Annual Conference of the IEEE Industrial Electronics Society (IECON'18).* IEEE, 2018, pp. 2908–2913.

[3] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014.

[4] A. Alex and M. Norbert, "Ldbc use case analysis and choke point analysis," 2013, accessed: 2019-03-01. [Online]. Available: http://ldbcouncil.org/sites/default/files/LDBC_D3.3.1.pdf

[5] I. V. Tetko, O. Engkvist, U. Koch, J.-L. Reymond, and H. Chen, "Bigchem: Challenges and opportunities for big data analysis in chemistry," *Molecular informatics*, vol. 35, no. 11-12, pp. 615–621, 2016.

[6] R. J. Hall, C. W. Murray, and M. L. Verdonk, "The fragment network: A chemistry recommendation engine built using a graph database," *Journal of medicinal chemistry*, vol. 60, no. 14, pp. 6440–6450, 2017.

[7] M. A. Rodriguez and P. Neubauer, "Constructions from dots and lines," *Bulletin of the American Society for Information Science and Technology*, vol. 36, no. 6, pp. 35–41, 2010.

[8] ——, "The graph traversal pattern," in *Graph Data Management: Techniques and Applications*. IGI Global, 2012, pp. 29–46.

[9] G. M. Kuper, "The logical data model: A new approach to database logic." STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1985.

[10] F. W. Tompa, "A data model for flexible hypertext database systems," *ACM Transactions on Information Systems (TOIS)*, vol. 7, no. 1, pp. 85–100, 1989.

[11] M. P. Consens and A. O. Mendelzon, "Expressing structural hypertext queries in graphlog," in *Proceedings of the second annual ACM conference on Hypertext*. ACM, 1989, pp. 269–292.

[12] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht, "A graph-oriented object database model," *IEEE Transactions on Knowledge & Data Engineering*, no. 4, pp. 572–586, 1994.

[13] S. Abiteboul, "Querying semi-structured data," Stanford InfoLab, Tech. Rep., 1996.

[14] C. Sharma, "Flux: From sql to gql query translation tool," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1379–1381.

[15] C. Sharma and R. Sinha, "A schema-first formalism for labeled property graph databases: Enabling structured data loading and analytics," in *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, 2019, pp. 71–80.

[16] C. Sharma, R. Sinha, and P. Leitao, "Iaselect: Finding best-fit agent practices in industrial cps using graph databases," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1. IEEE, 2019, pp. 1558–1563.

[17] O. Lassila, R. R. Swick *et al.*, "Resource description framework (rdf) model and syntax specification," 1998.

[18] J. J. Carroll and G. Klyne, "Resource description framework ({RDF}): Concepts and abstract syntax," 2004.

[19] P. Jorge, M. Arenas, C. Gutierrez *et al.*, "nsparql: A navigational language for rdf," *The Semantic Web-ISWC 2008*, vol. 5318, pp. 66–81, 2008.

[20] P. Cudré-Mauroux and S. Elnikety, "Graph data management systems for new application domains," *Proceedings of the VLDB Endowment*, vol. 4, no. 12, 2011.

[21] M. Arenas and M. Ugarte, "Designing a query language for rdf: marrying open and closed worlds," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 4, p. 21, 2017.

[22] G. H. Fletcher, H. Voigt, and N. Yakovets, "Declarative graph querying in practice and theory," in *EDBT/ICDT 2017 Joint Conference 20th International Conference on Extending Database Technology*, 2017.

[23] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 1, 2008.

[24] M. Ciglan, A. Averbuch, and L. Hluchy, “Benchmarking traversal operations over graph databases,” in *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, 2012, pp. 186–189.

[25] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc, “Foundations of modern graph query languages,” *CoRR, abs/1610.06264*, 2016.

[26] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, “Cypher: An evolving query language for property graphs,” in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1433–1445.

[27] R. Angles, “The property graph database model,” 2018.

[28] D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta, and J. L. Larriba-Pey, “A discussion on the design of graph database benchmarks,” in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2010, pp. 25–40.

[29] L. Libkin, J. L. Reutter, A. Soto, D. Vrgoč *et al.*, “Trial: A navigational algebra for rdf triplestores,” *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 1, p. 5, 2018.

[30] P. Barceló, J. Pérez, and J. L. Reutter, “Relative expressiveness of nested regular expressions.” *AMW*, vol. 12, pp. 180–195, 2012.

[31] G. Bell, T. Hey, and A. Szalay, “Beyond the data deluge,” *Science*, vol. 323, no. 5919, pp. 1297–1298, 2009.

[32] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, “Foundations of modern query languages for graph databases,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–40, 2017.

[33] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood, “Expressive languages for path queries over graph-structured data,” *ACM Transactions on Database Systems (TODS)*, vol. 37, no. 4, p. 31, 2012.

[34] D. Vrgoc, “Querying graphs with data,” 2014.

[35] J. Hellings, M. Gyssens, Y. Wu, D. Van Gucht, J. Van den Bussche, S. Vansummeren, and G. H. Fletcher, “Relative expressive power of downward fragments of navigational query languages on trees and chains,” in *Proceedings of the 15th Symposium on Database Programming Languages*, 2015, pp. 59–68.

[36] L. Libkin, J. Reutter, and D. Vrgoč, “Trial for rdf: adapting graph query languages for rdf data,” in *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, 2013, pp. 201–212.

[37] L. Libkin, W. Martens, and D. Vrgoč, “Querying graphs with data,” *Journal of the ACM (JACM)*, vol. 63, no. 2, p. 14, 2016.

[38] I. Robinson, J. Webber, and E. Eifrem, *Graph databases*. " O’Reilly Media, Inc.", 2013.

[39] R. L. Griffith, “Three principles of representation for semantic networks,” *ACM Transactions on Database Systems (TODS)*, vol. 7, no. 3, pp. 417–442, 1982.

[40] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley Reading, 1995, vol. 8.

[41] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[42] R. Angles, H. Thakkar, and D. Tomaszuk, “Mapping rdf databases to property graph databases,” *IEEE Access*, vol. 8, pp. 86 091–86 110, 2020.

[43] J. Pokornỳ, “Graph databases: Their power and limitations,” in *14th Computer Information Systems and Industrial Management (CISIM)*. Springer, 2015, pp. 58–69.

[44] ——, “Conceptual and database modelling of graph databases,” in *Proceedings of the 20th International Database Engineering & Applications Symposium*, 2016, pp. 370–377.

[45] J. Pokornỳ, M. Valenta, and J. Kovačič, “Integrity constraints in graph databases,” *Procedia Computer Science*, vol. 109, pp. 975–981, 2017.

[46] M. S. Barik, C. Mazumdar, and A. Gupta, “Network vulnerability analysis using a constrained graph data model,” in *International Conference on Information Systems Security*. Springer, 2016, pp. 263–282.

[47] J. F. Sowa, “Conceptual graphs for a data base interface,” *IBM Journal of Research and Development*, vol. 20, no. 4, pp. 336–357, 1976.

[48] ——, “Conceptual graphs summary,” *Conceptual Structures: current research and practice*, vol. 3, p. 66, 1992.

[49] J. Sowa, “Conceptual graphs: Draft proposed american national standard,” in *International Conference on Conceptual Structures*. Springer, 1999, pp. 1–65.

[50] J. F. Sowa, “Conceptual graphs,” *Foundations of Artificial Intelligence*, vol. 3, pp. 213–237, 2008.

[51] Neo4j, “Neo4j,” 2018, accessed: 2018-10-01. [Online]. Available: https://neo4j.com/developer/cypher/

[52] Oracle, "Oracle," 2018, accessed: 2018-11-01. [Online]. Available: https://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytix/downloads/index.html

[53] Apache, "Apache tinkerpop," 2021, accessed: 2021-01-02. [Online]. Available: https://tinkerpop.apache.org/

[54] A. Bonifati, W. Martens, and T. Timm, "An analytical study of large sparql query logs," *The VLDB Journal*, vol. 29, no. 2, pp. 655–679, 2020.

[55] J. Hellings, Y. Wu, M. Gyssens, and D. Van Gucht, "The power of tarski's relation algebra on trees," in *International Symposium on Foundations of Information and Knowledge Systems*. Springer, 2018, pp. 244–264.

[56] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi, "Answering regular path queries using views," in *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 2000, pp. 389–398.

[57] E. M. Alfonso and G. Stamou, "On horn conjunctive queries," in *International Joint Conference on Rules and Reasoning*. Springer, 2018, pp. 115–130.

[58] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational data bases," in *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1977, pp. 77–90.

[59] C. Chekuri and A. Rajaraman, "Conjunctive query containment revisited," in *International Conference on Database Theory*. Springer, 1997, pp. 56–70.

[60] A. Klug, "On conjunctive queries containing inequalities," *Journal of the ACM (JACM)*, vol. 35, no. 1, pp. 146–160, 1988.

[61] Y. E. Ioannidis and R. Ramakrishnan, "Containment of conjunctive queries: Beyond relations as sets," *ACM Transactions on Database Systems (TODS)*, vol. 20, no. 3, pp. 288–324, 1995.

[62] P. Barceló, L. Libkin, and J. L. Reutter, "Querying graph patterns," in *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2011, pp. 199–210.

[63] V. Bárány, B. Ten Cate, and M. Otto, "Queries with guarded negation," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1328–1339, 2012.

[64] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat, "Controlling diversity in benchmarking graph databases," *arXiv preprint arXiv:1511.08386*, vol. 11, 2015.

[65] L. Libkin, W. Martens, and D. Vrgoč, "Querying graph databases with xpath," in *Proceedings of the 16th International Conference on Database Theory*, 2013, pp. 129–140.

[66] P. Barceló, M. Romero, and M. Y. Vardi, "Semantic acyclicity on graph databases," *SIAM Journal on computing*, vol. 45, no. 4, pp. 1339–1376, 2016.

[67] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database," 2010.

[68] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, vol. 2324, 2013, p. 36.

[69] J. Su, Q. Zhu, H. Wei, and J. X. Yu, "Reachability querying: can it be even faster?" *IEEE Transactions on Knowledge & Data Engineering*, no. 1, pp. 1–1, 2017.

[70] J. X. Yu and J. Cheng, "Graph reachability queries: A survey," in *Managing and Mining Graph Data*. Springer, 2010, pp. 181–215.

[71] L. Libkin and D. Vrgoč, "Regular path queries on graphs with data," in *Proceedings of the 15th International Conference on Database Theory*. ACM, 2012, pp. 74–85.

[72] P. Barceló, G. Fontaine, and A. W. Lin, "Expressive path queries on graphs with data," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2013, pp. 71–85.

[73] P. Barceló Baeza, "Querying graph databases," in *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, 2013, pp. 175–188.

[74] J. Hellings, "On tarski's relation algebra: querying trees and chains and the semi-join algebra," 2018.

[75] A. Tarski, "On the calculus of relations," *The Journal of Symbolic Logic*, vol. 6, no. 3, pp. 73–89, 1941.

[76] P. Barceló, D. Figueira, and L. Libkin, "Graph logics with rational relations and the generalized intersection problem," in *2012 27th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2012, pp. 115–124.

[77] A. Bonifati, W. Martens, and T. Timm, "An analytical study of large sparql query logs," *arXiv preprint arXiv:1708.00363*, 2017.

[78] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat, "gmark: Schema-driven generation of graphs and queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 4, pp. 856–869, 2016.

[79] ——, "Generating flexible workloads for graph databases," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1457–1460, 2016.

[80] A. Badia and D. Lemire, “A call to arms: revisiting database design,” *ACM SIGMOD Record*, vol. 40, no. 3, pp. 61–69, 2011.

[81] F. Reina, A. Huf, D. Presser, and F. Siqueira, “Modeling and enforcing integrity constraints on graph databases,” in *International Conference on Database and Expert Systems Applications*. Springer, 2020, pp. 269–284.

[82] V. M. de Sousa and L. M. d. V. Cura, “Logical design of graph databases from an entity-relationship conceptual model,” in *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, 2018, pp. 183–189.

[83] M. Šestak, K. Rabuzin, and M. Novak, “Integrity constraints in graph databases–implementation challenges,” in *Proceedings of Central European Conference on Information and intelligent Systems*, 2016, pp. 23–30.

[84] M. Šestak, M. Heričko, T. W. Družovec, and M. Turkanović, “Applying k-vertex cardinality constraints on a neo4j graph database,” *Future Generation Computer Systems*, vol. 115, pp. 459–474, 2021.

[85] P2660.1, ““recommended practices on industrial agents: Integration of software agents and low level automation functions.”,” 2020, accessed: 2021-03-16. [Online]. Available: https://standards.ieee.org/standard/2660_1-2020.html

[86] H. Khalajzadeh, M. Abdelrazek, J. Grundy, J. Hosking, and Q. He, “Bidaml: A suite of visual languages for supporting end-user data analytics,” in *2019 IEEE International Congress on Big Data (BigDataCongress)*. IEEE, 2019, pp. 93–97.

[87] Airbnb, “Inside airbnb: Adding data to the debate,” 2018, accessed: 2019-02-03. [Online]. Available: http://insideairbnb.com/get-the-data.html

[88] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, “A graphical query language supporting recursion,” in *ACM SIGMOD Record*, vol. 16, no. 3. ACM, 1987, pp. 323–330.

[89] ——, “G+: Recursive queries without recursion,” in *Proceedings of the Second International Conference on Expert Database Systems*, 1988, pp. 355–368.

[90] B. Amann and M. Scholl, “Gram: a graph data model and query languages,” in *Proceedings of the ACM conference on Hypertext*, 1993, pp. 201–211.

[91] M. Gemis and J. Paredaens, “An object-oriented pattern matching language,” in *International Symposium on Object Technologies for Advanced Software*. Springer, 1993, pp. 339–355.

[92] J. Hidders and J. Paredaens, “Goal, a graph-based object and association language,” in *Advances in Database Systems*. Springer, 1994, pp. 247–265.

[93] M. Gyssens, J. Paredaens, and D. V. Gucht, “A graph-oriented object model for database end-user interfaces,” in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of data*, 1990, pp. 24–33.

[94] R. H. Güting, “Graphdb: Modeling and querying graphs in databases,” in *VLDB*, vol. 94. Citeseer, 1994, pp. 12–15.

[95] M. Graves, E. R. Bergeman, and C. B. Lawrence, “Querying a genome database using graphs,” in *Proceedings of the 3th International Conference on Bioinformatics and Genome Research*, 1994.

[96] J. Paredaens, P. Peelman, and L. Tanca, “G-log: A graph-based query language,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 3, pp. 436–453, 1995.

[97] M. Levene and G. Loizou, “A graph-based data model and its ramifications,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 5, pp. 809–823, 1995.

[98] N. M. Alex A., “Ldbc use case analysis and choke point analysis,” 2013, accessed: 2019-03-01. [Online]. Available: http://ldbcouncil.org/sites/default/files/LDBCD3.3.1.pdf

[99] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener, “The lorel query language for semistructured data,” *International journal on digital libraries*, vol. 1, no. 1, pp. 68–88, 1997.

[100] M. Fernández, D. Florescu, A. Levy, and D. Suciu, “Declarative specification of web sites with s,” *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 9, no. 1, pp. 38–55, 2000.

[101] P. Buneman, M. Fernandez, and D. Suciu, “Unql: a query language and algebra for semistructured data based on structural recursion,” *The VLDB Journal*, vol. 9, no. 1, pp. 76–110, 2000.

[102] A. Poulovassilis and S. G. Hild, “Hyperlog: A graph-based system for database browsing, querying, and update,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 2, pp. 316–333, 2001.

[103] R. Giugno and D. Shasha, “Graphgrep: A fast and universal method for querying graphs,” in *Object recognition supported by user interaction for service robots*, vol. 2. IEEE, 2002, pp. 112–115.

[104] G. Weikum, G. Kasneci, M. Ramanath, and F. Suchanek, “Database and information-retrieval methods for knowledge discovery,” *Communications of the ACM*, vol. 52, no. 1, pp. 56–64, 2009.

[105] S. Amer-Yahia, L. Lakshmanan, and C. Yu, “Socialscope: Enabling information discovery on social content sites,” *arXiv preprint arXiv:0909.2058*, 2009.

[106] A. Dries, S. Nijssen, and L. De Raedt, “A query language for analyzing networks,” in *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 2009, pp. 485–494.

[107] R. Ronen and O. Shmueli, “Soql: A language for querying and creating data in social networks,” in *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 2009, pp. 1595–1602.

[108] H. He and A. K. Singh, “Query language and access methods for graph databases,” in *Managing and mining graph data*. Springer, 2010, pp. 125–160.

[109] M. San Martın, C. Gutierrez, and P. T. Wood, “Snql: A social networks query and transformation language,” *cities*, vol. 5, p. r5, 2011.

[110] S. Sakr, S. Elnikety, and Y. He, “G-sparql: a hybrid engine for querying large attributed graphs,” in *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2012, pp. 335–344.

[111] Apache, “Gremlin query language apache tinkerpop,” accessed: 2021-01-02. [Online]. Available: https://tinkerpop.apache.org/docs/current/tutorials/gremlin-language-variants/

[112] OrientDB, “Open source graph database,” 2010, accessed: 2018-10-01. [Online]. Available: https://www.orientdb.org/

[113] J. Hidders, “Typing graph-manipulation operations,” in *International Conference on Database Theory*. Springer, 2003, pp. 394–409.

[114] W3C, “Resource description framework,” 2021, accessed: 2021-02-27. [Online]. Available: https://www.w3.org/RDF/

[115] P. T. Wood, “Query languages for graph databases,” *ACM Sigmod Record*, vol. 41, no. 1, pp. 50–60, 2012.

[116] C. Stadler, M. Saleem, A.-C. N. Ngomo, and J. Lehmann, “Efficiently pinpointing sparql query containments,” in *International Conference on Web Engineering*. Springer, 2018, pp. 210–224.

[117] D. Figueira, “Reasoning on words and trees with data,” Ph.D. dissertation, École normale supérieure de Cachan-ENS Cachan, 2010.

[118] L. Segoufin, “Automata and logics for words and trees over an infinite alphabet,” in *International Workshop on Computer Science Logic*. Springer, 2006, pp. 41–57.

[119] M. Kaminski and N. Francez, “Finite-memory automata,” *Theoretical Computer Science*, vol. 134, no. 2, pp. 329–363, 1994.

[120] S. Demri and R. Lazić, “Ltl with the freeze quantifier and register automata,” *ACM Transactions on Computational Logic (TOCL)*, vol. 10, no. 3, p. 16, 2009.

[121] F. Neven, T. Schwentick, and V. Vianu, “Finite state machines for strings over infinite alphabets,” *ACM Transactions on Computational Logic (TOCL)*, vol. 5, no. 3, pp. 403–435, 2004.

[122] M. Bojańczyk, A. Muscholl, T. Schwentick, and L. Segoufin, “Two-variable logic on data trees and xml reasoning,” *Journal of the ACM (JACM)*, vol. 56, no. 3, p. 13, 2009.

[123] G. Dodig-Crnkovic, “Scientific methods in computer science,” in *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, 2002, pp. 126–130.

[124] D. Florescu, A. Levy, and D. Suciu, “Query containment for conjunctive queries with regular expressions,” in *PODS*, vol. 9, 1998, pp. 139–148.

[125] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev10, and I. Toma11, “The linked data benchmark council: a graph and rdf industry benchmarking effort,” *SIGMOD Record*, vol. 43, no. 1, p. 27, 2014.

[126] P. Bourhis, M. Krötzsch, and S. Rudolph, “How to best nest regular path queries,” 2014.

[127] J. L. Reutter, M. Romero, and M. Y. Vardi, “Regular queries on graph databases,” *Theory of Computing Systems*, vol. 61, no. 1, pp. 31–83, 2017.

[128] J. Hellings, “Conjunctive context-free path queries.” in *ICDT*, 2014, pp. 119–130.

[129] B. Ten Cate and M. Marx, “Navigational xpath: calculus and algebra,” *ACM SIGMOD Record*, vol. 36, no. 2, pp. 19–26, 2007.

[130] H. S. Kunii, “Dbms with graph data model for knowledge handling,” in *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, 1987, pp. 138–142.

[131] M. Chein and M.-L. Mugnier, *Graph-based knowledge representation: computational foundations of conceptual graphs.* Springer Science & Business Media, 2008.

[132] M.-L. Mugnier and M. Chein, “Conceptual graphs: Fundamental notions,” *Revue d’intelligence artificielle*, vol. 6, no. 4, pp. 365–406, 1992.

[133] R. Angles, “A comparison of current graph database models,” in *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, 2012, pp. 171–177.

[134] A. Ghrab, O. Romero, S. Skhiri, A. Vaisman, and E. Zimányi, “Grad: On graph database modeling,” *arXiv preprint arXiv:1602.00503*, 2016.

[135] A. Ghrab, O. Romero, S. Skhiri, and E. Zimányi, “Analytics-aware graph database modeling,” Technical report, Tech. Rep., 2014.

[136] P. Barceló and P. Muñoz, “Graph logics with rational relations: the role of word combinatorics,” *ACM Transactions on Computational Logic (TOCL)*, vol. 18, no. 2, pp. 1–41, 2017.

[137] M. Levene and A. Poulovassilis, “An object-oriented data model formalised through hypergraphs,” *Data & Knowledge Engineering*, vol. 6, no. 3, pp. 205–224, 1991.

[138] Y. Yu and J. Heflin, “Extending functional dependency to detect abnormal data in rdf graphs,” in *International Semantic Web Conference*. Springer, 2011, pp. 794–809.

[139] Y. A. Megid, N. El-Tazi, and A. Fahmy, “Using functional dependencies in conversion of relational databases to graph databases,” in *International Conference on Database and Expert Systems Applications*. Springer, 2018, pp. 350–357.

[140] S. Finkelstein, M. Schkolnick, and P. Tiberio, “Physical database design for relational databases,” *ACM Transactions on Database Systems (TODS)*, vol. 13, no. 1, pp. 91–128, 1988.

[141] G. Daniel, G. Sunyé, and J. Cabot, “Umltographdb: mapping conceptual schemas to graph databases,” in *International Conference on Conceptual Modeling*. Springer, 2016, pp. 430–444.

[142] O. Hartig and J. Hidders, “Defining schemas for property graphs by using the graphql schema definition language,” in *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, 2019, pp. 1–11.

[143] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu, “Nose: Schema design for nosql applications,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 10, pp. 2275–2289, 2017.

[144] Y. Park, M. Shankar, B.-H. Park, and J. Ghosh, “Graph databases for large-scale healthcare systems: A framework for efficient data management and data services,” in *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 2014, pp. 12–19.

[145] N. Roy-Hubara, L. Rokach, B. Shapira, and P. Shoval, “Modeling graph database schema,” *IT Professional*, vol. 19, no. 6, pp. 34–43, 2017.

[146] I. Comyn-Wattiau and J. Akoka, “Model driven reverse engineering of nosql property graph databases: The case of neo4j,” in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 453–458.

[147] R. C. Anderson, R. J. Spiro, and M. C. Anderson, “Schemata as scaffolding for the representation of information in connected discourse,” *American Educational Research Journal*, vol. 15, no. 3, pp. 433–440, 1978.

[148] R. E. Butts, “Kant’s schemata as semantical rules,” in *Historical Pragmatics*. Springer, 1993, pp. 67–78.

[149] O. Hartig, “Reconciliation of rdf* and property graphs,” *arXiv preprint arXiv:1409.3288*, 2014.

[150] J. Akoka, I. Comyn-Wattiau, and N. Prat, “A four v’s design approach of nosql graph databases,” in *International Conference on Conceptual Modeling*. Springer, 2017, pp. 58–68.

[151] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, “The semantic web,” *Scientific american*, vol. 284, no. 5, pp. 28–37, 2001.

[152] A. Chebotko, A. Kashlev, and S. Lu, “A big data modeling methodology for apache cassandra,” in *2015 IEEE International Congress on Big Data*. IEEE, 2015, pp. 238–245.

[153] T. Jia, X. Zhao, Z. Wang, D. Gong, and G. Ding, “Model transformation and data migration from relational database to mongodb,” in *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 2016, pp. 60–67.

[154] D. Serrano and E. Stroulia, “From relations to multi-dimensional maps: A sql-to-hbase transformation methodology,” in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 2016, pp. 156–165.

[155] V. Herrero, A. Abelló, and O. Romero, “Nosql design for analytical workloads: variability matters,” in *International Conference on Conceptual Modeling*. Springer, 2016, pp. 50–64.

[156] I. Zečević, P. Bjeljac, B. Perišić, S. Stankovski, D. Venus, and G. Ostojić, “Model driven development of hybrid databases using lightweight metamodel extensions,” *Enterprise Information Systems*, vol. 12, no. 8-9, pp. 1221–1238, 2018.

[157] J. Zhang and Y. Tay, “Gscaler: Synthetically scaling a given graph.” in *EDBT*, vol. 16, 2016, pp. 53–64.

[158] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, “Diversified stress testing of rdf data management systems,” in *International Semantic Web Conference*. Springer, 2014, pp. 197–212.

[159] *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*, ISO/IEC Std. 25 010, 2011.

[160] S. Karnouskos, R. Sinha, P. Leitão, L. Ribeiro, and T. I. Strasser, “Assessing the integration of industrial agents and low-level automation functions with iso 25010,” in *Proceedings of the IEEE 16th International Conference on Industrial Informatics (INDIN’18).* IEEE, 2018, pp. 61–66.

[161] R. Lööf and K. Pussinen, “Visualisation of requirements and their relations in embedded systems,” 2014.

[162] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 26, no. 1, pp. 64–69, 1983.

[163] W3C, “Sparql 1.1 query language w3c recommendation,” 2013, accessed: 2021-01-02. [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#pp-language

[164] OpenCypher, “Opencypher,” 2018, accessed: 2018-10-01. [Online]. Available: https://www.opencypher.org/

[165] J. Hellings, B. Kuijpers, J. Van den Bussche, and X. Zhang, “Walk logic as a framework for path query languages on graph databases,” in *Proceedings of the 16th International Conference on Database Theory*, 2013, pp. 117–128.

[166] T. Timm, “Exploration of large-scale sparql query collections: Finding structure and regularity for optimizing database systems,” Ph.D. dissertation, 2020.

[167] L. Zou, L. Chen, and M. T. Özsu, “Distance-join: Pattern match query in a large graph database,” *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 886–897, 2009.

[168] G. O. Arocena, “Weboql: Exploiting document structure in web queries,” Ph.D. dissertation, Citeseer, 1997.

[169] J. Tucker and K. Stephenson, “Data, syntax and semantics,” 2003.

[170] R. D. King, J. Rowland, S. G. Oliver, M. Young, W. Aubrey, E. Byrne, M. Liakata, M. Markham, P. Pir, L. N. Soldatova *et al.*, “The automation of science,” *Science*, vol. 324, no. 5923, pp. 85–89, 2009.

[171] J. Castro and A. Soto, “A comparison between cypher and conjunctive queries.” in *AMW*, 2017.

[172] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," in *International semantic web conference*. Springer, 2006, pp. 30–43.

[173] H. Thakkar, D. Punjani, S. Auer, and M.-E. Vidal, "Towards an integrated graph algebra for graph pattern matching with gremlin (extended version)," *arXiv preprint arXiv:1908.06265*, 2019.

[174] J. L. Reutter, "Containment of nested regular expressions," *arXiv preprint arXiv:1304.2637*, 2013.

[175] X. Zhang, Z. Feng, X. Wang, G. Rao, and W. Wu, "Context-free path queries on rdf graphs," in *International Semantic Web Conference*. Springer, 2016, pp. 632–648.

[176] C. M. Medeiros, M. A. Musicante, and U. S. Costa, "Efficient evaluation of context-free path queries for graph databases," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1230–1237.

[177] J. Kuijpers, G. Fletcher, N. Yakovets, and T. Lindaaker, "An experimental study of context-free path query evaluation methods," in *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, 2019, pp. 121–132.

[178] Y. Susanina, "Context-free path querying via matrix equations," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2821–2823.

[179] J. Pérez, M. Arenas, and C. Gutierrez, "nsparql: A navigational language for rdf," in *International Semantic Web Conference*. Springer, 2008, pp. 66–81.

[180] K. Anyanwu, A. Maduko, and A. Sheth, "Sparq2l: towards support for subgraph extraction queries in rdf databases," in *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 797–806.

[181] K. Anyanwu and A. P. Sheth, "?-queries: enabling querying for semantic associations on the semantic web," in *WWW*, 2003.

[182] K. Anyanwu and A. Sheth, "ρ-queries: enabling querying for semantic associations on the semantic web," in *Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 690–699.

[183] G. H. Fletcher, M. Gyssens, D. Leinders, D. Surinx, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu, "Relative expressive power of navigational querying on graphs," *Information Sciences*, vol. 298, pp. 390–406, 2015.

[184] A. Bonifati, W. Martens, and T. Timm, "Navigating the maze of wikidata query logs," in *The World Wide Web Conference*, 2019, pp. 127–138.

[185] O. Software, “Openlink,” 2019, accessed: 2020-11-29. [Online]. Available: https://www.openlinksw.com/

[186] LSQ, “The linked sparql queries dataset,” accessed: 2020-11-29. [Online]. Available: http://aksw.github.io/LSQ/

[187] Wikidata, “Wikidata:sparql query service/queries/examples,” accessed: 2020-11-29. [Online]. Available: https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples

[188] A. Gubichev, “Query processing and optimization in graph databases,” Ph.D. dissertation, Technische Universität München, 2015.

[189] L. H. Z. Santana and R. dos Santos Mello, “Querying in a workload-aware triplestore based on nosql databases,” in *International Conference on Database and Expert Systems Applications*. Springer, 2019, pp. 159–173.

[190] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, “S2rdf: Rdf querying with sparql on spark,” *arXiv preprint arXiv:1512.07021*, 2015.

[191] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente, “An empirical study of real-world sparql queries,” *arXiv preprint arXiv:1103.5043*, 2011.

[192] A. Bonifati, W. Martens, and T. Timm, “Darql: Deep analysis of sparql queries,” in *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 187–190.

[193] ——, “Sharql: Shape analysis of recursive sparql queries,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2701–2704.

[194] A. Bielefeldt, J. Gonsior, and M. Krötzsch, “Practical linked data access via sparql: The case of wikidata.” in *LDOW@ WWW*, 2018.

[195] J. Oskam and A. Boswijk, “Airbnb: the future of networked hospitality businesses,” *Journal of Tourism Futures*, vol. 2, no. 1, pp. 22–42, 2016.

[196] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda *et al.*, “G-core: A core for future graph query languages,” in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1421–1432.

[197] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi, “Containment of conjunctive regular path queries with inverse,” *KR*, vol. 2000, pp. 176–185, 2000.

[198] N. Schweikardt, T. Schwentick, and L. Segoufin, “Database theory: Query languages,” *Algorithms and theory of computation handbook*, vol. 2, 2009.

[199] P. G. Kolaitis and M. Y. Vardi, “Conjunctive-query containment and constraint satisfaction,” *Journal of Computer and System Sciences*, vol. 61, no. 2, pp. 302–332, 2000.

[200] H. Jiang, H. Wang, S. Y. Philip, and S. Zhou, “Gstring: A novel approach for efficient search in graph databases,” in *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 2007, pp. 566–575.

[201] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, and J. M. Patel, “Saga: a subgraph matching tool for biological graphs,” *Bioinformatics*, vol. 23, no. 2, pp. 232–239, 2007.

[202] L. Bertossi, G. Gottlob, and R. Pichler, “Datalog: bag semantics via set semantics,” *arXiv preprint arXiv:1803.06445*, 2018.

[203] B. Glimm and I. Horrocks, “Handling cyclic conjunctive queries.” in *Description Logics*, 2005.

[204] M. Arenas, P. Barceló, and J. Reutter, “Query languages for data exchange: Beyond unions of conjunctive queries,” *Theory of Computing Systems*, vol. 49, no. 2, pp. 489–564, 2011.

[205] A. O. Mendelzon and P. T. Wood, “Finding regular simple paths in graph databases,” *SIAM Journal on Computing*, vol. 24, no. 6, pp. 1235–1258, 1995.

[206] M. Benedikt, W. Fan, and G. Kuper, “Structural properties of xpath fragments,” *Theoretical Computer Science*, vol. 336, no. 1, pp. 3–31, 2005.

[207] V. Bárány, G. Gottlob, and M. Otto, “Querying the guarded fragment,” in *2010 25th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2010, pp. 1–10.

[208] V. Bárány, M. Benedikt, and B. Ten Cate, “Rewriting guarded negation queries,” in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 2013, pp. 98–110.

[209] S. T. Schweikardt N and S. L, “Database theory: Query languages,” 2001, accessed: 2020-12-02. [Online]. Available: http://www.lsv.fr/~segoufin/Papers/Mypapers/DB-chapter.pdf

[210] G. L. Sanders and S. Shin, “Denormalization effects on performance of rdbms,” in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. IEEE, 2001, pp. 9–pp.

[211] Z. Wei, J. Dejun, G. Pierre, C.-H. Chi, and M. van Steen, “Service-oriented data denormalization for scalable web applications,” in *Proceedings of the 17th international conference on World Wide Web*, 2008, pp. 267–276.

[212] E. F. Codd, "A relational model of data for large shared data banks," in *Software pioneers*. Springer, 2002, pp. 263–294.

[213] A. Khan, Y. Wu, and X. Yan, "Emerging graph queries in linked data," in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 1218–1221.

[214] E. Sciore, M. Siegel, and A. Rosenthal, "Using semantic values to facilitate interoperability among heterogeneous information systems," *ACM Transactions on Database Systems (TODS)*, vol. 19, no. 2, pp. 254–290, 1994.

[215] A. A. Frozza, S. R. Jacinto, and R. dos Santos Mello, "An approach for schema extraction of nosql graph databases," in *2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE, 2020, pp. 271–278.

[216] G. Fitzgerald, A. Philippides, and S. Probert, "Information systems development, maintenance and enhancement: findings from a uk study," *International Journal of Information Management*, vol. 19, no. 4, pp. 319–328, 1999.

[217] M. L. Brodie and J. T. Liu, "The power and limits of relational technology in the age of information ecosystems," in *On the Move Federated Conferences*, 2010.

[218] R. B. Johnson, A. J. Onwuegbuzie, and L. A. Turner, "Toward a definition of mixed methods research," *Journal of mixed methods research*, vol. 1, no. 2, pp. 112–133, 2007.

[219] H. Khalajzadeh, A. Simmons, M. Abdelrazek, J. Grundy, J. Hosking, and Q. He, "An end-to-end model-based approach to support big data analytics development," *Journal of Computer Languages*, p. 100964, 2020.

[220] C. Sharma, R. Sinha, and K. Johnson, "Practical and comprehensive formalisms for modeling contemporary graph query languages," *Information Systems*, p. 101816, 2021.

[221] M. Graves, E. R. Bergeman, and C. B. Lawrence, "A graph-theoretic data model for genome mapping databases," in *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, vol. 5. IEEE, 1995, pp. 32–41.

[222] M. Levene and A. Poulovassilis, "The hypernode model and its associated query language," in *Proceedings of the 5th Jerusalem Conference on Information Technology, 1990.'Next Decade in Information Technology'*. IEEE, 1990, pp. 520–530.

[223] TigerGraph, "A modern graph query language," 2020, accessed: 2020-28-06. [Online]. Available: https://www.tigergraph.com/gsql/

[224] J. J. Marciniak, *Encyclopedia of software engineering (vol. 1 AN).* Wiley-Interscience, 1994.

[225] E. M. Clarke and J. M. Wing, “Formal methods: State of the art and future directions,” *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[226] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, K. W. Hare, J. Hidders, V. E. Lee, B. Li, L. Libkin, W. Martens *et al.*, “Pg-keys: Keys for property graphs,” in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2423–2436.

[227] P. P.-S. Chen, “The entity-relationship model—toward a unified view of data,” *ACM transactions on database systems (TODS)*, vol. 1, no. 1, pp. 9–36, 1976.

[228] A. G. D. of Health, “National bowel cancer screening program,” 2017. [Online]. Available: https://www1.health.gov.au/internet/main/publishing.nsf/Content/nbcsp.htm

[229] Apache, ““apache java library for parsing xls document”,” 2020, accessed: 2021-01-17. [Online]. Available: https://mvnrepository.com/artifact/org.apache.poi/poi

[230] T. Halpin, “Orm 2,” in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems".* Springer, 2005, pp. 676–687.

[231] S. Batra and C. Tyagi, “Comparative analysis of relational and graph databases,” *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, no. 2, pp. 509–512, 2012.

[232] Z. J. Zhang, “Graph databases for knowledge management,” *IT professional*, vol. 19, no. 6, pp. 26–32, 2017.

[233] LDBC, “Ldbc task force,” 2021, accessed: 2021-01-02. [Online]. Available: http://ldbcouncil.org/

[234] Q. Li and Y.-L. Chen, “Entity-relationship diagram,” in *Modeling and Analysis of Enterprise and Information Systems.* Springer, 2009, pp. 125–139.

[235] J. Rumbaugh, I. Jacobson, and G. Booch, “The unified modeling language,” *Reference manual*, 1999.

[236] ACATECH, “Cyber-Physical Systems: Driving force for innovation in mobility, health, energy and production,” ACATECH – German National Academy of Science and Engineering, Tech. Rep., Dec. 2011.

[237] E. A. Lee, “Cyber physical systems: Design challenges,” in *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC).* Institute of Electrical & Electronics Engineers (IEEE), May 2008.

[238] P. Leitão, A. W. Colombo, and S. Karnouskos, “Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges,” *Computers in Industry*, vol. 81, pp. 11–25, Sep. 2015.

[239] P. Leitão, “Agent-based distributed manufacturing control: A state-of-the-art survey,” *Engineering Applications of Artificial Intelligence*, vol. 22, no. 7, pp. 979–991, Oct. 2009.

[240] P. Leitão, S. Karnouskos, L. Ribeiro, P. Moutis, J. Barbosa, and T. I. Strasser, “Common practices for integrating industrial agents and low level automation functions,” in *Proceedings of the 43rd Annual Conference of the IEEE Industrial Electronics Society (IECON’17)*. IEEE, 2017, pp. 6665–6670.

[241] S. Karnouskos, R. Sinha, P. Leitão, L. Ribeiro, and T. I. Strasser, “The Applicability of ISO/IEC 25023 Measures to the Integration of Agents and Automation Systems,” in *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, Oct 2018, pp. 2927–2934.

[242] S. Morimoto, D. Horie, and J. Cheng, “A security requirement management database based on iso/iec 15408,” in *International Conference on Computational Science and Its Applications*. Springer, 2006, pp. 1–10.

[243] S. Morimoto and J. Cheng, “A security specification library with a schemaless database,” in *International Conference on Computational Science*. Springer, 2007, pp. 890–893.

[244] R. Hull and R. King, “Semantic database modeling: Survey, applications, and research issues,” *ACM Computing Surveys (CSUR)*, vol. 19, no. 3, pp. 201–260, 1987.

[245] R. Sinha, S. Patil, C. Pang, V. Vyatkin, and B. Dowdeswell, “Requirements engineering of industrial automation systems: Adapting the cesar requirements meta model for safety-critical smart grid software,” in *Industrial Electronics Society, IECON 2015-41st Annual Conference of the IEEE*. IEEE, 2015, pp. 002 172–002 177.

[246] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, “On supporting containment queries in relational database management systems,” in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, 2001, pp. 425–436.

[247] M. Jarrah, B. Al-khatieb, N. Mahasneh, B. Al-khateeb, and Y. Jararweh, “Gdbapex: A graph-based system to enable efficient transformation of enterprise infrastructures,” *Software: Practice and Experience*, vol. 51, no. 3, pp. 517–531, 2021.

[248] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, 1974, pp. 249–264.

[249] P. Gulutzan and T. Pelzer, *SQL-99 complete, really*. CMP books, 1999.

[250] P. Boncz, I. Fundulaki, A. Gubichev, J. Larriba-Pey, and T. Neumann, "The linked data benchmark council project," *Datenbank-Spektrum*, vol. 13, no. 2, pp. 121–129, 2013.

[251] A. Silberschatz, M. Stonebraker, and J. D. Ullman, "Database systems: Achievements and opportunities," *ACM Sigmod Record*, vol. 19, no. 4, pp. 6–22, 1990.

[252] S. Chanda and D. Foggon, *Beginning ASP. NET 4.5 Databases*. Apress, 2013.

[253] J. Rachapalli, V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham, "Retro: a framework for semantics preserving sql-to-sparql translation," *The University of Texas at Dallas*, vol. 800, pp. 75 080–3021, 2011.

[254] B. A. Steer, A. Alnaimi, M. A. Lotz, F. Cuadrado, L. M. Vaquero, and J. Varvenne, "Cytosm: Declarative property graph queries without data migration," in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, 2017, pp. 1–6.

[255] M. N. Mami, D. Graux, H. Thakkar, S. Scerri, S. Auer, and J. Lehmann, "The query translation landscape: a survey," *arXiv preprint arXiv:1910.03118*, 2019.

[256] A. Halverson, V. Josifovski, G. Lohman, H. Pirahesh, and M. Mörschel, "Rox: relational over xml," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 264–275.

[257] T. Wilmes, "Gremlin to sql," 2016, accessed: 2020-03-01. [Online]. Available: https://github.com/twilmes/sql-gremlin

[258] T. H. D. Araujo, B. T. Agena, K. R. Braghetto, and R. Wassermann, "Ontomongo-ontology-based data access for nosql." *ONTOBRAS*, vol. 1908, pp. 55–66, 2017.

[259] A. Chebotko, S. Lu, H. M. Jamil, and F. Fotouhi, "Semantics preserving sparql-to-sql query translation for optional graph patterns," Citeseer, Tech. Rep., 2006.

[260] N. Bikakis, N. Gioldasis, C. Tsinaraki, and S. Christodoulakis, "Querying xml data with sparql," in *International conference on database and expert systems applications*. Springer, 2009, pp. 372–381.

[261] H. Thakkar, D. Punjani, Y. Keswani, J. Lehmann, and S. Auer, "A stitch in time saves nine–sparql querying of property graphs using gremlin traversals," *arXiv preprint arXiv:1801.02911*, 2018.

[262] M. Droop, M. Flarer, J. Groppe, S. Groppe, V. Linnemann, J. Pinggera, F. Santner, M. Schier, F. Schöpf, H. Staffler *et al.*, “Translating xpath queries into sparql queries,” in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2007, pp. 9–10.

[263] W. Fan, J. X. Yu, H. Lu, J. Lu, and R. Rastogi, “Query translation from xpath to sql in the presence of recursive dtds,” in *VLDB*, 2005, pp. 337–348.

[264] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie, “Sql-graph: An efficient relational-based property graph store,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1887–1901.

[265] S. Ceri and G. Gottlob, “Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries,” *IEEE Transactions on software engineering*, no. 4, pp. 324–345, 1985.

[266] E. Meijer and G. Bierman, “A co-relational model of data for large shared data banks,” *Communications of the ACM*, vol. 54, no. 4, pp. 49–58, 2011.

Appendix A

Introduction to Manuscript 4

The ongoing fourth Industrial Revolution depends mainly on robust Industrial Cyber-Physical Systems (ICPS). ICPS includes computing (software and hardware) abilities to control complex physical processes in distributed industrial environments. Industrial agents, originating from the well-established multi-agent systems field, provide complex and cooperative control mechanisms at the software level, allowing us to develop larger and more feature-rich ICPS. The IEEE P2660.1 standardisation project, "Recommended Practices on Industrial Agents: Integration of Software Agents and Low Level Automation Functions" focuses on identifying Industrial Agent practices that can benefit ICPS systems of the future. A key problem within this project is identifying the best-fit industrial agent practices for a given ICPS. This paper reports on the design and development of a tool to address this challenge. This tool, called `IASelect`, is built using graph databases and provides the ability to flexibly and visually query a growing repository of industrial agent practices relevant to ICPS. `IASelect` includes a front-end that allows industry practitioners to interactively identify best-fit practices without having to write manual queries.

Appendix B

IASelect: Finding Best-fit Agent Practices in Industrial CPS Using Graph Databases (Manuscript 4)

B.1 Introduction

Industrial Cyber-physical systems (ICPS) are seen as a core ingredient in the 4th Industrial Revolution [236], which complemented with emergent ICT technologies, such as Internet of Things, Big Data, Cloud Computing and Data Analytics, promotes the deployment of more interoperable, flexible, responsive and reconfigurable devices and systems. ICPS contain deep integration of computational applications with physical automation devices and are designed as networks of interacting cyber and physical elements [237, 238, 236]. Each component of an ICPS integrates its physical hardware function with a software (cyber) application acting as a virtual representation of its tangible counterpart.

The Multi-Agent Systems paradigm, derived from distributed artificial intelligence, promotes distribution, decentralization, intelligence, autonomy and adaptation, contributing to achieve flexibility, robustness, responsiveness and re-configurability [239]. This paradigm provides a fundamentally different way to design complex control systems based on the distribution of intelligence and decentralization of control functions over distributed autonomous and cooperative entities, called agents. Used in industrial contexts, agents, or more specifically Industrial Agents, can help to develop highly adaptive ICPS. In industrial environments, and aligned with the ICPS principles, the interconnection of intelligent software agents with the automation control devices, e.g., robots and PLCs (Programmable Logic Controllers), assumes a crucial role. Usually, this interconnection is created in a proprietary, case-by-case, and ad hoc manner. However, the use of a standardized way to implement this interface can help achieve transparency, interoperability, and scalability.

The IEEE P2660.1 standardization project, "Recommended Practices on Industrial Agents: Integration of Software Agents and Low Level Automation Functions", has been working on a methodology to rank and select best-fit Industrial Agent practices for the interfacing between software agents and automation control devices. Previous work was devoted to identifying the patterns derived from a survey of existing implementations of industrial agents [240], and to assess their characteristics, using the ISO/IEC 25010 standards family as a starting point [160].

This paper describes the design and development of a tool called `IASelect` for implementing the methodology to select recommended interfacing practices. `IASelect` uses a graph database to store interfacing practices templates and their technological instantiations along with their characterization according to a set of quality criteria. It provides a front-end for users to interactively retrieve the best interface practice for a particular application scenario. In particular, the primary contributions of this paper are:

1. The design and development of a graph database to store the available data on

Industrial Agent practices. This approach provides several benefits including, better data governance, data visualization, and interactive querying. A summary of available industrial agent practices is discussed in Sec. B.2 and the design of the graph database is presented in Sec. B.3.

2. The creation of query patterns and templates to allow industrial practitioners to use and query graph databases more easily. These patterns allow more flexibility than more static mechanisms like forms and spreadsheets. We discuss the design and development of these patterns in Sec B.3.

3. An implementation of the proposed graph database and query patterns using Neo4j and Java into a tangible tool called `IASelect`. This GUI-based tool can be used by both users and administrators to identify practices and/or manage the knowledge base. The implementation is presented in Sec. B.4.

B.2 Background

Software agents can work with low-level automation functions in a variety of ways. A survey of commonly-encountered practices helped the P2660.1 working group to develop a set of generic interface practices clustered according to two dimensions, as illustrated in Fig. B.1 [2]. *Coupling*, shown on the X-axis, is dependent on the integration between high-level control (agents) and low-level control. Tight coupling indicates a direct and permanent coupling, as in the use of remote procedure calls. Loose coupling involves a mediated connection, such as through a queue. The Y-axis pertains to the *location* of the agent. Agents can be *on-device*, where they run on the same controller as the low-level functions. *Hybrid* systems have agents running externally rather than on the same controller. The survey carried out by the working group classified available practices into four primary interface practices: *Tightly Coupled–Hybrid*, *Tightly Coupled–On-device*, *Loosely Coupled–Hybrid* and *Loosely Coupled–On-device*. Each one of the generic interfacing practices shown in Fig. B.1 can be instantiated using several different technologies.

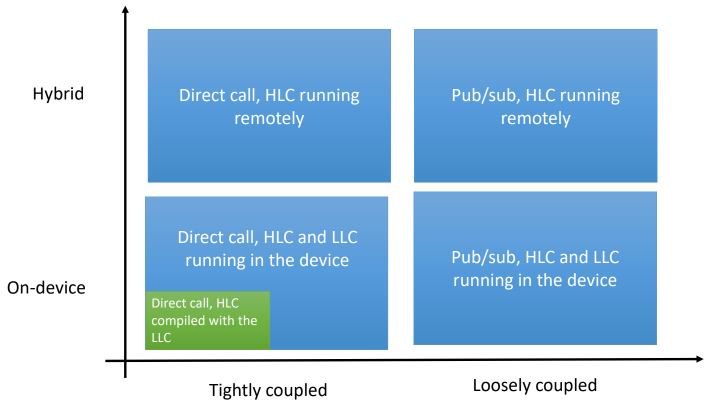

Hybrid
On-device
Direct call, HLC running remotely
Pub/sub, HLC running remotely
Direct call, HLC and LLC running in the device
Direct call, HLC compiled with the LLC
Pub/sub, HLC and LLC running in the device
Tightly coupled
Loosely coupled

Figure B.1: Interface patterns considering interaction mode and location levels of abstraction. [2]

Each practice has an associated set of qualities or characteristics, which make it more suitable for use in specific contexts. Selecting a best-fit interface practice for a given system context, therefore, requires identifying these associated qualities for each practice. The P2660.1 working group used the comprehensive yet generic set of characteristics from ISO/IEC 25010 [159], formerly ISO/IEC 9126, as a starting point to differentiate between practices. The ISO/IEC 25010 standard groups system qualities into eight characteristics. Each characteristic is then further separated into multiple sub-characteristics. Subsequently, a survey was conducted with a team of experts in the domain to identify qualities that are most relevant to the field [160]. The survey found that *testability, availability, time behavior, interoperability, availability, fault-tolerance* and *reusability* emerged as the most important characteristics for the practices. Further work conducted by the working group showed how specific measures from the standard could be used to evaluate the practices [241]. In this paper, the characterization of practices and implementations is extended into a tool that allows stakeholders, especially industry experts, to identify best-fit practices through the qualities that are most desirable in their context.

There are existing tools used for managing and querying data sets for domains similar to ICPS. For instance, in [242] authors discuss an approach to store information related to security standards in relational databases and Structured Query Language (SQL) is used for data retrieval. In [243], authors have discussed an approach to store security requirements in a schema less XML database. However, a database with no schema based restrictions has higher risk of data corruption. In [161], authors have presented a tool to visualize requirements, and Neo4j database has been used to maintain the graph structure. Similar kind of tools have also been used in other domains for example in [6] where authors have proposed a tool for storing chemical compounds as graphs and graph algorithms are used to search chemical structures over the database. In [95], authors have proposed an approach based on graph databases for genome sequencing. The approaches discussed so far, except [242] use graph theory concepts to handle and inquiry data. In this paper, we present a tool that uses a graph database to store the P2660.1 data set and, automates the analysis of existing interfacing practices on user-defined selection criteria.

B.3 Designing a Graph Database for Selecting Industrial Agent Practices

As data size increases, managing data with traditional tools such as spreadsheets becomes a complicated task. Based on the law of entropy, an increase in data size also means that over time, disorder in the data set will increase. Spreadsheets are too cumbersome to maintain, primarily when shared and used by multiple stakeholders, such as users and administrators simultaneously. Database management systems serve as an alternative for organizing large data sets. Furthermore, they assist in correlating and analyzing collected data.

B.3.1 Rationale for using Graph Databases

Rational database management systems (RDBMS) are the most popular tools for managing data. They have proven to be persistent in providing concurrency control and integration mechanism for data since 1970s [68]. RDBMS are highly efficient in handling large data banks. However, RDBMS have limited ability to capture the overall semantics of a domain [244, 68]. Moreover, as the number of relationships between data grows, RDBMS become inefficient in managing and querying data [67]. On the other hand, Graph databases (GDBs) are gaining wide acceptance in the industry due to there application in domains that deal with the querying and analysis of connected data [68, 67, 23, 27, 25, 125]. A graph database contains nodes and edges where nodes represent the entities and edges represent relationships between the entities [23, 67]. Together, nodes and edges capture the overall semantics of the domain. The resulting structure is more straightforward and is at the same time more expressive than those produced by RDBMS and Not Only SQL (NO-SQL) databases such as wide-column stores, document stores and key-value stores [133, 8].

For searching data, a spreadsheet or a RDBMS performs a search and match operation. This operation represents a relational join between different tables to calculate relationships at the time when a query is running. This operation tends to be computationally expensive in highly interconnected data-sets. Graph databases are more efficient in such cases as the relationships between data are created at database creation stage and are stored inside the database. Hence, the overhead of calculating relationships at the time when data is being retrieved from the database is minimized in graph databases.

Graph database solutions such as Neo4j are based on the property graph data model [68, 27]. A property graph data model is more expressive than other graph data models, such as the resource description framework (RDF) [114]. A property graph stores information inside nodes and edges as key-value pairs which means that information can be embedded inside relationships which is an advantage over the RDF data model.

Another advantage of using graph databases is that they scale well. A property graph data model supports multi-graphs where two nodes can be connected via multiple edges with each edge containing separate information about the relationship between the two nodes. Adding more information, therefore, does not require a refactoring or restructuring of the database. Current graph database solutions such as Neo4j are schema optional [38], which means that the graph database can easily accommodate any structural changes. While there are higher chances of data corruption, this risk can be mitigated by enforcing integrity constraints and writing additional logic in a programming language like Java or Python. The use of such integrity constraints at the database creation stage ensures data integrity and data consistency.

B.3.2 A Graph Database for Industrial Agent practices

The P2660.1 data set relates each interfacing practice to a score for specific system qualities, some of which come from ISO/IEC 25010. Table B.1 shows the list of these qualities. This mapping was represented as a two-dimensional adjacency matrix where each mapping between a practice and a sub-characteristic was assigned a value, called its *weight*. The adjacency matrix can be visualized as a graph where interfacing practice and sub-characteristics are represented as nodes. Furthermore, the edge between a practice and a sub-characteristic is labeled with the appropriate score for that relationship.

Table B.1: System qualities mapped in the P2660.1 data set

Domain	**Function**	**Maintenance**	**Performance Efficiency**
Factory Automation Building Automation Energy	Monitoring Control Simulation	Re-usability Capacity To Host agents	Time behaviour Scalability

Graph Schema for P2660.1 data set

Creating a graph database requires information about how data can be connected and structure. This information is called a *graph schema*. A graph schema provides an general view of the entire database by capturing its topology. Intuitively, nodes and edges of the graph schema represent the node, and edge types of the graph database. Node and edge types also assist in grouping together the nodes and edges of the graph database later for searching and visualization.

Fig. B.2 shows the graph schema constructed using the P2660.1 data set. The schema is a labeled directed graph where the node types represent the relevant characteristics from Table B.1, as well as the two possible location levels (OnDevice and Hybrid). The graph schema allows storing the weights on edges that reflect the scores as per the P2660.1 data set. The direction of an edges shows that there exists a weighted mapping from a practice to a characteristic. For example, in the graph schema in Fig. B.2 there is an outgoing edge from hybrid practice and an incoming edge to maintenance.

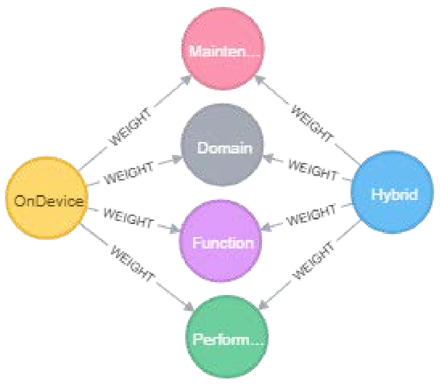


Figure B.2: Graph Schema for storing information about practices

A Graph Database for the P2660.1 data set

A graph database (GDB) is instantiated from its graph schema. Fig. B.3 represents the graph database for the P2660.1 data set, instantiated from the graph schema in Fig. B.2. Information related to nodes and edges of the graph database is contained as attributes of these elements and are stored as key-value pairs. For example, the node with name $OT:1$ represents a OnDevice tightly technique where the attributes key $apiClient$ has a value $java$ assigned to it. Similarly, there are edges between nodes where attribute key $weight$ has a value assigned to it.

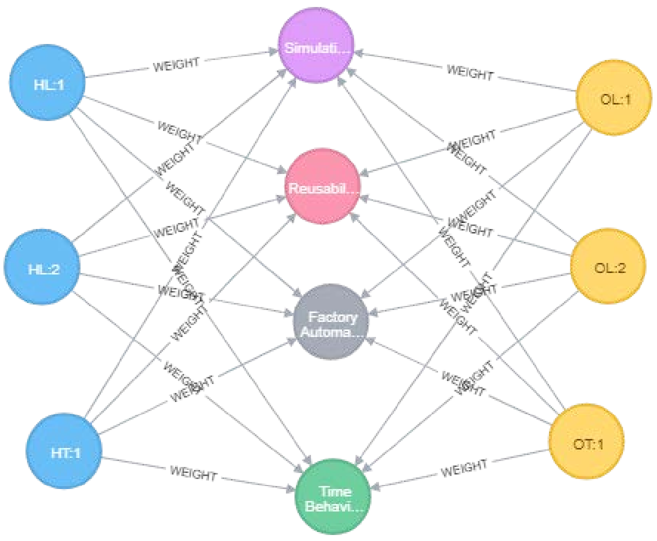


Figure B.3: Sample Graph Database representation of P2660.1 data set

B.3.3 Querying the P2660.1 Graph Database

In this research, we are using Neo4j as a graph database for storing data. To retrieve data from the database we need to define the kind of data we want to extract. Neo4j provides a declarative query language called Cypher [38] to retrieve data from the graph database. Searching a graph database requires defining a sub-graph as a pattern to look for within the database. This sub-graph is expressed using a *graph pattern*.

A graph pattern assists in defining the sub-graph of interest so that similarly structured data can be extracted from the database. Structurally, a graph pattern is expressed as a sub-graph of the graph schema. A pattern matching algorithm then uses the graph pattern to search for the sub-graph over the graph database. For example, by referring to the graph schema as in Fig. B.2 one can search for queries such as find all hybrid techniques for factory automation domain which have been assigned weight greater than 2. Such a query can be expressed as a graph pattern in Cypher as follows:

QUERY 29: Example of searching a graph pattern in the P2660.1 data-set

```
MATCH (h:Hybrid)-[w:WEIGHT]->(d:Domain)
WHERE w.value > 2
AND d.name = "Factory Automation"
RETURN *
```

The graph pattern is expressed in the `MATCH` clause of the query. A graph pattern consists of node/edge types and node/edge variables. The node/edge types assist in specifying the type of data, and the node/edge variables assist in accessing the node/edge attributes of the graph database. The `MATCH` clause uses a pattern matching algorithm to find all the matching sub-graphs in the graph database. The sub-graphs are further restricted based on the filter conditions specified in the `WHERE` clause. Filter conditions are set based on the attribute value stored in the database and are designed using the node/edge variables. Finally, the `RETURN` clause outputs the sub-graph shown in Fig. B.4 and marks the end of the query.

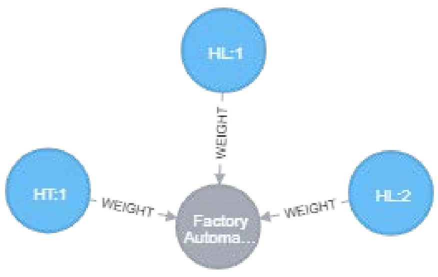


Figure B.4: Result of running Query 29 on the Graph Database shown in Figure B.3

B.4 Implementation

Graph databases are at an early stage of industry-wide adoption. Moreover, users working in domains such as cyber-physical systems and multi-agent systems may not be familiar with graph database query languages like Cypher. Therefore, we have developed a tool `IASelect` that assists in querying graph databases without requiring a working knowledge of Cypher.

B.4.1 **`IASelect`- Architecture**

`IASelect` must feature several important qualities. We use the terminology from ISO/IEC 25010 to list the following system characteristics:

- *Functional suitability:* Functionally, `IASelect` must provide features such as the ability for administrators to manage the underlying database, and the ability for users to query the database to rank available practices that are relevant to their context.
- *Usability:* `IASelect` must be highly usable for both administrators and users. It must allow users to enter information interactively and provide appropriate user error protection, and also present the results clearly. The tool must be accessible for multiple users from different sub-domains of industrial control.
- *Availability:* `IASelect` must be accessible to multiple users, possibly present in different locations, at the same time.
- *Portability:* `IASelect` should be independent of the users' computer configurations.

Functional suitability is supported through the design of the database, as described in Sec B.3, which allows all desired features to be included within the tool. The

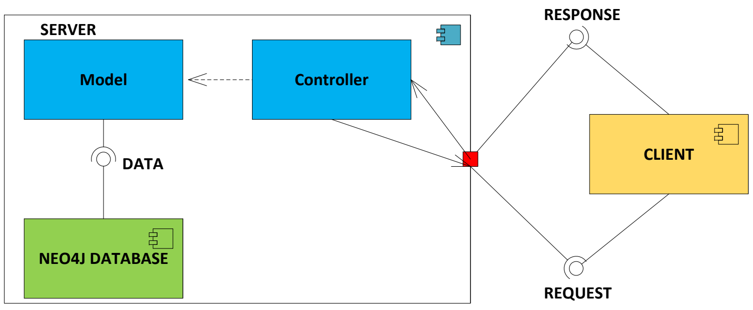


Figure B.5: Component Diagram for `IASelect`

architecture of `IASelect`, shown in Fig. B.5, supports all other characteristics.

To achieve high *availability*, `IASelect` is based on a client-server architecture. The client side is a web-page that can be run on most machines (supporting *portability*). The server runs both the Neo4j database and an application to handle requests from the client. Decoupling the server side from the client's machine allows us to (a) allows users to use `IASelect` without installing any new software such as Neo4j, (b) control the server side for both privacy and performance, and (c) allow for easily modifying or scaling the server or the client side without affecting the other. The application server provides a restful web service so that users can query graph database over the web. Currently, the server application and database are deployed on a cloud data center.

For *usability*, which is a primary characteristic of `IASelect`, we embed a Model-View-Controller (MVC) design pattern inside the client-server architecture. The client-side web-page contains the View which can change depending on who the user is. Currently, we support two views: the administrator view and user view. Administrators can to update the database while users can only query it. At the time of writing, only the user view has been fully integrated into `IASelect`. The Model and Controller are java class objects which run on the server side application. The Controller class object handles the conversion of requests from a View into Cypher queries, and the Model class runs the queries on the Neo4j database. Using the MVC pattern support scalability and enables the addition of Views (for additional user types) easily. It also ensures that the code base is cleaner and understandable, making it easier to maintain.

The client web-page further improves *usability* by providing drop-down lists (for reducing user error) for users to select appropriate context-specific qualities and metrics. The results returned from the server side application are then displayed as a ranked list which can easily be understood.

B.4.2 Software Implementation of `IASelect`

Technology Stack

The server side has been implemented in Java and integrated with Gradle and Maven. Gradle is used for build automation. We have added the Spring boot plugin to the Gradle project to provide an embedded Tomcat server to host the server side application. We use the Maven libraries to connect the Java project with the Neo4j database. The database queries written in Cypher are embedded inside the server side application's Java code and are executed through the appropriate method call. The client side is a web-page that is built using HTML5, CSS3, and Javascript. We have used AJAX to communicate between the client and the server using the XMLHttpRequest.

Transaction Sequence

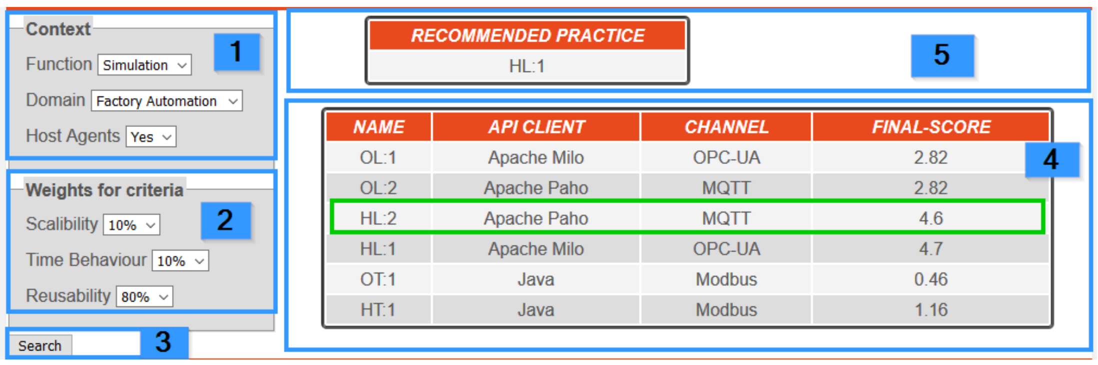
The web-page running on the client machine is accessed through a URL. The tool assists users in extracting data from the graph database based on the specified criteria. `IASelect` generates a report which lists all the matching practices and recommends the most suitable practice. For generating a practice report in `IASelect`, users are presented with a web form. The web form serves as a boilerplate [245] to specify the criteria for generating a practice report. Boilerplates are semi-complete query structures that can be completed through user input.

When a request is submitted, the controller object running within the server side application receives the request. At the same time, the application establishes a connection with the Neo4j database instance using the Bolt protocol. Bolt is a TCP based network protocol which is integrated into Neo4j for connecting to other applications. Once the connection between the database and the model has been established, the controller passes the request to the model by calling the appropriate method. The model then requests a session with the Neo4j database instance and sends a query written in Cypher to be executed at the Neo4j database. The request parameters received from the client are embedded inside the Cypher query. The model then returns the query results obtained from Neo4j database to the controller. The controller then passes the result-set to the server and, finally the server sends the result-set back to the client in the form of a response. At the client, the response is further processed and is displayed on the web page.

Tool Usage

The user provides the necessary context-specific details using the following steps. In the first step, a user sets the context of search by specifying relevant qualitative requirements that the Industrial practices must fulfill. This is done by selecting the sub-characteristics related to function and domain as listed in Tab. B.1. The sub-characteristics defines the application context for the interface practice. Furthermore, the user also specifies if the practice should be capable to host agents. For example, as shown in section 1 of Fig. B.6, the context is set for searching practices for factory automation domain, simulation function and the practice should be capable to host agents. A practice report can be generated for other sub-characteristics of function and domain by using the drop down menu in the web form.

In the second step, the user sets criteria based on maintenance and performance efficiency related to the practices. Users specify which sub-characteristics are deemed most relevant in their context. For determining the relevance of sub-characteristics a



Context

Function	Simulation
Domain	Factory Automation
Host Agents	Yes

Weights for criteria

Scalibility	10%
Time Behaviour	10%
Reusability	80%

RECOMMENDED PRACTICE

HL:1

NAME	API CLIENT	CHANNEL	FINAL-SCORE
OL:1	Apache Milo	OPC-UA	2.82
OL:2	Apache Paho	MQTT	2.82
HL:2	Apache Paho	MQTT	4.6
HL:1	Apache Milo	OPC-UA	4.7
OT:1	Java	Modbus	0.46
HT:1	Java	Modbus	1.16

Figure B.6: Web form based client interface of `IASelect`

percentage scale is assigned on the weights between practices and sub-characteristics related to maintenance and performance efficiency. For example, as shown in section 2 of Fig. B.6, scalability, time-behaviour and re-usability are set with percentage scale of 10, 10 and 80 respectively (the total must be 100). In this particular scenario, the user clearly prefers a practice with high level of re-usability, and that scalability and time-behaviour are of lower relevance.

Finally, based on the context and criteria, a practice report is generated which displays a list of matching practices with technique name, API client, channel and final score assigned to each practice. The final score is calculated by multiplying cumulative percentage weight for each practice with a respective average weight between practice and particular function sub-characteristics. For example, as shown in section 4 of Fig. B.6 technique *HL:2* has *Apache Milo*, *MQTT* and, *4.6* as API client, channel and, final score, respectively.

The recommended practice as highlighted in section 5 of Fig. B.6, corresponds to the practice that got the highest final score, for example, *HL:1* is the recommended practice for this scenario. The tool also provides the list of alternative interface practices sorted based on the score values.

B.5 Conclusions and Future Directions

This paper presents the construction of a graph database tool, called `IASelect`, to allow industry practitioners to identify best-fit industrial agent practices for industrial CPS. `IASelect` is easy to use and, its architecture enables scalability and flexibility. For instance, the edges of the graph database currently contain the weights between practices and sub-characteristics, which makes the database equivalent to a spreadsheet table. In the future, additional properties can be added to nodes and edges without altering the topology of the graph. Such scalability and flexibility are not present in spreadsheets.

The front end of `IASelect` enables users to query graph databases without having a working knowledge of query languages like Cypher. `IASelect` uses a boilerplates based approach that enables users to query the graph database. Furthermore, the boilerplate based approach is not limited to the P2660.1 data-set and can be extended to domains other than ICPS. `IASelect` has been deployed in the cloud as a restful web service. This enables other users to access data related to industrial agent practices via Restful web API. Furthermore, users can integrate the web service into their own applications. Deploying `IASelect` in the cloud also provides advantages specific to cloud computing technology such as scalability, availability, reliability and security.

`IASelect` is an attempt to harness the potential of property graph databases in the domains such as ICPS. However, currently we are only partially utilizing the power of graph database query languages. Graph databases enable users to identify, search and, extract patterns from data. Users can specify a sub-graph of interest to search all similar occurrences of sub-graph in graph database. In `IASelect` however, we are searching for very specific patterns which are tailor made to meet the requirements from the P2660.1 standard and it cannot yet be used to search for generic patterns. In the future, `IASelect` also needs to feature an administrator view for inserting new data, and updating and deleting data from the database.

Appendix C

Introduction to Manuscript 5

With the influx of Web 3.0 the focus in Big Data Analytics has shifted towards modelling highly interconnected data and analysing relationships between them. Graph databases befit the requirements of Big Data Analytics yet organizations still depend on relational databases. A major roadblock in the industry wide adoption of graph databases is that a standard query language is still in its inception stage hence withholding interoperability between the two technologies. In this research we propose a tool FLUX for translating relational database queries to graph database queries.

Appendix D

FLUX: From SQL to GQL query translation tool (Manuscript 5)

D.1 Motivation for the study

D.1.1 Relational Databases and query languages

Relational databases organize data in multiple tables which consist of rows and columns. In order to extract information (*or query*) [246] a relation database multiple tables have to be combined together with a search and match operation [16, 247]. Moreover, the search and match operation called as a *join*, is performed every time a database is queried. For querying relational databases use the *de facto* Structured Query Language (SQL) [248]. SQL is being updated consistently [249] and has been accepted as an ISO standard query language[1].

[1] `https://www.iso.org/standard/16661.html`

D.1.2 Graph databases and query languages

A graph database consists of nodes and edges where nodes are used to store data and relationships or interactions between nodes are stored as edges of the graph database [43, 28, 7, 8]. Storing relationships as edges means that relationships between data are computed at the database creation stage. This means that unlike relational databases, querying graph database is computationally less expensive with respect to the search and match operations.

A major disadvantage of current graph database solutions is that there is no standard query language like SQL is for relational databases. This lack of standard query language has resulted in initiatives such as *Linked Data Benchmark Council* (LDBC) [125, 250], *OpenCypher* [164] and ISO/IEC 39075[2] which are working towards creating a standard query language (GQL) for graph databases. The GQL manifesto[3] proposed by ISO/IEC 39075 aims to integrate features from current graph query languages such as Cypher[4], PGQL[5], GSQL[6],SQL/PG[7] and G-Core [196].

D.1.3 Research problem

Relational databases have been around since the 1970s [251] and their importance is unlikely to diminish [252]. However, they are becoming incompetent in handling the size and complexity of large data sets in the current age of Big Data Analytics [4, 5]. Big Data Analytics projects still rely heavily on relational databases. Graph databases suit Big Data as they provide a better alternative for handling highly interconnected data sets [7, 8]. However, lack of a standard query language has hindered the adoption

[2]`https://www.iso.org/standard/76120.html`

[3]`https://www.gqlstandards.org/existing-languages`

[4]`https://neo4j.com/developer/cypher-query-language/`

[5]`https://pgql-lang.org/spec/1.2/`

[6]`https://www.tigergraph.com/gsql/`

[7]`https://www.w3.org/Data/events/data-ws-2019/assets/lightning/OskarVanRest.pdf`

of graph database technology by relational database practitioners. In order to facilitate an easy migration from relational to graph databases in this research we propose a tool **FLUX** for translating relational database queries to graph database queries. Particularly, this research aims to answer the following research questions:

- What are the existing formalisms for translating relational database queries into graph database queries?
- Which query translation methods can be used to build a tool that translates SQL queries into GQL queries?

Since GQL is still in its inception stage we consider two of the most highly adopted languages in academia and industry namely Cypher and PGQL in the initial development stages of **FLUX**. Moreover, these languages are part of the GQL manifesto.

D.2 Background and Related work

In this section we discuss various query translations studies that have been conducted in the past amongst several query languages for relational and graph database. Figure 1 represents a visual representation, a graph where nodes represent query languages and directed relationships represent the translation direction. The relationships are labeled with a reference to the research papers that discuss the translation.

For example, a directed edge between the nodes SQL and SPARQL means that a study exists that discusses the translation of a SQL query to SPARQL. The edge label [253] represents a reference to citation in the bibliography. Similarly, authors in [254] discuss the translation of queries in Cypher to queries in SQL. In [255, 253, 256, 257] authors discuss the translation of SQL queries into query languages such as XPATH, DOCUMENT based, SPARQL and Gremlin respectively. In [258, 259, 260, 261] authors discuss the translation of SPARQL queries into DOCUMENT based, SQL,

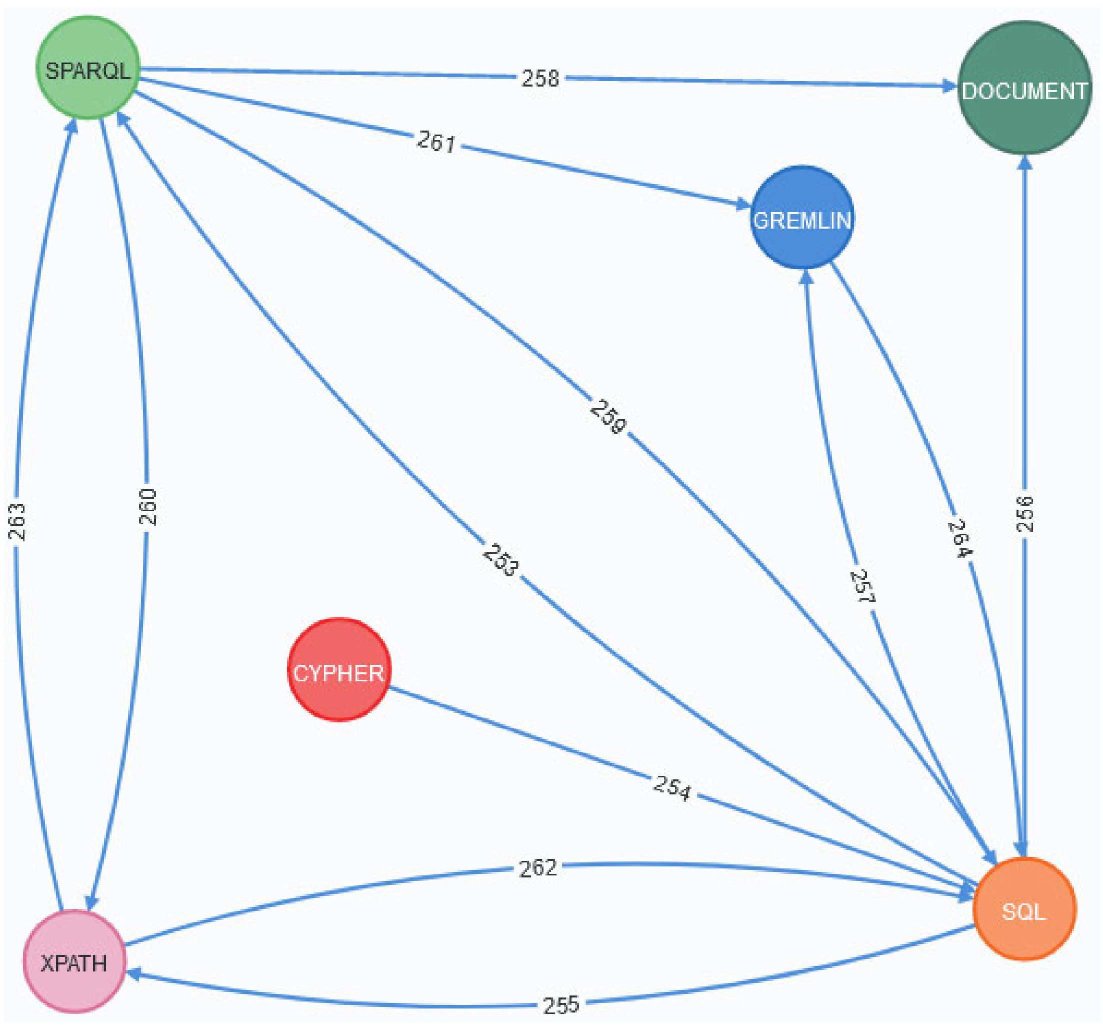


Figure D.1: A graph depicting query translation studies conducted between various database query languages

XPath and Gremlin queries. In [262, 263] authors discuss the translation of XPATH queries into SQL and SPARQL. In [264] authors discuss the translation of Gremlin queries into SQL. Based on the existing literature and as shown in Figure 1 it is evident that there is a lack of research that discusses the translation of SQL queries into graph query languages such as Cypher and PGQL hence this brief review suggests novelty in our study.

D.3 Research Methodology

We take a formal approach to study query languages for relational and graph databases. We define formal semantics for language translation as doing so assures reproducibility and easy exchange of queries [170, 169].

D.3.1 Database schema

In order to efficiently translate a relational query into a graph query the underlying relational and graph data models need to follow similar schema. We use the schema based data model translation approach proposed in [15] to assist this study.

D.3.2 Query language formalisms

Relational database query languages

SQL for relational database is based on relational algebra [265, 266] which is a high level procedural language. Relational algebra uses operations such as join, selection, projection, cartesian product, aggregations and outer join[212].

Graph database query languages

Graph database query languages are broadly divided into two categories (i)*graph pattern matching* and (ii) *graph navigation*.

- *Graph pattern matching* queries use the formalism of *conjunctive queries*[32]. Moreover, relational algebra operations such as join, selection, projection, aggregations and outer join are also used.

- *Graph navigation* queries use the formalism of Regular Path Queries (RPQ) and their extensions [56, 197].

Relational database query languages and *graph pattern matching* have some overlaps. Furthermore, a formalism based on Tarski's algebra has been studied for *graph navigation* which subsumes relational algebra and can be represented as semi join algebra [74, 55]. Based on the brief discussion in this section we can observe some common characteristics between query languages for relational and graph databases that can assist in query translation.

D.3.3 Prototype of **FLUX**

In order to build **FLUX** we use prototype approach where we apply findings from the query formalisms to build a smaller version of **FLUX** with limited features. And more features and support for other query languages are added in subsequent iterations.

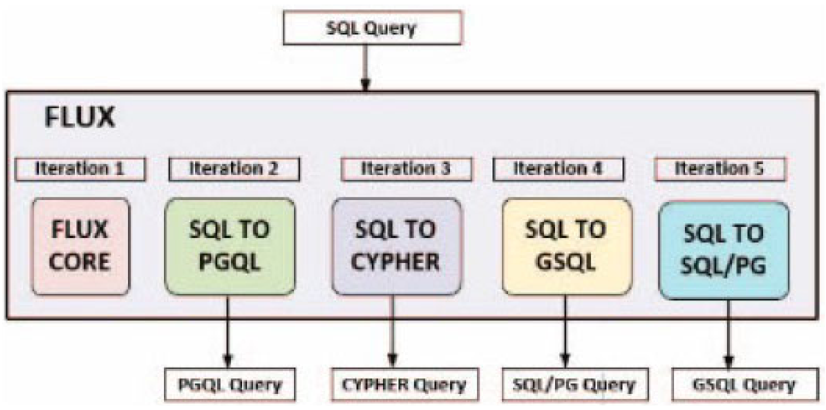


Figure D.2: Block diagram illustrating five iterations for constructing **FLUX**

Prototype construction for **FLUX** is presented in Figure 2 where in the 1^{st} iteration we develop the core features for **FLUX**. These core features are derived from existing query formalisms. In the 2^{nd} and 3^{rd} iterations we add modules that convert SQL to PGQL and SQL to Cypher respectively. We further enhance **FLUX** to support translation of SQL to SQL/PG and SQL to GSQL in the 4^{th} and 5^{th} iteration. As shown in Figure 2 **FLUX** in its full operational form takes a SQL query as input and output's an equivalent query in PGQL, Cypher, SQL/PG and/or GSQL.

D.4 Contribution and significance

Given the suitability of graph databases for Big Data Analytics and initiatives such as ISO/IEC 39075 have made the industry wide adoption of graph databases much easier. On the other hand organizations have invested heavily in relational databases and rely on them. In order to migrate from relational to graph databases a database administrator has to manually convert data model and the associated queries. A tool such as **FLUX** will facilitate the industry wide adoption of graph databases. By using **FLUX** organizations can port their existing relational database based Big Data Analytics solutions to graph databases henceforth, harness the power of graph databases. Moreover, a formal approach followed to construct **FLUX** ensures accuracy in the query translation process and opens up opportunity for future research.

This research make significant progress in answering the research questions mentioned in Section D.1.3 through the following primary contributions:

- Define the formal semantics for translating queries written in SQL to GQL queries.
- Develop a query translation tool **FLUX** that can be used to convert queries written in SQL to GQL queries.

D.4.1 Scope of the study

The scope of this study is limited to converting SQL queries to Cypher and PGQL. Support for other languages such as GSQL and SQL/PG will be added in next iterations. The language G-Core has not been fully implemented yet hence support for G-Core will be added in the future iterations of **FLUX**.

Appendix E

List of Acronyms

ACM Association of Computing Machinery

API Application Programming Interface

ATRA Alternating Top-down tree on Register Automata

BiDaML Big Data Analytics Modeling Language

BUDA Bottom-Up alternating Tree Automata with one register

CFP Context Free Paths

CNRE Conjunctive Nested Regular Expressions

CRPQ Conjunctive Regular Path Queries

CSV Comma Separated Value

CQ Conjunctive Queries

CTL Computational Tree Logic

C2RPQ Conjunctive Two-way Regular Path Queries

DD Downward Data

EDBT Extending Database Technology

ELT Extract Transform Load

ERD Entity Relationship Diagram

FSA Finite State Automata

GDM Graphical Data Model

GOOD Graph Oriented Object Database model

GQL Graph Query Language

HNQL Hyper Node Query Language

HTML Hyper Text Markup Language

ICDT International Conference on Database Theory

ICPS Industrial Cyber Physical systems

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

ISO International Organization for Standardization

LDBC Linked Data Benchmark Council

LDM Logical Data Model

LPG Labeled Property Graphs

LTL Linear Temporal Logic

NBCSP National Bowel Cancer Screening Program

NRE Nested Regular Expressions

OLAP Online Analytical Processing

OLTP Online Transaction Processing

PaMaL Pattern Matching Language

PGQL Property Graph Query Language

PODS Symposium on Principles of Database Systems

RDF Resource Description Framework

RDPQ Regular Data Path Queries

RPQ Regular Path Queries

SNQL Social Networks Query and Transformation Language

SoQL Social Networks Query Language

SQL Structured Query Language

SysML	System Modeling Language
TA	Tarski's Algebra
TKDE	Transactions on Knowledge and Data Engineering
TODS	Transactions on Database Systems
2RPQ	Two-way Regular Path Queries
UCRPQ	Union of Conjunctive Regular Path Queries
UCQ	Union of Conjunctive Queries
UC2RPQ	Union of Conjunctive Two-way Regular Path Queries
UCN2RPQ	Union of Conjunctive Nested Two-way Regular Path Queries
UML	Unified Modeling Language
UnQL	Unstructured data Query Language
VLDB	Very Large Data Base
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

Appendix F

Benchmark queries

Table F.1: Fixed length navigation patterns in Cypher and PGQL

		Exp	Cypher	PGQL

τ_1	WROTE|OWNS|HAS	1a	MATCH (x)-[:WROTE|OWNS|HAS]->(y)
			WHERE x <> y
			RETURN DISTINCT x.name, y.name
				SELECT DISTINCT x.name,y.name
				MATCH
				(x)-[:WROTE|OWNS|HAS]->(y)
				WHERE x <> y

	HAS|OWNS$^-$	1b	MATCH (x)-[HAS]->(y)
			WHERE x <> y
			RETURN x.name,y.name
			UNION
			MATCH (x)-[:OWNS]->(y)
			WHERE x <> y
			RETURN x.name,y.name
				N/A

	OWNS	1c	MATCH (x)-[:OWNS]->(y)
			WHERE x <> y
			RETURN DISTINCT x.name, y.name
				SELECT DISTINCT x.name,y.name
				MATCH (x)-[:OWNS]->(y)
				WHERE x <> y

	OWNS$^-$|FRIEND_OF$^-$	1d	MATCH (x)<-[:OWNS|FRIEND_OF]-(y)
			WHERE x <> y
			RETURN DISTINCT x.name, y.name
				SELECT DISTINCT x.name, y.name
				MATCH (x)<-[OWNS|FRIEND_OF]-(y)
				WHERE x <> y

τ_2	KNOWS$^-$.OWNS	2a	MATCH (x)<-[:KNOWS]-(p)
			MATCH (p)-[:OWNS]->(y)
			WHERE x <> y
			RETURN DISTINCT x.name, y.name
				SELECT DISTINCT x.name,y.name
				MATCH (x)<-[:KNOWS]-()
				-[:OWNS]->(y)
				WHERE x <> y

Table F.1 continued from previous page

		Exp	Cypher	PGQL

	(OWNS|REVIEW_FOR).(HAS|KNOWS)	2b	MATCH (x)-[:OWNS|REVIEW_FOR]->(p)
			MATCH (p)-[:HAS|KNOWS]->(y)
			WHERE x <> y
			RETURN DISTINCT x.name,y.name
				SELECT DISTINCT x.name,y.name
				MATCH (x)-[:OWNS|REVIEW_FOR]->
				()-[:HAS|KNOWS]->(y)
				WHERE x <> y

	(OWNS|KNOWS$^-$).(OWNS|WROTE)	2c	N/A	N/A

	KNOWS$^-$.(KNOWS|OWNS)	2d	MATCH (x)<-[:KNOWS]-(p)
			MATCH (p)-[:KNOWS|OWNS]->(y)
			WHERE x <> y
			RETURN DISTINCT x.name,y.name
				SELECT DISTINCT x.name,y.name
				MATCH (x)<-[:KNOWS]-()
				-[:KNOWS|OWNS]->()
				WHERE x <> y

	KNOWS$^-$.(KNOWS|FRIEND_OF$^-$)	2e	N/A	N/A

	KNOWS$^-$.(KNOWS$^-$|FRIEND_OF$^-$)	2f	MATCH (x)<-[:KNOWS]-(p)
			MATCH (p)<-[:KNOWS|FRIEND_OF]-(y)
			WHERE x <> y
			RETURN DISTINCT x.name,y.name
				SELECT DISTINCT x.name,y.name
				MATCH (x)<-[:KNOWS]-()
				<-[:KNOWS|FRIEND_OF]-(y)
				WHERE x <> y

τ_3	WROTE.REVIEW_FOR.HAS	3a	MATCH (x)-[:WROTE]->(p)
			MATCH (p)-[:REVIEW_FOR]->(q)
			MATCH (q)-[HAS]->(y)
			WHERE x <> y
			RETURN DISTINCT x.name,y.name
				SELECT DISTINCT x.name.y.name
				MATCH (x)-[:WROTE]->()
				-(:REVIEW_FOR)-> ()-[:HAS]->(y)
				WHERE x <> y

	(KNOWS|WROTE). (FRIEND_OF$^-$|FRIEND_OF). (REVIEW_FOR|WROTE$^-$|OWNS)	3b	N/A	N/A

Table F.1 continued from previous page

		Exp	Cypher	PGQL

	`HAS.(REVIEW_FOR⁻|OWNS⁻).`
	`(FRIEND_OF⁻|WROTE⁻|KNOWS⁻)`
		3c	`MATCH (x)-[:HAS]->(p)`
			`MATCH (p)<-[:REVIEW_FOR|OWNS]-(q)`
			`MATCH (q)<-[FRIEND_OF|WROTE|KNOWS]-(y)`
			`WHERE x <> y`
			`RETURN DISTINCT x.name,y.name`
				`SELECT DISTINCT x.name, y.name`
				`MATCH (x)-[:HAS]->()`
				`  <-[:REVIEW_FOR|OWNS]-()`
				`  <-[FRIEND_OF|WROTE|KNOWS]-(y)`
				`WHERE x <> y`

$(\tau_1\|\tau_2)$	`WROTE|(REVIEW_FOR.HAS)`	4a	`MATCH (x)-[:WROTE]->(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
			`UNION`
			`MATCH (x)-[:REVIEW_FOR]->(p)`
			`WHERE x <> y`
			`MATCH (p)-[:HAS]->(y)`
			`RETURN x.name,y.name`
				N/A

Table F.1 continued from previous page

		Exp	Cypher	PGQL

	`OWNS|(KNOWS`$^-$`.OWNS)|WROTE`	4b	`MATCH (x)-[:OWNS]->(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
			`UNION`
			`MATCH (x)<-[:KNOWS]-(p)`
			`MATCH (p)-[:OWNS]->(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
			`UNION`
			`MATCH (x)-[:WROTE]->(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
				`N/A`

Table F.1 continued from previous page

		Exp	Cypher	PGQL

	`KNOWS|FRIEND_OF⁻|`
	`(OWNS.REVIEW_FOR⁻)|`
	`(WROTE.REVIEW_FOR)`
		4c	`MATCH (x)-[:KNOWS]->(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
			`UNION`
			`MATCH (x)<-[:FRIEND_OF]-(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
			`UNION`
			`MATCH (x)-[:OWNS]->(p)`
			`MATCH (p)<-[:REVIEW_FOR]-(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
			`UNION`
			`MATCH (x)-[:WROTE]->(p)`
			`MATCH (p)-[:REVIEW_FOR]->(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
				N/A

Table F.2: Arbitrary length navigation patterns in Cypher and PGQL

		Exp	Cypher	PGQL

$(\tau_1)^*$	(WROTE|OWNS|HAS)*	5a	MATCH (x)-[:WROTE|OWNS|HAS*]->(y)
			WHERE x <> y
			RETURN DISTINCT x.name,y.name
				PATH p AS ()-[:WROTE|OWNS|HAS]->()
				SELECT DISTINCT x.name,y.name
				MATCH (x)-/:p+/->(y)
				WHERE x <> y

	(HAS|OWNS$^-$)*	5b	N/A	N/A

	(KNOWS)*	5c	MATCH (x)-[:KNOWS*]->(y)
			WHERE x <> y
			RETURN DISTINCT x.name, y.name
				SELECT DISTINCT x.name,y.name
				MATCH (x)-/:KNOWS+/->(y)
				WHERE x <> y

	(OWNS$^-$|FRIEND_OF$^-$)*	5d	MATCH (x)<-[:OWNS|FRIEND_OF*]-(y)
			WHERE x <> y
			RETURN DISTINCT x.name, y.name
				SELECT DISTINCT x.name, y.name
				MATCH (x)<-[:OWNS|FRIEND_OF]-(y)
				WHERE x <> y

$(\tau_2)^*$	(KNOWS$^-$.OWNS)*	6a	N/A	PATH p AS ()<-[:KNOWS]-()-[:OWNS]->()
				SELECT DISTINCT x.name,y.name
				MATCH (x)-/:p+/->(y)
				WHERE x <> y

	((OWNS|REVIEW_FOR). (HAS|KNOWS))*	6b	N/A	PATH p AS ()-[:OWNS|REVIEW_FOR]->() -[:HAS|KNOWS]->() SELECT DISTINCT x.name, y.name MATCH (x)-/:p+/->(y) WHERE x <> y
	((OWNS|KNOWS$^-$). (OWNS|WROTE))*	6c	N/A	N/A

Table F.2 continued from previous page

		Exp	Cypher	PGQL
	(KNOWS$^-$. (KNOWS|OWNS))*	6d	N/A	PATH p AS ()<-[KNOWS]-() -[KNOWS|OWNS]->() SELECT DISTINCT x.name,y.name MATCH (x)-/:p+/->(y) WHERE x <> y
	(KNOWS$^-$. (KNOWS|FRIEND_OF$^-$))*	6e	N/A	N/A
	(KNOWS$^-$. (KNOWS$^-$|FRIEND_OF$^-$))*	6f	N/A	PATH p AS ()<-[:KNOWS]-() <-[KNOWS|FRIEND_OF]-() SELECT DISTINCT x.name, y.name MATCH (x)-/:p+/->(y) WHERE x <> y
$(\tau_3)^*$	(WROTE.REVIEW_FOR.HAS)*	7a	N/A	PATH p AS ()-[:WROTE]->() -[:REVIEW_FOR]->()-[HAS]->() SELECT DISTINCT x.name,y.name MATCH (x)-/:p+/->(y) WHERE x <> y
	((KNOWS|WROTE). (FRIEND_OF$^-$|FRIEND_OF). (REVIEW_FOR|WROTE$^-$|OWNS))*	7b	N/A	N/A

Table F.2 continued from previous page

		Exp	Cypher	PGQL
	(HAS.(REVIEW_FOR$^-$|OWNS$^-$). (FRIEND_OF$^-$|WROTE$^-$|KNOWS$^-$))*	7c	N/A	PATH p AS ()-[:HAS]-> ()<-[:REVIEW_FOR|OWNS] -()<-[:FRIEND_OF|WROTE|KNOWS]-() SELECT DISTINCT x.name, y.name MATCH (x)-/:p+/->(y) WHERE x <> y
$(\tau_1\|\tau_2)^*$	(WROTE|(REVIEW_FOR.HAS))*	8a	N/A	N/A
	(OWNS|(KNOWS$^-$|OWNS).WROTE)*	8b	N/A	N/A
	(KNOWS|FRIEND_OF$^-$| (OWNS.REVIEW_FOR$^-$)| (WROTE.REVIEW_FOR))*	8c	N/A	N/A

$\tau_1^*.\tau_1^*$	KNOWS*.
	(FRIEND_OF|OWNS)*
		9a	MATCH (x)-[:KNOWS*]->(p)
			MATCH (p)-[:FRIEND_OF|OWNS*]->(y)
			WHERE x <> y
			RETURN DISTINCT x.name, y.name
				PATH p1 AS ()-[:KNOWS]->()
				PATH p2 AS ()-[:FRIEND_OF|OWNS]->()
				SELECT DISTINCT x.name,y.name
				MATCH (x)-/:p1+/->()-/:p2+/->(y)
				WHERE x <> y

	KNOWS*.
	FRIEND_OF^{-*}
		9b	MATCH (x)-[:KNOWS*]->(p)
			MATCH (p)<-[:FRIEND_OF*]-(y)
			WHERE x <> y
			RETURN DISTINCT x.name,y.name
				PATH p1 AS ()-[:KNOWS]->()
				PATH p2 AS ()-[:FRIEND_OF]->()
				SELECT DISTINCT x.name,y.name
				MATCH (x)-/:p1+/->()<-/:p2+/-()
				WHERE x <> y

Table F.2 continued from previous page

		Exp	Cypher	PGQL

$\tau_1^* . \tau_2^*$	`KNOWS`*.
	`(FRIEND_OF.KNOWS`$^-$`)`*
		10a	`N/A`	`PATH p AS ()-[:FRIEND_OF]->`
				`()<-[:KNOWS]-()`
				`SELECT DISTINCT x.name,y.name`
				`MATCH (x)-/:KNOWS+/->()`
				`-/:p+/->(y)`
				`WHERE x <> y`

	`(KNOWS|FRIEND_OF`$^-$`)`*. `((OWNS|REVIEW_FOR`$^-$`).HAS)`*	10b	`N/A`	`N/A`

	`(KNOWS|FRIEND_OF)`*.
	`((OWNS|REVIEW_FOR).HAS)`*
		10c	`N/A`	`PATH p1 AS ()-[:KNOWS|FRIEND_OF]->()`
				`PATH p2 AS ()-[:OWNS|REVIEW_FOR]->`
				`()-[HAS]->()`
				`SELECT DISTINCT x.name,y.name`
				`MATCH (x)-/:p1+/->()-/p2+/->(y)`
				`WHERE x <> y`

$\tau_1^* \| \tau_1^*$	`(KNOWS)`*|`(FRIEND_OF`$^-$`)`*	11	`MATCH (x)-[:KNOWS*]->(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
			`UNION`
			`MATCH (x)<-[:FRIEND_OF*]-(y)`
			`WHERE x <> y`
			`RETURN x.name,y.name`
				`N/A`

$\tau_1^* \| \tau_2^*$	`(KNOWS)`*|`(FRIEND_OF.OWNS)`*	12	`N/A`	`N/A`
$(\tau_1^*)^*$	`(KNOWS`*`)`*	13	`N/A`	`N/A`
$(\tau_2^*)^*$	`(KNOWS.OWNS`*`)`*	14	`N/A`	`N/A`

Table F.2 continued from previous page

		Exp	Cypher	PGQL
$(\tau_1^* \vert \tau_2^*)^*$	((FRIEND_OF)|*(KNOWS.OWNS)*)*	15	N/A	N/A
$(\tau_1^*.\tau_2^*)^*$	((FRIEND_OF).*((KNOWS.OWNS))*)*	16	N/A	N/A