

UNIVERSITÉ PARIS CITÉ

École doctorale 386 Sciences Mathématiques de Paris Centre
Institut de Recherche en Informatique Fondamentale
Data Intelligence Institute of Paris

Design principles of property graph languages

A theoretical and experimental study

Par ALEXANDRA ROGOVA

Thèse de doctorat d'informatique

Dirigée par AMÉLIE GHEERBRANT

Et par LEONID LIBKIN

Présentée et soutenue publiquement le 27/11/2024

Devant un jury composé de :

ANDREA CALI, PROF.	Université de Naples	Rapporteur
IOANA MANOLESCU, DR	INRIA Paris Saclay	Rapportrice
CLAIRE DAVID, MCF	Université Gustave Eiffel	Examinatrice
DIEGO FIGUEIRA, DR	Université de Bordeaux	Examinateur
CRISTINA SIRANGELO, PROF.	Université Paris Cité	Examinatrice
MICHAEL THOMAZO, CR-HDR	ENS Paris	Examinateur
AMÉLIE GHEERBRANT, MCF	Université Paris Cité	Directrice de thèse
LEONID LIBKIN, PROF.	University of Edinburgh	Directeur de thèse

Acknowledgments

I would like to sincerely thank the following people for their guidance, inspiration, friendship and support in these last three years. Writing this thesis would have been much less fun without them.

To my advisors, Amélie and Leonid, who have taught me so much about databases, science and life, despite the ever increasing chaos. This thesis is a tribute to their knowledge, patience and talent in science and supervision.

To my co-authors, especially Liat and Filip, writing with you was always a pleasure, even during the world's rapid descent into madness. I have learned a great deal from you and I am ever grateful for your kindness and goodwill.

To Julo, for having the patience to listen to me talk about cooking, sewing, knitting (and a tiny bit about databases), and for always encouraging me to try new things.

To Adrienne, who is always ready to fight the good fight. It was an honour to represent the non-permanent members of IRIF with you and it is a real joy to be your friend. May the beers we share in the future be as enjoyable as the ones we have shared in the past.

To the members of the automata team, especially Thomas, Sylvain and Marie, for the time and energy they put into creating a community within the team.

To Aliaume, Bernardo, Olivier S., Mickael and Nicolas, my wonderful office mates. The time spent chatting about anything and everything, silly and serious, was well worth it.

To Delia, Cristina and Sophie, who have helped and pushed me, all throughout my studies, to pursue research. You are an inspiration, each in your own way, and I would not be here without you.

To Mariana, Gio, Avi, Laurent, Colin, Olivier I., Florian, Cécile, Loïc and everyone with whom we have shared food and drink. From politics to cooking, by way of games, advisors and quick quick detour to lambda calculus, our chats were always interesting and I hope we will have many more.

To Tristan, Olivier T., Léa, Anne-Claire, Madeleine, Alice, Théo and Lilya, who have put up with me for so many years that I have stopped counting. Your friendship truly means a lot.

To the members of my jury, Andrea, Ioana, Michael, Claire, Diego and Cristina, and the members of my Comité de suivi, Arnaud and Angela. Thank you for taking the time to listen and read my work, and for the thorough feedback. Your corrections and suggestions have allowed me to improve this thesis and myself as a researcher.

And of course, thank you to my mother for her support and advice throughout all my life.

Résumé court

Principes de conceptions des languages de graphes de propriété
une étude théorique et expérimentale

Au cours des 50 dernières années, la quantité et la complexité des informations stockées dans les bases de données ont augmenté et les besoins des utilisateurs ont évolué. Ces changements ont conduit à la création de nouveaux modèles tels que XML, les bases de données clés-valeurs, de séries temporelles, etc. Le sujet de cette thèse est l'un de ces modèles : la base de données graphe. Dans une base de données graphe, les données sont stockées sous la forme d'un *graphe*, ou d'une collection de nœuds, représentant les entités, reliés entre eux par des arêtes, représentant les relations entre les entités. Des informations supplémentaires sur les entités et leurs relations peuvent être attachées aux nœuds et aux arêtes. Ce modèle puissant, popularisé par Neo4j en 2010, est aujourd'hui incontournable dans divers secteurs tels que la biologie, les réseaux sociaux et la banque. En mettant en avant les liens entre les données, les bases de données de graphes permettent aux utilisateurs de raisonner non seulement sur les éléments individuels, mais aussi sur la structure du graphe entier. En conséquence, l'objectif d'une requête de graphe typique est de trouver un *chemin* reliant des nœuds spécifiques.

Comme les traversées de graphes reposent intrinsèquement sur la transitivité, les langages de requête traditionnels ne sont pas adaptés au contexte des graphes, et de nouveaux langages ont donc été créés. Dans la communauté théorique, les éléments de base d'un langage de graphe sont les requêtes de chemins réguliers (RPQ), qui définissent les contraintes de chemin au moyen d'expressions régulières. Le pouvoir d'expression et la complexité des RPQ et de leurs extensions (par l'union, la conjonction, la navigation bidirectionnelle, les comparaisons de valeurs de données et les propriétés de chemin, par exemple) sont étudiés depuis les années 1990, mais leurs propriétés commencent à peine à être comprises.

Le langage graphe pratique le plus populaire aujourd'hui est Cypher de Neo4j. Dans Cypher, un chemin peut être décrit par une expression régulière, mais il comprend également de nombreuses autres fonctionnalités, notamment l'agrégation, différentes sémantiques de chemin, la projection, les sous-requêtes et les mises à jour. Ces éléments diffèrent de ceux d'autres langages, comme GSQL de Tigergraph ou PGQL d'Oracle, mais tous les systèmes de graphes partagent le même noyau : le *pattern matching*.

En 2019, un nouveau groupe de l'Organisation internationale de normalisation (ISO) a été créé pour superviser la normalisation des langages graphes pratiques. Cela a donné lieu à deux nouvelles normes: SQL/PGQ et GQL. L'idée de SQL/PGQ est d'ajouter un mécanisme de pattern matching basé sur les vues à SQL et d'interpréter les données relationnelles en tant que graphe uniquement lorsque cela est nécessaire, tandis que GQL est autonome et stocke les données en tant que graphe natif. Bien que ce travail de normalisation soit un pas dans la bonne direction, il manque un ingrédient crucial: un modèle théorique correspondant.

L'objectif de cette thèse est de définir un langage théorique pour les bases de données de graphes, semblable à l'algèbre relationnelle pour SQL, qui reflète les aspects essentiels de GQL et de SQL/PGQ tout en étant suffisamment simple pour une étude théorique. Nous commençons par présenter en détail les caractéristiques de pattern matching partagées par SQL/PGQ et GQL. Nous identifions et formalisons ensuite le noyau de ce langage. Ensuite, nous positionnons nos formalisations de SQL/PGQ et GQL par rapport à l'algèbre relationnelle, ce qui nous permet de mettre en évidence leurs styles distincts, tout en prouvant leur équivalence. Enfin, nous explorons l'impact de l'extension du pattern matching avec des fonctions de liste et montrons que cet ajout n'est pas seulement dangereux en théorie, mais qu'il échoue également en pratique.

Mots clefs: théorie des bases de données, langages de requete, informatique theorique, informatique fondamentale

Short abstract

Design principles of property graph languages
a theoretical and experimental study

In the last 50 years, the amount and complexity of information stored in databases has grown and the needs of database users have evolved. These changes have led to the creation of new models such as XML, key-value stores, time series databases and so on. The subject of this thesis is one such model: the graph database. In a graph database, data is stored as a "graph", or a collection of nodes, representing the entities, connected together by edges, representing relationships between the entities. Additional information about the entities and their relationships can be attached to the nodes and edges. This powerful model, popularized by Neo4j in 2010, is now a staple in various industries such as biology, social networks and banking. By putting the links between the data points front and center, graph databases allow users to reason not only about individual elements but also about the structure of the graph. Accordingly, the goal of a typical graph query is to find a "path" connecting specific nodes.

Because graph traversals inherently rely on transitivity, traditional query languages are not suitable in the graph context, and thus new languages have been created. In the theoretical community, the basic building blocks of a graph language are the Regular Path Queries (RPQs), which define path constraints by way of regular expressions. The expressive power and complexity of RPQs and their extensions (by union, conjunction, two-way navigation, data value comparisons and path properties for example) have been studied since the 1990s but their properties are barely beginning to be understood.

The most popular practical graph language today is Neo4j's Cypher. In Cypher, a path can be described by a regular expression but it also includes many other features among which aggregation, different path semantics, projection, subqueries and updates. These elements differ from those of other languages, like Tigergraphs' GSQL, or Oracle's PGQL, but all graph systems share the same kernel: pattern matching.

In 2019, a new International Organisation for Standardization (ISO) group was created to oversee the standardization of practical graph languages. This led to two new standards: SQL/PGQ and GQL. The idea of SQL/PGQ is to add a view-based pattern matching mechanism to SQL and interpret the relational data as a graph only when necessary, whereas GQL is standalone and stores the data as a native graph. While this standardization work is a step in the right direction, there is one crucial ingredient missing: a corresponding theoretical model.

The goal of this thesis is to define a theoretical language for graph databases, akin to relational algebra for SQL, that reflects the essential aspects of GQL and SQL/PGQ while being simple enough for theoretical study. We start by presenting in detail the pattern matching features shared by SQL/PGQ and GQL. We then identify and formalize the core of this language. Afterwards, we position our formalisations of SQL/PGQ and GQL in comparison to relational algebra, which allows us to highlight their distinct styles, while also proving their equivalence. Finally, we explore the impact of extending pattern matching with list functions and show that this addition is not only dangerous in theory, but also fails in practice.

Keywords: database theory, query languages, theoretical computer science

List of symbols and acronyms

Theoretical query languages

Symbol	Meaning	Chapter
FO	First Order Logic	2
RA	Relational Algebra	2
1NF	First Normal Form	5
CQ	Conjunctive Query	2
UCQ	Union of Conjunctive Queries	2
BCCQ	Boolean Combinations of Conjunctive Queries	5
DFA	Deterministic Finite Automaton	2
RA	A k-Register Automaton	2
Regex	Regular Expression	2
REM	Regular Expression with Memory	1
RPQ	Regular Path Query	1
CRPQ	Conjunctive Regular Path Query	1
UCRPQ	Union of Conjunctive Regular Path Queries	1
2RPQ	2-way Regular Path Query	1
C2RPQ	2-way Conjunctive Regular Path Query	2
UC2RPQ	Union of 2-way Conjunctive Regular Path Queries	2
NRE	Nested Regular Expression	1
ECRPQ	Extended Conjunctive Regular Path Query	1
RDPQ	Regular Data Path Query	2
RQM	Regular Query with Memory	2
GPC	Graph Pattern Calculus	4
GPC+	GPC extended with projection and union	4
LCRA	Linear Composition Relational Algebra	5
sLCRA	Simple Linear Composition Relational Algebra	5
+NF	+ Normal Form	5

Complexity

Symbol	Meaning	Chapter
DLOGSPACE	Deterministic Logarithmic Space complexity class	2
NLOGSPACE	Non-Deterministic Logarithmic Space complexity class	2
PTime	Deterministic Polynomial Time complexity class	2
NP	Non-Deterministic Polynomial Time complexity class	2
PSPACE	Deterministic Polynomial Space complexity class	2
EXPSPACE	Deterministic Exponential Space complexity class	2
2EXPSPACE	Deterministic Twice-Exponential Space complexity class	2
AC^0	Complexity class of problems solvable using a boolean circuit of constant depth and polynomial width	2

Practical query languages

Symbol	Meaning	Chapter
SQL	The standard query language for relational databases	1
ISO	The International Organisation for Standardization	1
ISO/IEC JTC1 SC32 WG3	The ISO working group focused on SQL and GQL	2
XML	A standard markup language	1
PGQL	A graph query language by Oracle	1
GSQL	A graph query language by Tigergraph	1
SAP HANA	A database system by SAP	2
Cypher	A graph query language by Neo4j	1
GQL	The new standard query language for graph databases	1
SQL/PGQ	The new SQL extension for graph databases	1
CODASYL	The Conference/Committee on Data Systems Languages	2
DBPedia	A database referencing Wikipedia data	2
YAGO	Yet Another Great Ontology, an open source knowledge base	2
RDF	Resource Description Framework, the standard ontology language	2
SPARQL	The query language for RDF	5
IRI	International Resource Identifier, the identifier set for RDF	2
W3C	The World Wide Web Consortium	2
*RDF**	An extension to RDF with a notion of nested triples	2
DBMS	Database Management System	4

Contents

1 Introduction

When databases are mentioned, most people think about the relational model, or SQL, and with good reason. Introduced by Codd in 1970 [1], this model combines a strong mathematical basis, the tradition of bookkeeping and the philosophy that the user should be as far removed from the implementation as possible. This combination creates a tool that is not only easy to use[1], but also close to a theoretical model that can be formally studied. It is still today the dominant database model at all levels of data management.

[1]: Codd (1970), "A Relational Model of Data for Large Shared Data Banks"

1: at least relatively to what was previously available

Howver the last 50 years have seen an enormous growth in the size of data and a shift in the needs of users. These changes have led to the creation of new models, some specific to a particular domain, such as time series databases for time-linked data, others capable of modelling any kind of data, such as XML or key-value stores. The subject of this thesis is one such specific model: the *graph database*.

As the name implies, the focus of graph databases is graph-like data, i.e. data that represents heavily-linked information where the topology is an important aspect. The family of graph models is popular in both practical applications, from biology, to social networks, to banking, and theoretical fields such as mathematics and computer science. It is no surprise, then, that graphs were already considered a suitable model for storing data in the 1970s, as attested by the hierarchical [2] and network [3] models. In the former, data was sorted according to a strict tree schema, detailing the relationships between its components. For example, a schema could specify that an entry "book" requires a parent "author". The latter relaxes the tree constraint to general graphs in which entries can have multiple parents, or even be the parent of their parent, allowing for example a book to have multiple authors in the previous example schema.

[2]: Tsichritzis et al. (1976), "Hierarchical Data-Base Management: A Survey"

[3]: Taylor et al. (1976), "CODASYL Data-Base Management Systems"

However, both of these models suffer from the pre-relational flaw: their query languages were intrinsically linked with the in-memory structure and required complicated procedures in order to extract simple information. By the end of the 1990s, hierarchical and network databases had almost completely disappeared as the industry's interest shifted to semi-structured data and the web.

From the 1980s to the 2000s, a number of graph database models have been proposed (see [4] for an in-depth survey) but it is only in 2010 that a specific system, Neo4j, gained enough popularity to restart the trend in earnest. As of July 2024, Neo4j is still the market leader, ranking 1st among graph-specific systems, and 21st overall [5]. In a Neo4j graph, called a *property graph*, nodes represent individuals or entities, and the edges show the relationships between them. Nodes can be thought of as the subject and object of a sentence, while the edge is a verb. To store extra information about individuals or relationships, nodes and edges can be decorated with labels, denoting that they belong to a given group, like "Book", and properties, specifying the value of some attribute, such as a "Title". Unlike in a hierarchical or network database, a Neo4j property graph does not have an associated schema, meaning data can be added or updated regardless of the previous state, making the system arguably easier to use and maintain.

[4]: Angles et al. (2008), "Survey of graph database models"

[5]: https://db-engines.com (),

One major difference between relational and graph databases is that the focal point of the former is on the individual elements, whereas for the latter, it is the links and structure between the elements. For example, the purpose of a typical relational query could be to get the titles of some author's books, while a graph query might ask for groups of authors who often write together. While it is possible to extend a relational language, such as SQL, in order to incorporate navigational tools, this tends to result in heavy, complicated queries. Instead, graph database systems have developed their own query languages, often focused on reachability and paths.

The first modern graph query language is G [6]. Introduced by I. Cruz, A. Mendelzon and P. Wood in 1987, a query in G is a graph whose edges are labelled by regular expressions. The answer to such a query is the set of subgraphs of the database that *match* the graph described by the query, i.e. where each regular expression has a corresponding path in which the word formed by the labels of its edges belongs to the language defined by the regular expression. This *pattern matching* mechanism is now at the heart of all contemporary graph query languages. G was then extended to G+ [7] and GraphLog [8] to incorporate conjunction and a limited form of negation[2].

[6]: Cruz et al. (1987), "A Graphical Query Language Supporting Recursion"

[7]: Cruz et al. (1988), "G+: Recursive Queries Without Recursion"

[8]: Consens et al. (1990), "GraphLog: a Visual Formalism for Real Life Recursion"

2: As well as an operation to reformat the output graphs, but this idea did not survive.

The notion of searching for paths using regular expressions was quickly adopted by the research community and soon the 'Regular Path Query' (RPQ) was considered the basic building block of any reasonable graph query language. A new branch of study emerged, with the goal of understanding RPQs and its various extensions. The data complexity of query evaluation was determined to be tractable even with conjunctions (CRPQs), unions (UCRPQs), backwards edges (2RPQs), nesting (NREs), data comparisons (REMs) and path properties (ECRPQs) [8–11]. The combined complexity of the same problem turns out to be polynomial only for basic RPQs [12], 2RPQs and NREs [9] and NP or harder for all the other classes [9, 11]. A tractable class of UC2RPQs was identified for the containment problem [13] and NREs were proven to be incomparable with C2RPQs [14], among countless other results.

[8]: Consens et al. (1990), "GraphLog: a Visual Formalism for Real Life Recursion"
[9]: Barceló (2013), "Querying graph databases"
[10]: Libkin et al. (2016), "Querying Graphs with Data"
[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"
[12]: Wood (2012), "Query languages for graph databases"
[13]: Figueira (2020), "Containment of UC2RPQ: The Hard and Easy Cases"
[14]: Barceló et al. (2012), "Relative Expressiveness of Nested Regular Expressions"

Although pattern matching also appears prominently in industry languages such as Neo4j's Cypher [15], Oracle's PGQL [16] and Tigergraph's GSQL [17], every graph database system has developed its own style and dialect, usually incompatible with the others. For instance, the syntax of PGQL is much closer to SQL, with `SELECT`, `FROM` and `WHERE` clauses, while Cypher's is more imperative with `MATCH` and `RETURN`. A path in GSQL is by definition the shortest, whereas in Cypher it can be of any length as long as edges are not repeated.

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"
[16]: Language (2021), *PGQL 1.4 Specification*
[17]: Deutsch et al. (2020), "Aggregation Support for Modern Graph Analytics in TigerGraph"

Despite these kinds of variations, Cypher, PGQL and GSQL have much in common: they use the same property graph model, a query is defined by a regular expression, extra conditions on elements and properties can be added with a `WHERE` clause, the output is a table containing graph elements often grouped into paths, and so on. In 2019, a new International Organisation for Standardization (ISO) group was created to oversee the standardization of graph query languages. The work of this group resulted in two new standards

- SQL/PGQ, a new part of the SQL Standard and
- GQL, an independent language focused solely on graphs.

In SQL/PGQ, graphs are stored inside a (possibly larger) relational structure and viewed as graphs only during the pattern matching phase.

The advantages of this approach are that graph data can be easily integrated with relational data, and that we can continue to benefit from all the techniques that we have developed for relational databases. The main drawback is that a relational structure is ill-suited to graphs and forces the data to comply with a strict schema. GQL does not impose a restriction on the storage of graphs, which is left to the discretion of the implementation. This has the advantage of simplifying the implementation of algorithms specific to graphs and optimisations that are not necessarily applicable in the relational context. Moreover, a "standard compliant" system should be easier to develop for GQL than for SQL/PGQ, as the specification is "only" 600 pages long, compared to SQL's 4400. Apart from these differences (as well as some other details concerning updates and the like), GQL and SQL/PGQ are identical, in the sense that they share the same language for pattern matching. The idea behind the design of this common core, was to combine the existing practical languages, keeping their shared components and leaving to the user the choice to handle disagreements. For example, the kind of path to be returned, be it shortest, no repeated nodes or no repeated edges, can be selected for each individual query.

While this standardization work is a step in the right direction, there is one crucial ingredient missing: a corresponding theoretical model. When SQL was first being designed, the committee relied on relational algebra and related theoretical work to create a powerful yet usable language. On the other hand, GQL and SQL/PGQ are assembled together from parts of already existing languages without a formal understanding of how these parts behave, and especially how they interact. This is the goal of this thesis: to define a theoretical language for graphs, akin to relational algebra for SQL, that reflects the essential aspects of GQL and SQL/PGQ while being simple enough for theoretical study.

Related Work

As mentioned previously, modern graph languages have been studied since at least 1987, and RPQs are by now well understood. However, none of these languages capture the core of GQL and SQL/PGQ. For instance, data values have rarely been considered, and never in conjunction with path restrictors (shortest and the like). The only proposal comparable to what we will introduce in the next chapters is G-CORE [18], a language developed by a group including theoreticians and implementers from the most influential graph database systems. As G-CORE was published in 2018, when the reflection on GQL and SQL/PGQ was in its infancy, the published standards have deviated from it in non-trivial ways; for example paths are no longer stored explicitly and the central `CONSTRUCT` operation was not included. Furthermore, while G-CORE is certainly more formal than the standards, it is still too far removed from the simplicity of relational algebra and thus not quite suitable for in-depth theoretical study.

[18]: Angles et al. (2018), "G-CORE: A Core for Future Graph Query Languages"

Outline and main contributions

Chapter 2 is dedicated to the background and preliminaries required to understand the rest of this thesis. In it we cover the definitions of relational algebra and Datalog, two ubiquitous relational languages, give an in-depth analysis of the differences between property graphs and the competing model RDF, introduce RPQs and their most common

extensions and discuss the different path restrictors included in GQL. Every language definition is accompanied by selected results whenever it makes sense, to give an intuition of what is (or isn't) feasible in our context.

Im Chapter 3 we introduce the pattern matching language of SQL/PGQ and GQL by way of examples, and present a first formalization that aims to remain as close as possible to the standards.

In Chapter 4, we define the Graph Pattern Calculus, in the spirit of the relational calculus, meant to reflect key aspects of graph pattern matching. Together with its syntax and semantics, we present a typing system that ensures that its expressions are well defined, and show two results, the first on the complexity of enumerating answers, the second on its expressive power compared to the common theoretical languages.

In Chapter 5 we bridge the gap between the graph and relational worlds and formalizes the 'linear' nature of GQL by proposing an alternative, but equivalent, definition of relational algebra, allowing us to formally distinguish between GQL and SQL/PGQ. We also prove that their common pattern matching language is not powerful enough to express a number of very simple and natural queries, and even the full languages are not as expressive as Datalog on graphs.

In Chapter 6 we consider lists and list operations, a popular extension to core pattern matching, and show that not only do they add unwanted complexity in theory but they also inevitably break the systems in practice.

Finally, we conclude in Chapter 7 and list some open questions and possible extensions for future work.

Navigating this document As explained above, this thesis is separated into Chapters, themselves separated into sections and subsections. All of these parts are listed under Contents (page 8). Clicking on either a title (or its corresponding page) will take you to its beginning.

Whenever an outside work is cited, its authors, title and publication date appear in the large margin to the side of the text[3]. Clicking on the id of a particular citation will take you to the end-of-chapter bibliography page, where more detailed information is available. A global list containing all citations of the thesis is available from page 157. Citations which are links to websites are simply underlined and do not appear in the margin.

3: This is also where the footnotes are placed.

This document uses the knowledge package. With this package, all mathematical notions (in grey) become links to their definition (in *red and italicized*). Please click on them if the acronym soup becomes too thick! If you prefer reading on paper, an index of these same notions is available at the very end (from page 154).

While the use of this package was initially motivated by the need for an index, it turned out to be of great help for writing as well. Having a new, overarching way of looking at all the topics mentioned within the text can be a great guide to determine the overall structure, or to see what's missing or out of place. It is also a good tool to catch mistakes such as forgetting to define a particular notion or having multiple names for the same concept. I highly recommend giving it a go, even for smaller documents.

Background and Preliminaries 2

2.1 Storing data

In the beginning of the 20th century, the world started shifting from hand-written (and human-read) to electronic data. Over the course of the next few decades, many systems were developed to store and retrieve this data efficiently. One thing they all had in common was that the user needed to know the exact physical location of a piece of data (its address on a disk for example) to be able to access it. Convinced that there was a better way, Edgar F. Codd introduced the *relational model* in his seminal paper *A Relational Model of Data for Large Shared Data Banks* [1]. As the name suggests, this model is based on the mathematical concept of *relations*, subsets of the Cartesian product of a collection of sets. Codd argues that this model has several advantages over those proposed before. First, the relational model separates the representation of the data the user has access to from the way the data is organized in memory. This is not only easier to handle for the database users, but also allows to simplify the query languages, which no longer need to talk about the physical representation of data. Moreover, the strong logical basis streamlines the study of the model independently of the implementation and clarifies common definitions such as redundancy and consistency. The rest of this section is dedicated to the basic definitions of the relational model and the languages used to query it. As this will not be the main focus of this thesis, only a handful of results are mentioned.

[1]: Codd (1970), "A Relational Model of Data for Large Shared Data Banks"

2.1.1 The relational model

Unless stated otherwise, all definitions in this section are taken from [19].

[19]: Arenas et al. (2022), *Database Theory*

We first fix disjoint infinite countable sets Rel of relation names (table names), Attr of attribute names (column names), Const of constants (actual values) and Vars of variable names. We usually denote elements of Rel by R, S, T, of Const by a, b, c and of Vars by x, y, z with occasional subscripts.

A relational database is composed of *tables*, which can be described by their name and their attributes. Formally, we say that a relational database has a schema which associates each relation name in that database to its set of attribute names. By abuse of notation, we sometimes refer to relations by the name that represents them.

Definition 2.1.1 — Schema

A *schema* S is a partial function from a finite subset of relation names in Rel to finite sets of attribute names from Attr.

$$S : \text{Rel} \to \mathscr{P}_{\text{fin}}(\text{Attr})$$

where $\mathscr{P}_{\text{fin}}(\text{Attr})$ is the finite powerset, i.e. the set of finite subsets, of Attr.
For a relation name $R \in S$ such that $S(R) = A_1, A_2, \dots A_n$, we call

$A_1, A_2, \dots A_n$ the *sort* of R, denoted by $\texttt{sort}(R)$, and n the *arity* of R, denoted by $\texttt{arity}(R)$.
The set of relation names that appear in a given schema S is called its *domain* and denoted by $Dom(S)$.

One way of representing relationships between some individuals is to store the information about specific persons in one table, and the information about the relationship between them in as many tables as there are relationships.

Example 2.1.2

The following schema S can be used to represent a group of individuals who are either friends or members of the same family.

$$\begin{aligned} S(\texttt{Person}) &= \{\texttt{id}, \texttt{Name}, \texttt{Age}\} \\ S(\texttt{Friend}) &= \{\texttt{Person 1}, \texttt{Person 2}, \texttt{Since}\} \\ S(\texttt{Relative}) &= \{\texttt{Person 1}, \texttt{Person 2}\} \end{aligned}$$

The *entries* of tables are called tuples. For a tuple to be part of a table it must be of the same sort, that is have the same attributes.

Definition 2.1.3 — Tuple

Given a finite subset of attribute names $\mathscr{A} = \{A_1, A_2, \dots A_k\} \in \text{Attr}^k$, a $k-$*tuple* t is a partial function from attribute names in $\mathscr{A}$ to constants from Const.

$$t : \mathscr{A} \to \text{Const}$$

We call $\mathscr{A}$ the *sort* of t, denoted by $\texttt{sort}(t)$ and k the *arity* of t, denoted by $\texttt{arity}(t)$. When needed, we denote tuples explicitly by $(A_1 : a_1, A_2 : a_2, \dots)$ where each A_i is an attribute name and each a_i is a constant.

Example 2.1.4

The following functions t_1 and t_2 are tuples over the set of attributes $\{\texttt{id}, \texttt{Name}, \texttt{Age}\}$:

$t_1(\texttt{id}) = 111$ $\quad t_1(\texttt{Name}) = \texttt{Alice}$ $\quad t_1(\texttt{Age}) = 35$
$t_2(\texttt{id}) = 444$ $\quad t_2(\texttt{Name}) = \texttt{Dana}$ $\quad t_2(\texttt{Age}) = 30$

We fix Tup to be the set of all tuples of any sort and any arity, and RelInst to be set of sets of tuples of the same sort, i.e *relation instances*. The sort of a relation instance is the sort of the tuples it contains.

A database is a function that associates the relation names from its schema to the tuples these relations contain.

Person

id	Name	Age
111	Alice	35
222	Bob	35
333	Charlie	42
444	Dana	30

Friend

Person 1	Person 2	Since
111	222	02/02/2024
222	333	04/04/2024
222	444	03/03/2024
444	111	01/01/2024

Relative

Person 1	Person 2
222	333
222	444
333	444
444	111

Figure 2.1: A database of friends and relatives

Definition 2.1.5 — Database

Given a schema S, a database $\mathscr{D}$ over S is a function from the relation names of S to relation instances over the sorts of S.

$$\mathscr{D} : Dom(S) \to \text{RelInst}$$

such that for any $R \in Dom(S)$, $\texttt{sort}(R) = \texttt{sort}(\mathscr{D}(R))$.

Example 2.1.6

Putting together the schema from Example 2.1.2, the tuples from Example 2.1.4 and some other tuples, we get the database shown in figure 2.1. Going forward, we will interpret the attributes *Person 1* and *Person 2* of the relations **Friend** and **Relative** as *foreign keys* of, or references to, the attribute *id* of the relation **Person**. [a] For example, we can say that the person with id `111` (whose name we know is `Alice`) is friends with the person with id `222` (whose name is `Bob`) since Feb. 2nd 2024.

[a] The formal definition of foreign keys is outside of the scope of this thesis.

The definitions above assume the *named perspective*, in which attributes have names, as it is closer to actual database management systems. The alternative, *unnamed perspective*, talks instead about the order of the elements and is more suited to mathematical study. It is well-known that these two models are equivalent [20].

[20]: Abiteboul et al. (1995), *Foundations of Databases*

2.1.2 Querying Relational Data

In order to extract information from a database, we need to formulate our questions in some language. We call these questions queries.

Definition 2.1.7 — Queries and query languages

A query q of arity $k \geq 0$ over a schema S is a function of the form

$$q : \mathscr{D} \to \mathscr{P}_{\text{fin}}(\text{Const}^k)$$

where $\mathscr{D}$ is a database over S and $\mathscr{P}_{\text{fin}}(\text{Const}^k)$ is the finite powerset, i.e. the set of finite subsets, of Const^k. A query language is a set of queries.

The *semantics* of a query executed on a database is its output, or what it returns. The formal definition can change slightly depending on the considered language, but in general it is the set of elements of the database that render the query true.

The usual mathematical language we use to talk about relations is a slight variation of *First Order Logic* (FO), called "Relational Calculus" in the context of database theory. As a full introduction of FO is outside the scope of this thesis, we refer the reader to any introductory logic or model theory textbook such as [21] or [22].

[21]: Libkin (2004), *Elements of Finite Model Theory*

[22]: Hodges (1997), *A Shorter Model Theory*

Example 2.1.8 — FO as a query language

The following FO formula is a query over the schema of Example 2.1.2.

$$\varphi = \exists s,\ \text{Friend}(x, y, s) \wedge \text{Relative}(x, y)$$

As the variables x and y are free, the semantics of φ is the set of pairs (p, s) such that p and s are both Friends and Relatives. On the data presented in figure 2.1, this would be the set $\{(222, 333), (222, 444), (444, 111)\}$.

First order logic is inherently a "declarative" language, it describes the results that we want without explaining how to get them. While this is usually a desirable feature for the end user, it is not enough for the machine to operate on. The naïve solution of translating directly from FO to some low-level machine code is often not suitable: whenever multiple strategies exist, the information contained within the FO query is not enough to chose one over the other. What we do instead is translate from FO to an intermediate level query language, called Relational Algebra, in which each query corresponds to a different low-level strategy. Whenever an FO query has multiple translations, these Relational Algebra queries can be compared in order to choose the best one. Furthermore, Relational Algebra happens to be a very useful tool for theoretical reasoning and we will see in chapter 5 that, no matter how far we stray, we always end up coming back to it.

Definition 2.1.9 — Relational Algebra Syntax

A *condition* θ over a set of attributes $U \subseteq \text{Attr}$ is a Boolean combination of statements of the form $A = B$, $A = a$, $A \neq B$ and $A \neq a$ where $a \in \text{Const}$ and $A, B \in U$.
Given a schema $\mathscr{S}$, Relational Algebra (RA) expressions, and their *sort*, are defined inductively as follows:
Relations: Any $R \in \mathscr{S}$ is an RA expression of sort $\mathscr{S}(R)$
Constants: If a is a constant from Const and A is an attribute name form Attr, then $\{(A : a)\}$ is an RA expression of sort $\{A\}$
Selection: If e is an RA expression of sort S and θ is a condition over S, then $\sigma_\theta(e)$ is an RA expression of sort S
Projection: If e is an RA expression of sort S and $\bar{A}$ is a subset of S, then $\pi_{\bar{A}}(e)$ is an RA expression of sort $\bar{A}$
Join: If e_1 and e_2 are two RA expressions of sort S_1 and S_2, then $e_1 \bowtie e_2$ is an RA expression of sort $S_1 \cup S_2$
Rename: If e is an RA expression of sort S and $\bar{A}$ and $\bar{B}$ are sets of attributes such that $\bar{A} \in S$ and $\bar{B} \in \text{Attr} - S$, then $\rho_{\bar{A} \to \bar{B}}(e)$ is an RA

$$
\begin{aligned}
[\![R]\!]_{\mathscr{D}} &= \mathscr{D}(R) \\
[\![(A:a)]\!]_{\mathscr{D}} &= \{(A:a)\} \\
[\![\sigma_\theta(e)]\!]_{\mathscr{D}} &= \{t \mid t \in [\![e]\!]_{\mathscr{D}} \text{ and } t \vDash \theta\} \\
[\![\pi_{\bar{A}}(e)]\!]_{\mathscr{D}} &= \{t \restriction_{\bar{A}} \mid t \in [\![e]\!]_{\mathscr{D}}\} \\
[\![e \bowtie e']\!]_{\mathscr{D}} = &= \{t \mid \forall t_1 \in [\![e]\!]_{\mathscr{D}}, t_2 \in [\![e']\!]_{\mathscr{D}}, R \in \mathtt{sort}(e) \cup \mathtt{sort}(e') \text{ s.t. } t_1(A) = t_2(A) \forall A \in \mathtt{sort}(e) \cap \mathtt{sort}(e'), \\
&\qquad t(R) = t_1(R) \text{ if } R \in \mathtt{sort}(e) \text{ and } t(R) = t_2(R) \text{ if } R \in \mathtt{sort}(e') \setminus \mathtt{sort}(e)\} \\
[\![\rho_{\bar{A}\to\bar{B}}]\!]_{\mathscr{D}} &= \{t \mid t(B) = t'(A) \text{ and } t(C) = t'(C) \text{ for all } C \in \mathtt{sort}(e) - \{A\} \text{ for all } t' \in [\![e]\!]_{\mathscr{D}}\} \\
[\![e \cup e']\!]_{\mathscr{D}} &= [\![e]\!]_{\mathscr{D}} \cup [\![e']\!]_{\mathscr{D}} \\
[\![e \setminus e']\!]_{\mathscr{D}} &= [\![e]\!]_{\mathscr{D}} \setminus [\![e']\!]_{\mathscr{D}}
\end{aligned}
$$

Figure 2.2: The semantics of RA

expression of sort $(S \setminus A) \cup B$
Union: If e_1 and e_2 are two RA expressions of the same sort S, then $e_1 \cup e_2$ is an RA expression of sort S
Difference: IF e_1 and e_2 are two RA expressions of the same sort S, then $e_1 \setminus e_2$ is an RA expression of sort S

The semantics of a Relational Algebra query over a given database $\mathscr{D}$ is defined as a set of tuples from $\mathscr{D} \cup \text{Const}$. This set is generated by applying the relational operators of the query one after the other to a *working table* which contains the current tuples needed for the computation. Initially, the working table is empty. The Relations operator can be used to populate it with tuples from the database, while the Constants operator can be used for constants. The union, difference and selection operators can manipulate the tuples, either adding or removing some, while projection and renaming change their sort, and join can do both at the same time.

A tuple t satisfies a condition θ if it satisfies the full boolean expression. Formally, we say that $t \vDash \theta$ if:

θ is of the form $A = B$ and $t(A) = t(B)$
θ is of the form $A \neq B$ and $t(A) \neq t(B)$
θ is of the form $A = a$ (resp. $A \neq a$) and $t(A) = a$ (resp. $t(A) \neq a$)
θ is of the form $\theta_1 \vee \theta_2$ and $t \vDash \theta_1 \vee t \vDash \theta_2$
θ is of the form $\theta_1 \wedge \theta_2$ and $t \vDash \theta_1 \wedge t \vDash \theta_2$
θ is of the form $\neg\theta'$ and $t \nvDash \theta'$

The definition of the *semantics* of an RA expression e over a schema $\mathscr{S}$ evaluated on a database $\mathscr{D}$, denoted by $[\![e]\!]_{\mathscr{D}}$, is shown in Figure 2.2.

Example 2.1.10

The following RA expression is a query over the schema of Example 2.1.2.

$$\pi_{\text{Name, Name 1}}(\sigma_{\text{Age=Age 1}}(\rho_{\text{ID, Name, Age}\to\text{ID 1, Name 1, Age 1}}(\text{Person}) \bowtie \text{Person}))$$

This query describes the following algorithm: take 2 copies of the table Person and rename the attributes of one to differentiate it from the other. In the resulting table, select the tuples in which the attribute "Age" is equal to the attribute "Age 1" (i.e. "Age" from the first and second copy respectively). Output the attributes "Name"

and "Name 1" for each such tuple. In other words, this query computes the pairs of people who have the same age.
On the data presented in figure 2.1, this would be the set $\{(Alice, Alice), (Bob, Bob), (Charlie, Charlie), (Dana, Dana), (Alice, Bob), (Bob, Alice)\}$

We have assumed here the named perspective to define RA. As in the case of tuples, there exists an equivalent unnamed variant of RA [20].

[20]: Abiteboul et al. (1995), *Foundations of Databases*

As we will explain in section 2.2, if a query relies on transitivity (or more generally recursion), we cannot write it in FO nor RA. To write these kinds of queries we use a language called Datalog.

Definition 2.1.11 — Datalog Syntax

A *Datalog rule* is an expression of the form

$$R(\bar{u}) \leftarrow R_1(\bar{u}_1), \dots, R_n(\bar{u}_n) \qquad n \geq 1$$

where $R, R_1, \dots, R_n$ are relation names and $\bar{u}, \bar{u}_1, \dots, \bar{u}_n$ are sequences of variables and constants over $\mathsf{Vars} \cup \mathsf{Const}$ such that each $v \in \bar{u}$ appears in at least one of $\bar{u}_1, \dots, \bar{u}_n$.
A *Datalog program* is a finite set of Datalog rules.

Given a Datalog rule $e = R(\bar{u}) \leftarrow R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$, we call $R(\bar{u})$ its *head* and $R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$ its *body*.

For a Datalog program Π, the set of constants occurring in its rules, also called its *active domain*, is denoted by $\mathrm{adom}(\Pi)$.

We say that a relation is *extensional* if it occurs only in the body of the rules, and *intensional* if it occurs in the head of at least one rule. It is sometimes useful to assume the existence of a special intensional predicate called `Ans` or `Out`, which specifies the output of a program. The *extensional schema* and *intensional schema* of a Datalog program Π, that is the sets of its extensional and intensional relations respectively, are denoted by $\mathrm{edb}(\Pi)$ and $\mathrm{idb}(\Pi)$. The *schema* of Π is the union of $\mathrm{edb}(\Pi)$ and $\mathrm{idb}(\Pi)$.

The *semantics* of Datalog can be defined in several different ways. We present here the "model theoretic" approach, in which the Datalog program is interpreted as a set of first-order sentences, to accentuate the link between Datalog and FO.

Given a Datalog rule $\rho = R(\bar{u}) \leftarrow R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$, we denote by φ_ρ the corresponding first-order sentence $\forall x_1, \dots, x_m\ (R_1(\bar{u_1}) \wedge \dots \wedge R_n(\bar{u}_n) \rightarrow R(\bar{u}))$ where $x_1, \dots, x_m$ are the variables in ρ. The set of sentences corresponding to a Datalog program Π is denoted by Σ_Π.

We say that a database $\mathscr{D}$ is a *model* of Datalog program Π if all sentences of the translation of Π are satisfied in $\mathscr{D}$, i.e. $\mathscr{D} \models \pi$ for all $\pi \in \Sigma_\Pi$.

For a Datalog program Π and a database D, the semantics of Π on D, denoted by $[\![\Pi]\!]_D$, is the unique minimum model[1] over the schema of Π that satisfies all sentences in Σ_Π and contains D. Notice that we require the model to be minimal since, if such a model exists, there are necessarily infinitely many others that also satisfy these constraints.

1: The unique minimal model can be obtained by taking the intersection of all finite models.

Example 2.1.12

The following set of rules is a Datalog program over the schema of Example 2.1.2.

1 Family(x,y) ← Relative(x,y)
2 Family(x,y) ← Relative(x,z), Relative(z,y)
3 Family(x,y) ← Family(y,x)

This query computes the transitive (rule 2) and symmetric (rule 3) closure of the Relative table, i.e. the family tree. To compute the family tree of a specific person we can simply fix the first argument to the appropriate id.

FamilyAlice(y) ← Family('111',y)

The above rule uses Alice's id. On the data presented in figure 2.1, the set corresponding to Alice's family tree is $\{Bob, Charlie, Dana, Alice\}$.

2.2 Comparing Query Languages

Relational algebra and first order logic are two examples of relational query languages. Other languages can be defined as extensions (e.g. with the addition of some aggregation operator like sum and average) or restrictions (e.g. by removing negation) of these, or even from scratch, like Datalog. To know how these languages relate to one another, we can study either their expressive power, to answer questions like "Is there a query that I can write in language 1 and that I cannot write in language 2?", or their complexity, to answer questions like "Is it harder to find answers for queries written in language 1 or language 2?".

The strategy to prove a positive expressive power result, i.e. "All queries that can be written in language $\mathscr{L}_1$ can also be written in language $\mathscr{L}_2$", is usually to provide an explicit translation algorithm, from $\mathscr{L}_1$ to $\mathscr{L}_2$. Proving a negative expressive power result, i.e. "There exists a query in language $\mathscr{L}_1$ that cannot be written in language $\mathscr{L}_2$", can be challenging as it typically relies on tools from finite model theory, automata theory, or even arithmetic.

For complexity results, languages are compared with respect to a specific problem, hence the first step is to analyze the complexity of the languages for that problem independently. The result of this analysis is a complexity class to which, the language *belongs*. In the rest of this thesis, we will refer to the common complexity classes using the standard notation like NLOGSPACE, NP, PSPACE and so on. As usual, we say that a problem is *complete* for a class if the problem is both hard and a member of this class. For more information on these classes a good introduction is provided in [23] or [24]. In addition to these, we will also refer to two less standard classes. The first is AC^0, the class of problems which can be solved using only a boolean circuit of constant depth and polynomial width. The second is the class of problems for which the answers can be enumerated in *polynomial delay*, i.e. when the time between an answer and the next is bounded by a function polynomial in the input size.

Once we know the complexity classes of all the languages, we can compare them and conclude, if say the first class is *larger* than the second, that the first language is *harder* than the second for a specific problem.

[23]: Sipser (1996), "Introduction to the Theory of Computation"

[24]: Hopcroft et al. (2007), *Introduction to automata theory, languages, and computation, 3rd Edition*

In the rest of this section we give an example of positive and negative expressive power results and present the two main problems we study in this thesis.

Expressive power

What happens if we add a new rule to the grammar of a language? Naturally, we can now write more queries, but are they different from the ones we could write before, or can each one of them be rewritten in the smaller language? The same question can be asked for languages defined independently, when one is not the extension or restriction of the other. The answer to this question allows us to classify languages with respect to one another. When all queries of some language L_1 can also be written in a language L_2, we say that L_1 is subsumed by L_2. When this is true for both directions, L_1 is subsumed by L_2 and L_2 is subsumed by L_1, the languages are said to be *equivalent*. In the following formal definition, we assume that the data model for both languages is the same, although it need not always be the case.

Definition 2.2.1 — Language subsumption and equivalence

Given two queries Q_1 and Q_2, we say that Q_1 and Q_2 are *equivalent* if $[\![Q_1]\!]_{\mathscr{D}} = [\![Q_2]\!]_{\mathscr{D}}$ on any database $\mathscr{D}$.
A (query) language L_1 is *subsumed* by a (query) language L_2 if for all queries $Q \in L_1$ there exists a query $Q' \in L_2$ such that Q_1 and Q_2 are equivalent. If L_1 is subsumed by L_2 and L_2 is subsumed by L_1, L_1 and L_2 are said to be equivalent.
A (query) language L_1 is *less expressive* than a (query) language L_2 if there exists a query $Q \in L_1$ such that, for all queries $Q' \in L_2$, Q_1 and Q_2 are not equivalent.

A famous database theory example of language equivalence is relational algebra and Relational Calculus, known as Codd's theorem.

Theorem 2.2.2 — Codd's Theorem

Relational Algebra and Relational Calculus are equivalent [19, 20].

[19]: Arenas et al. (2022), *Database Theory*
[20]: Abiteboul et al. (1995), *Foundations of Databases*

The proof of this Theorem is by straightforward induction on the structure of the queries in both directions.

Thanks to this equivalence, Relational Algebra can benefit from the numerous results on FO. One such classic result is of particular interest to us as it will be the main motivating factor of the need for graph-specific languages: graph reachability is not expressible in FO.

For now, we define a *graph* G as a set of *vertices* V and a set of *edges* E where an edge is an ordered pair of vertices (see section 2.3 for the definition of property graphs which will be the model used in the rest of this thesis). The problem of graph reachability asks whether there exists a *path* between two fixed vertices v_s and v_t, i.e. a sequence of edges $e_0, \dots e_n$ such that the the second element of e_i is the first element of e_{i+1}, e_0 starts in v_s and e_n ends in v_t.

Definition 2.2.3 — The Reachability Problem

Problem	Reachability
Input	Graph $G = (V, E)$, two vertices u, v in V
Output	`true` if there exists a path from u to v in G, `false` otherwise

Intuitively, this problem cannot be solved in FO because we would need infinitely many queries, one for each possible length of path. Assuming our edges are stored in some relational table E with schema $S(E) = \{v_1, v_2\}$, we can write a query to answer the question "Is there a path of length 1 between v_1 and v_2?": $\exists x, y\ E(x, y) \wedge x = v_1 \wedge y = v_2$. We can do the same for a path of length 2: $\exists x, y, z\ E(x, z) \wedge E(z, y) \wedge x = v_1 \wedge y = v_2$. By iterating this construction, i.e. by adding more and more existentially quantified intermediate variables, we may write a query for any fixed length path. But notice that, no matter the query, we can always build a graph in which v_1 and v_2 are further apart than the longest path we have a query for.

Theorem 2.2.4 — FO and the Reachability problem

There is no FO query Q such that for all graphs $G = (V, E)$ and vertices $u, v \in V$, $[\![Q(u, v)]\!]_G = true$ if and only if there a path from u to v in G. In other words, the Reachability problem is not expressible in FO [19, 21].

[19]: Arenas et al. (2022), *Database Theory*
[21]: Libkin (2004), *Elements of Finite Model Theory*

The proof of this Theorem, presented for example in [21], relies on the notion of "locality", which formalizes the above idea that FO can only touch elements finitely many steps away from the starting point. More generally, this entails that any inherently recursive problem is not expressible in FO [2].

2: Recursive queries are however expressible in SQL since 1999 but often involve many joins, usually one per recursive iteration. As this is a costly operator, we will explore other ways of expand the expressive power of FO.

Complexity

Some queries are easy, for example "does this element belong to the database?" can be answered by reading the whole database once. Other queries are harder, for example "find all pairs of people of the same age" clearly requires some memory to remember everyone's age. And some queries cannot be answered at all, for example "is there a set of (integer) elements $x_1, \ldots, x_k$ from $\mathbb{Z}$ such that $f(x_1, \ldots, x_k) = 0$" where f is a multivariate polynomial (this is called the Diophantine equation problem and is known to be undecidable [25]). The study of complexity tries to classify query languages into groups where all elements are roughly as hard to evaluate as the others. The complexity of a language is that of its hardest query, i.e. we always study the worst-case scenario.

[25]: Jones (1980), "Undecidable diophantine equations"

The complexity class of a query language does not depend only on the language itself, but also on the type of problem we are trying to solve. For example, it could be easier to answer whether there exists a pair of people of the same age than to list all pairs of such people. The first kind of problem is called a *decision problem*, as we are trying to decide whether a property is true or not, the second is an *enumeration problem*, as we want to enumerate all the answers without repetition. These are

the two problems that we will study in the rest of this thesis and we present them now.

In the context of databases, the decision problem is called the query evaluation problem and the goal is to determine whether the input tuple is an answer to the input query on the input database. The complexity of the more intuitive problem of simply outputting all the answers is often a direct consequence of the complexity of query evaluation as we can enumerate all possible candidates and check whether each one is, in fact, an answer.

[19]: Arenas et al. (2022), *Database Theory*

We follow [19] to define, for any language $\mathscr{L}$, the *query evaluation problem* for $\mathscr{L}$ as follows.

Definition 2.2.5 — Query evaluation

Problem	$\mathscr{L}$-evaluation
Input	Query $Q \in \mathscr{L}$, database $\mathscr{D}$, tuple $\bar{a}$ over Const
Output	`true` if $\bar{a} \in Q(\mathscr{D})$, `false` otherwise

Studying query evaluation is usually the starting point for understanding any language. In section 2.4, we will give the complexity of query evaluation for all considered languages as, although this characterization is quite broad and only considers the worst-case, it gives a first idea of how these languages compare.

To give an idea of the kinds of results one can expect when studying the query evaluation problem for database query languages, we now recall the textbook result for Relational Algebra.

Example 2.2.6 — Relational algebra evaluation complexity

For the language of relational algebra, the complexity of query evaluation is PSPACE-complete [21].

[21]: Libkin (2004), *Elements of Finite Model Theory*

[24]: Hopcroft et al. (2007), *Introduction to automata theory, languages, and computation, 3rd Edition*

Hardness is proved by reduction from the Quantified Boolean Formula problem, the canonical complete problem for PSPACE [24], which asks whether the input quantified boolean formula evaluates to `true`. Membership is given via a straightforward algorithm that tries to assign a value to each variable.

As PSPACE is considered an "intractable" class, i.e. its problems have a good chance of timing-out in practice, restrictions of FO have been investigated to find "well-behaved" classes, i.e. "tractable" ones. One particular class, that of *Conjunctive Queries* (CQs), is now considered the basic building block of all the others as it corresponds to the most frequently asked queries in real life. A conjunctive query is an FO formula in which the only allowed operators are the conjunction $\wedge$ and the existential quantifier $\exists$.

Theorem 2.2.7 — Conjunctive Query Evaluation

For the language of conjunctive queries, the complexity of query evaluation is NP-complete [26].

[26]: Chandra et al. (1977), "Optimal Implementation of Conjunctive Queries in Relational Data Bases"

Hardness can be proved by reduction from the 3-colorability problem,

which asks whether there exists a partition of the vertices of a graph $G = (V, E)$ into three sets A, B and C such that for all edges $(u, v) \in E$, the vertices u and v do not belong to the same set. The algorithm for membership is a series of guesses for each variable of the query [21].

[21]: Libkin (2004), *Elements of Finite Model Theory*

Of course, the class of conjunctive queries is rather restricted with regards to expressive power so it is common to consider larger classes. A frequent extension is to *Unions of Conjunctive Queries* (UCQs), in which queries are defined as union of CQs. This has no effect on complexity of query evaluation as each CQ can be evaluated separately, but it can make other problems more difficult (notably query enumeration). Other extensions exist, such as negation or allowing inequality in conditions, but are not within the scope of this thesis.

Notice that in the above definition of query evaluation, the query, database and tuple are all three given as input. The resulting complexity is therefore called the *combined complexity*, as it combines both database and query in the input. Since, in practice, the query is usually much smaller than the database, we also study the problem of query evaluation for a fixed query and call its complexity *data complexity*. Formally, we define this as follows.

Definition 2.2.8 — Query evaluation for a fixed query

Problem	Q-evaluation
Input	Database $\mathscr{D}$, tuple $\bar{a}$ over Const
Output	`true` if $\bar{a} \in Q(\mathscr{D})$, `false` otherwise

By design, data complexity is generally lower than, and rarely equal to, the corresponding combined complexity for a particular problem. For RA, this gap is particularly large.

Example 2.2.9 — RA evaluation data complexity

For the language of relational algebra, the data complexity of query evaluation is AC^0 [21].

[21]: Libkin (2004), *Elements of Finite Model Theory*

As the database is by definition finite, we can explicitly construct the whole AC^0 circuit which corresponds to the FO formula equivalent to the query by replacing each existential (resp. universal) quantifier by a disjunction (resp. conjunction) over all possible values. As all other quantifiers have a corresponding boolean gate, we get the desired result.

The second problem we consider is query enumeration the goal of which is to list all tuples that satisfy the query without repetition. Given a language $\mathscr{L}$, we define the query enumeration problem as follows.

Definition 2.2.10 — Query enumeration

Problem	$\mathscr{L}$-enumeration
Input	Query $Q \in \mathscr{L}$, database $\mathscr{D}$
Output	Enumerate all tuples $\bar{a}$ such that $\bar{a} \in [\![Q(\mathscr{D})]\!]$ without repetition

As for query evaluation, we can define the query enumeration problem for a fixed query Q.

The interesting measure of complexity for query enumeration is slightly different than for query evaluation. Instead of counting the total time (and space) spent computing the final answer, we will measure the ressources spent before the first answer, we call this step the "precomputation phase", and also the ressources spent between two consecutive answers, we call this the "delay". For example, if the answers to a particular query can be enumerated in polynomial delay, the time spent in both the precomputation phase and the delay phase must be at most polynomial in the size of the input (here the query and the database).

To the best of our knowledge, the complexity of query enumeration for RA (or even CQs) is still an open question.

Note Another popular problem to study is *query containment*: given two queries Q_1 and Q_2 are all answers to Q_1 also answers to Q_2? This is equivalent to asking whether $[\![Q_1]\!]_{\mathscr{D}} \subseteq [\![Q_2]\!]_{\mathscr{D}}$ for all databases $\mathscr{D}$? This problem is strongly connected to optimisation questions as, in some cases, a positive answer means that a complicated, hard to evaluate query can be rewritten into a simpler one.

2.3 Graph databases

In essence, a graph is a collection of nodes connected by edges. This model is used to represent interconnected data in many, if not all, domains and communities. In biology, chemistry and physics, graphs may describe the structure of particles, atoms, molecules (and so on) and their interactions. In engineering, they explain how an object works and its internal structure. In social sciences, they show the links between people, factions, countries, etc. According to Manuel Lima, one of the first known graphs is an illustration of the biblical genealogy in the shape of an abstract tree, by Stephanus Garsia Placidus, from the 11th century [27]. It is no surprise then, that the study of these objects has motivated mathematicians since at least 1736 when Leonhard Euler solved the problem of the Seven Bridges of Königsberg. In the almost 300 years since, Graph Theory has become a many-faceted discipline and an integral part of mathematics and computer science.

[27]: Lima (2014), *The book of trees : visualizing branches of knowledge*

The formal definition of a graph depends on the level of detail required by the application. The simplest model is arguably the *undirected graph*, in which nodes are a set N and edges are a set of unordered pairs from $N \times N$. When the edges are replaced by a set of ordered pairs from $N \times N$, we say that the graph is *directed*. If two nodes can have multiple edges between them, the set is replaced by a multiset of pairs of nodes, or, equivalently, a set of "ids" and a function associating each id with a pair of nodes. This type of graph is called a *multigraph*. Extra information can be affixed to nodes and edges by adding labels, usually from a finite alphabet, or weights, from the natural numbers, integers, real numbers or any mathematical structure. In the first case, we say that the graph is *labelled*, in the second that it is *weighted*.

The first graph-based database model was the Network Model. Introduced in 1969 by the CODASYL Data Base Task Group, this model stores data as records that can be linked to one another under the condition that the overall structure satisfies the graph-like schema. In previous

"hierarchical" models, the schema was strictly hierarchical (hence the name), i.e. a tree, and thus did not allow children nodes to have more than one parent node. As many models predating the relational one, the Network model suffered from its low-level interface that required users to know the structure of the data in-memory and its use stopped in the 1980s. The idea of using a graph structure survived in the background, either as smaller projects or add-ons to other kinds of databases until the 2010s when it came back into focus. For more information on the historical context and a comparison of the different models see the survey of Renzo Angles and Claudio Gutierrez [4].

[4]: Angles et al. (2008), "Survey of graph database models"

Today, the graph database world is divided in two. On one side are property graphs, a model in which graphs are enhanced with as much information as needed: directed and undirected edges can co-exist in the same graph, nodes and edges have arbitrarily many labels and properties. This model was popularized by Neo4j in 2010 and has been implemented in other products such as Amazon Neptune, JanusGraph, ArangoDB and Apache TinkerPop. This model was standardized in 2023 by ISO, the organisation also responsible for SQL (among countless other standards). This is the focus of this thesis.

On the other side is the Resource Description Framework, a W3C standard in which everything is a triple. The graph can be extracted from these triples by viewing the middle element as an edge between the other two. This structure greatly facilitates the use of a schema, which can be integrated directly with the rest of the data, but makes properties harder to model and query. This is the model used in "Semantic Web" applications such as DBPedia, YAGO, Wikidata and Creative Commons. In subsection 2.3.2, we will show that these two models are "hardly" comparable as translation from one to the other can cause an exponential blow-up. Nevertheless, results about one model can often be directly applied, or at least easily translated, to the other so, unless necessary, the distinction will not be made when citing works focusing on RDF.

In the rest of this section we present the property graph model in detail and compare it to RDF.

2.3.1 Property Graphs

In a property graph, a node represents a "concept", either physical, such as a person or a soup, or immaterial, such as Polish cuisine. Edges describe the "relationships" between these concepts, like family relations or belonging of the soup to the cuisine. In general, for a sentence of the form "subject verb object", the subject and object will be modeled as nodes and the verb as an edge.

A *directed edge* represents an asymmetric relationship, such as belonging (the soup belongs to the cuisine, not the other way around). In this case, we say that the edge has a *source*, from which it starts, and a *target*, to which it goes. In the previous example, the soup is the source, and Polish cuisine is the target. A *undirected edge* represents a symmetric relationship, such as two dishes going well together. In this case, we say the the edge has two *endpoints* without a distinguished order or direction. If the source is the same as the target, or if both endpoints are the same, we say that the edge is a *self-loop*.

Labels usually represent the category of the concept or relationship. To

distinguish between two elements of a same category, the particularities of graph elements can be specified by *properties*. For example, Alice and Bob of figure 2.1, would both get label `Person` but could be distinguished by the value of the property `name`.

Nodes and *edges* are sets of ids that are linked together by functions, giving the model flexibility. The source and target nodes of a directed edge are given by the functions src and tgt respectively. The endpoints of an undirected edge are given as pairs of nodes by the function endpoints. The function λ returns the set of labels for each id while δ does the same for properties.

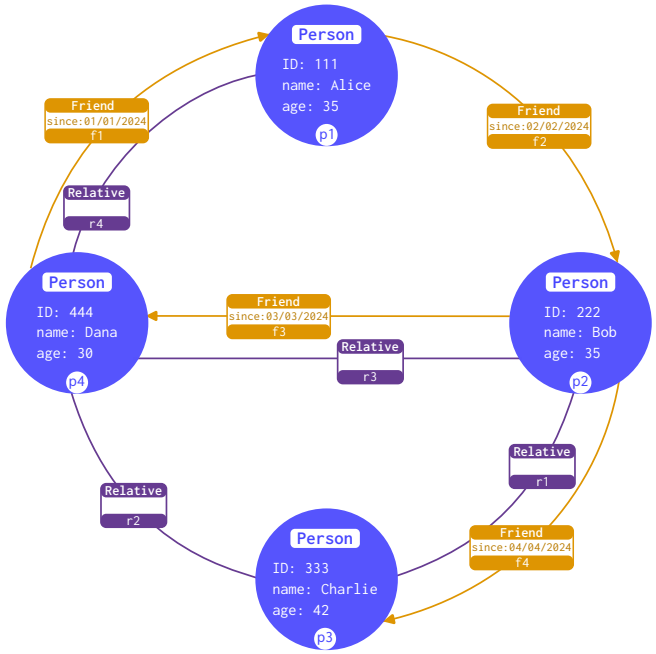


Figure 2.3: Alice's friends and relatives as a property graph

We assume disjoint countable sets $\mathcal{N}, \mathcal{E}_d, \mathcal{E}_u$ of node, directed, and undirected edge ids, $\mathcal{L}$ of labels and $\mathcal{K}$ of keys.

Definition 2.3.1 — Property Graph

A property graph is a tuple $G = \langle N, E_d, E_u, \lambda, \text{endpoints}, \text{src}, \text{tgt}, \delta \rangle$ where

- $N \subset \mathcal{N}$ is a finite set of node ids used in G;
- $E_d \subset \mathcal{E}_d$ is a finite set of directed edge ids used in G;
- $E_u \subset \mathcal{E}_u$ is a finite set of undirected edge ids used in G;
- $\lambda : N \cup E_d \cup E_u \to 2^{\mathcal{L}}$ is a labelling function that associates with every id a (possibly empty) finite set of labels from $\mathcal{L}$;
- $\text{src}, \text{tgt} : E_d \to N$ define source and target of a directed edge;
- $\text{endpoints} : E_u \to 2^N$ so that $|\text{endpoints}(e)|$ is 1 or 2 define endpoints of an undirected edge;
- $\delta : (N \cup E_d \cup E_u) \times \mathcal{K} \to \text{Const}$ is a partial function that associates a constant from Const with an id from $(N \cup E_d \cup E_u)$ and a key from $\mathcal{K}$.

We denote by $\mathbb{G}$ the set of all property graphs.

Example 2.3.2

The graph represented in figure 2.3 encodes the same data as the database shown in figure 2.1. In this representation, the circles depict the nodes, the purple lines are undirected edges and the yellow arrows are directed edges. The choice of translating the Friend relation as directed edges, and the Relative relation as undirected edges is arbitrary and many other translations are possible.

Nodes and edges can be assembled together into paths that go from one node to the next by following the edges that connect them. Formally, a *path* is an alternating sequence of nodes and edges that starts and ends with a node, that is, it is a sequence of the form

$$p = u_0 e_1 u_1 e_2 \cdots e_n u_n \ ,$$

where $u_0, \ldots, u_n$ are nodes and $e_1, \ldots, e_n$ are (directed or undirected) edges. Note that we allow $n = 0$, in which case the path consists of a single node and no edges. As for directed edges, we denote u_0 as $\mathrm{src}(p)$ and u_n as $\mathrm{tgt}(p)$; we also refer to u_0 and u_n as the path's *endpoints*. The *length* of a path p, denoted $\mathrm{len}(p)$, is n, i.e., the number of occurrences of edge ids in p. We call the path of length zero (composed of a single node) the *edgeless path*. We spell paths explicitly as $\mathrm{path}(u_0, e_1, u_1, \cdots, e_n, u_n)$. Whenever a node repeats in the path, i.e $\exists i, j \ u_i = u_j$, we say that the path contains a *cycle* composed of all the elements between u_i and u_j. We denote the set of all paths by Paths.

Given a graph G, a path in G is a path such that each edge in it connects the nodes before and after it in the sequence.[3] Formally, it is a path $\mathrm{path}(u_0, e_1, \ldots, e_n, u_n)$ such that at least one of the following holds for each $i \in [n]$:

(a) $\mathrm{src}(e_i) = u_{i-1}$ and $\mathrm{tgt}(e_i) = u_i$ in which case e_i is as a *forward edge* in the path;
(b) $\mathrm{src}(e_i) = u_i$ and $\mathrm{tgt}(e_i) = u_{i-1}$ in which case e_i is a *backward edge* in the path;
(c) $\mathrm{endpoints}(e_i) = \{u_{i-1}, u_i\}$ in which case e_i is an *undirected edge* in the path.

Here, both (a) and (b) can be true at the same time in the case of a directed self-loop, but (c) is not compatible with neither (a) nor (b). By $\mathrm{Paths}(G)$ we denote the set of paths in G. Notice that $\mathrm{Paths}(G)$ can be infinite.

Two paths $p = \mathrm{path}(u_0, e_0, u_1, ldots, u_k)$ and $p' = \mathrm{path}(u'_0, e'_0, u'_1 \ldots, u'_j)$ *concatenate* if $u_k = u'_0$, in which case their concatenation is defined as

$$p \cdot p' = \mathrm{path}(u_0, e_0, u_1 \ldots, u_k, e'_0, u'_1 \ldots, u'_j).$$

Note that if one of the paths consists of a single node, then it is a unit of concatenation and does not change the result. That is, $p \cdot \mathrm{path}(u)$ is defined iff $u = u_k$, in which case it equals p (likewise for $\mathrm{path}(u) \cdot p$ and $u = u_0$).

3: As is usual in the graph database literature [9, 12, 28, 29], we use the term path to denote what is called "walk" in the graph theory literature [30].

2.3.2 Short digression: The Resource Description Framework (RDF)

Originally standardized by the W3C in 1999 as a metadata description language, RDF has since become the most popular language for ressource description on the Web. RDF data holds information about *IRIs* (short for *International Resource Identifier*), which serve as identifiers for all resources. A resource can be any object, person or concept, physical or immaterial, real or abstract. IRIs can be assigned to their resource either by authorities such as the W3C, DBPedia or Wikidata, or by any person or organisation by publishing a document describing what each IRI refers to [31]. We denote the (countably) infinite set of possible IRIs by Res.

[31]: Hartig et al. (2024), *RDF 1.2 Concepts and Abstract Syntax*

To link resources with their values (like the name or birthday date of a person), RDF data also refers to *literals*, which can be either a string, a number or a date [31]. We denote the (countably) infinite set of literals by Lit. If a resource has a relationship to an unknown (or unimportant) entity, this entity can be replaced by a *blank node* [32]. We denote the (countably) infinite set of blank nodes by Blnk.

[31]: Hartig et al. (2024), *RDF 1.2 Concepts and Abstract Syntax*

[32]: Schreiber et al. (2014), *RDF 1.1 Primer*

IRIs, literals and blank nodes are assembled together into statements of the form "subject predicate object" called triples. In such a statement, the subject is the resource of interest, the object is another resource and the predicate encodes the nature of the relationship between the subject and the object.

Definition 2.3.3 — The RDF model [33]

An *RDF triple* is a tuple (s, p, o) such that s is an IRI or blank node from $\mathsf{Res} \cup \mathsf{Blnk}$, p is an IRI from Res, and o is an IRI, blank node or literal from $\mathsf{Res} \cup \mathsf{Blnk} \cup \mathsf{Lit}$. We call s the *subject*, p the *predicate* and o the *object* of the triple.
An RDF triple can be represented as a p-labelled edge from the node s to the node o, i.e. $s \xrightarrow{p} o$. An RDF graph is a set of RDF triples.

Example 2.3.4

Assume that there exists a document, say at `http://www.irif.fr/rdf`, specifying that Alice and Dana from Example 2.1.4 are identified by the IRIs `:Alice` and `:Dana` respectively, and the concepts of having an id, name, age and friend are identified by `:hasID`, `:hasName`, `:hasAge` and `:Friend`, the following RDF triples (written in the Turtle syntax [34]) encode a subset of the information presented in figure 2.1.

[34]: Carothers et al. (2014), *RDF 1.1 Turtle*

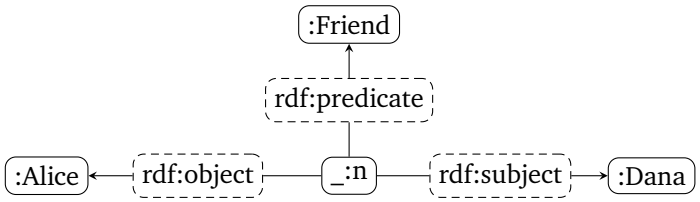
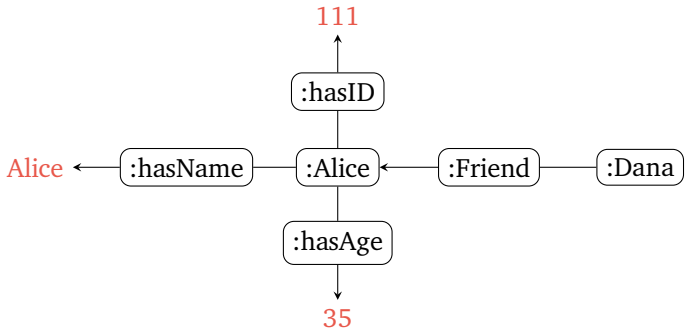


Figure 2.4: An illustration of the naïve approach to RDF reification

```
@prefix : <http://www.irif.fr/rdf> .
:Alice :hasID "111" .
:Alice :hasName "Alice" .
:Alice :hasAge "35" .
:Dana :Friend :Alice .
```

The same set of triples can be represented as the following graph.



One major difference between the RDF model and the property graph model is of course the lack of properties on the RDF graph elements. Indeed, in order to add the `Since` attribute to the last triple of Example 2.3.4 (the one describing the friendship between Dana and Alice), we need to extend the language with the ability to make statements about statements, also known as *reification*. This feature has been studied and discussed in depth since the creation of RDF, and has led to a number of proposals of extensions (see [35] for a survey of the different proposals and [36] for a case study on Wikidata).

[35]: Tomaszuk et al. (2020), "RDF 1.1: Knowledge Representation and Data Integration Language for the Web"

[36]: Hernández et al. (2015), "Reifying RDF: What Works Well With Wikidata?"

The naïve approach, the one that stays within the established framework of RDF and does not require any additions to the language, amounts to transforming the edge `:Friend` into a blank node `_:n` and linking `_:n` with `:Dana` via the special predicate `rdf:subject`, with `:Alice` via the special predicate `rdf:object` and with `:Friend` via the special predicate `rdf:predicate` (the resulting topology is illustrated in Figure 2.4). Since `:Friend` has now become the blank node `_:n`, we can use it as a subject in new triples and add additional information about the relationship such as the date via a new triple `_:n <:since> "01/01/2024"`. This approach has three considerable drawbacks: first, as each affected edge is replaced with three fresh edges, the size of the graph can increase dramatically; second, the model becomes more complex and less intuitive; third, it breaks all queries based on the previous model.

In [37], Hartig introduces *RDF** [4], which he described as "an extension of RDF with a notion of nested triples". In the new notation `<< triple >>`, one can use a triple as a subject, or object, of another triple. This proposal solves two of the three problems of the naïve approach: the size of the graph is unchanged and only one new triple is added for each new piece of information; and the queries are still compatible since the underlying data is unchanged. The complexity of the model is in-

[37]: Hartig (2017), "RDF* and SPARQL*: An Alternative Approach to Annotate Statements in RDF"

4: the name has since been changed to RDF-star

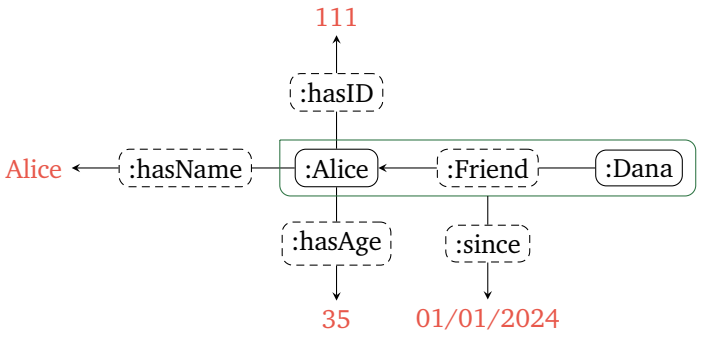
creased due to the added levels of nesting but queries about the added metadata remain readable and understandable.

Example 2.3.5

The information of Example 2.3.4 together with the `Since` attribute on the relationship between Dana and Alice, can be encoded as follows in RDF*.

```
@prefix : <http://www.irif.fr/rdf> .
:Alice :hasID "111" .
:Alice :hasName "Alice" .
:Alice :hasAge "35" .
<< :Dana :Friend :Alice >> :since "01/01/2024" .
```

In the graph representation of the above tuples, we display the nesting of the triple `<:Dana> <:Friend> <:Alice>` as a dashed box around the edge and both endpoints.



But property graphs are not necessarily more expressive than RDF. In fact, the very thing that RDF was designed for, metadata and schemas, cannot be easily translated into a property graph. For example, if we wanted to add the information that both Friend and Relative are types of "Relationships", so we can write queries asking for people with any kind of "Relationship" between them, we would need to either reify the graph to transform the Friend and Relative edges into nodes, or add a new label to each Friend or Relative edge. In RDF, this information can be encoded by simply using the IRI encoding Friend (or Relative) as the subject of a triple with predicate `:type_of` and object `:Relationship`.

[38]: Angles et al. (2022), *PG-Schema: Schemas for Property Graphs*

The current solution consists in storing schema information separately from the graph. In [38], the authors define the language PG-SCHEMA to encode the schema information in a way compatible with the Graph Query Language, which we introduce in chapter 3. The idea is to find all graph elements (nodes and edges) satisfying some constraints and automatically add the schema information if it is missing. In the above example, PG-SCHEMA would automatically add the label "Relationship" to all Friend and Relative edges.

[39]: Angles et al. (2019), "RDF and Property Graphs Interoperability: Status and Issues"

In [39], Angles, Thakkar and Tomaszuk survey and compare the different solutions proposed by the community to transform RDF graphs into property graphs and vice versa. Although the situation has evolved slightly since 2019, namely with the standardisation of the property graph model, their conclusion still holds. The two models are far enough apart that no unanimous decision has been reached on how to reconciliate their differences. For this reason, all mentions of "graph" in the rest of this thesis will refer to property graphs and we do not examine the impact of our results for RDF.

2.4 Querying property graphs

Even though property graphs can be encoded as a relational database, for example by having a relation for each element of the graph tuple, and can therefore be queried using RA or FO, doing so is often not a satisfactory solution, primarily because of the inability of FO to express graph reachability (as shown in Theorem 2.2.4).

Moreover, the interesting features of a graph database are not its individual elements, its nodes and edges, but rather their *topology*, that is the overall graph structure and the way its nodes and edges are connected to each other. Accordingly, graph query languages need to treat paths as first class citizens and be able to express complex conditions not only on sets of nodes, but also the structure linking them. Typically, these conditions are expressed as regular expressions, a language that arises from automata theory and formal languages, and is used in practice in countless applications such as text search, web scraping, input processing, syntax highlighting and so on.

The class of Regular Path Queries, or simply RPQs, which return pairs of nodes linked by some regular expression, is the simplest class of *navigational queries*, i.e. languages that query the topology of a graph. RPQs have been studied since (at least) 1997 [40], in the context of semistructured data, and most of their properties are by now well understood. This small language can be extended in the usual way, either with conjunction, leading to Conjunctive Regular Paths Queries (CRPQs), or unions of conjunctions, leading to Unions of Conjunctive Regular Paths Queries (UCRPQs). The language of RPQs with the extra ability to traverse edges backwards is called Two Way Regular Path Queries, or 2RPQs, and can again be extended to Conjunctive 2RPQs (C2RPQs) and Unions of Conjunctive 2RPQs (UC2RPQs). Adding a nesting operator, which checks for the existence of branching paths, leads to the class of Nested Regular Expressions. The end of this line of extensions is closed by Regular Queries, which can be seen either as the transitive closure of UC2RPQs, or Datalog applied to graphs.

[40]: Abiteboul et al. (1997), "Regular Path Queries with Constraints"

In another direction, Extended CRPQs add the ability to compare path properties to CRPQs, by changing the regular languages underlying RPQs to regular relations. The formalism that we explore last is that of Regular Data Path Queries in which the query can access the data values stored in the properties of the graph.

All of these languages describe ways to extract data from the graph, whether it is stored in nodes, edges, paths or more complicated structures. This description usually takes the form of an example ("give me all the paths that look somewhat like this one") or several examples, and instructions on how to combine them. We call these examples *patterns*, and refer to the action of finding the parts of the graph that correspond to these patterns as *pattern matching*.

In the rest of this section, we introduce the class of Regular Path Queries, define its extensions, and cite the main results known for these classes.

Regular Path Queries

To formally define Regular Path Queries, we first need to define regular languages, which can be described either imperatively by finite automata or declaratively by regular expressions.

We start by recalling the definition of Deterministic Finite Automata, for which we follow [23].

[23]: Sipser (1996), "Introduction to the Theory of Computation"

Definition 2.4.1 — Deterministic Finite Automata

A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_o, F)$ where

- Q is a finite set of *states*
- Σ is a finite set of *input symbols*
- $\delta : Q \times \Sigma \to Q$ is the *transition function* that takes as input a state from Q and a symbol from Σ and returns a state in Q.
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is a set of *final states*

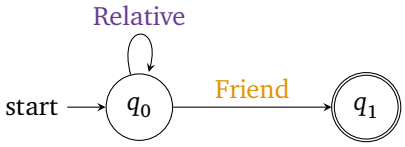
We say that a string $w = w_1 w_2 \dots w_n$ over Σ^n is *accepted* by an automaton $\mathscr{A} = (Q, \Sigma, \delta, q_o, F)$ if there exists a sequence of states $r_0 r_1 \dots r_n$ in Q such that

- $r_0 = q_0$,
- $\delta(r_i, w_i) = r_{i+1}$ for each i between 0 and $n-1$ and
- $r_n \in F$

The set of strings recognized by an automaton $\mathscr{A}$ is called the *language* of $\mathscr{A}$ and is denoted by $L(\mathscr{A})$.

Example 2.4.2

The following picture represents a finite automaton on $\Sigma = \{\text{Friend}, \text{Relative}\}$ using the standard notation of states as circles and transitions as labelled arrows. The initial state is pointed to by an arrow from the word `start` and the final states are denoted by a double circle.



This automaton accepts words which start with arbitrarily many Relative symbols and end with exactly one Friend symbol.

There are many ways to modify the above definition to obtain different automata models. For example, one may change the transition function to a transition relation to obtain non-deterministic finite automata (NFA) in which a state can have multiple outgoing transitions labelled by the same letter. In this case, the automaton will follow all possible routes in parallel in the hope that at least one reaches a final state. Another possible modification is to allow so-called ϵ transitions which can be taken without reading a letter. If Σ feels too restrictive, one can authorize the use of regular expressions in transitions. While all these alternatives might seem powerful, it has been known, since the very beginning of automata theory, that all these definitions are equivalent (although translating from one to the other may cause an exponential blow-up in some cases) [24]. Because of this, we refer to an automaton defined in an equivalent way as a finite automaton when the distinction is unimportant.

[24]: Hopcroft et al. (2007), *Introduction to automata theory, languages, and computation, 3rd Edition*

For regular expressions we follow the definition of [24].

[24]: Hopcroft et al. (2007), *Introduction to automata theory, languages, and computation, 3rd Edition*

Definition 2.4.3 — Syntax of Regular Expressions

Given a finite alphabet Σ, REGular EXpressions (*regex*, sometimes *regexp*) are defined by the following grammar:

$$e := \epsilon \mid \varnothing \mid a \mid e + e \mid e \cdot e \mid e^*$$

where a is a letter in Σ.

The *language* of a regular expression, i.e the set of words it describes, is defined inductively as follows.

(Base case) The languages of the constants ϵ, the *empty word*, and $\varnothing$, the *empty set*, are $L(\epsilon) = \{\epsilon\}$ and $L(\varnothing) = \varnothing$ respectively.

(Base case) The language of a letter $a \in \Sigma$ is $L(a) = \{a\}$.

- Given two regular expressions e_1 and e_2, whose languages are $L(e_1)$ and $L(e_2)$ respectively, the language of their *union* $e_1 + e_2$ is the union of both languages, i.e. $L(e_1 + e_2) = L(e_1) \cup L(e_2)$.
- Given two regular expressions e_1 and e_2, whose languages are $L(e_1)$ and $L(e_2)$ respectively, the language of their *concatenation* $e_1 \cdot e_2$ is the set of all words composed of one word from $L(e_1)$ followed by a word from $L(e_2)$, i.e. $L(e_1 \cdot e_2) = L(e_1)L(e_2)$.
- Given a regular expression e whose language is $L(e)$, the language of the closure of e under string concatenation e^* is the set of all words (finite and infinite) obtained by concatenating words from $L(e)$ (possibly by taking the same word multiple times), i.e. $L(e^*) = \bigcup_{i \geq 0} L(e)^i$ where $L(e)^0 = \{\epsilon\}$, $L(e)^1 = L(e)$ and $L(e)^i = \underbrace{L(e) \cdots L(e)}_{i \text{ times}}$ for $i > 1$.

Example 2.4.4

Assuming the same alphabet as in Example 2.4.2, namely $\Sigma = \{\text{Friend}, \text{Relative}\}$, the following regular expression describes the same language as the automaton of that same example.

$$\text{Relative}^* \cdot \text{Friend}$$

Even though they look different, and describe words in very different ways, it turns out that regular expressions and finite automata define exactly the same set of languages. We say that a language is regular if it can be described by a regular expression (or a finite automaton).

Theorem 2.4.5 — Regex and DFA equivalence

Given a regular expression e, there exists a deterministic finite automaton $\mathscr{A}$ such that $L(e) = L(\mathscr{A})$.
Given a deterministic finite automaton $\mathscr{A}$, there exists a regular expression e such that $L(\mathscr{A}) = L(e)$.

Now that we have defined regular languages, we can also define Regular Path Queries. Essentially, RPQs specify what kind of path we want to find between two nodes. As previously mentioned, these constraints are expressed as a regular language over edge labels. Formally, we say that the *label* of a path $p = \text{path}(u_0, e_1, \ldots, e_n, u_n)$ is the concatenation of its edge labels, i.e. $\lambda(p) = \lambda(e_0) \cdot \ldots \cdot \lambda(e_n)$ [5].

5: This standard definition assumes that each edge has exactly one label. This is easily extended to allow multiple labels by considering a set of words instead of a single word. For the sake of clarity, we keep this assumption for the rest of this section.

Definition 2.4.6 — Regular Path Query

Given a finite alphabet Σ, a Regular Path Query (RPQ) is an expression $Q = x \xrightarrow{L} y$ where L is a regular language over Σ.

The semantics of an RPQ Q evaluated on a graph database G, denoted by $[\![Q]\!]_G$, is the set of pairs of nodes in G such that there exists a path labelled by a word in L between them.

Example 2.4.7

The following RPQ Q is defined by the regular expression of Example 2.4.4 and returns all pairs of nodes connected by a path consisting of, first, an arbitrary amount of Relative labelled edges, and ended by a Friend labelled edge.

$$Q(x, y) := (x) \xrightarrow{\text{Relative}^* \cdot \text{Friend}} (y)$$

The semantics of Q, when evaluated on the graph represented in Figure 2.3, is the set of all pairs of nodes in the graph as there is always a way, from any fixed node, to reach any other node (including itself) by taking Relative labelled edges and a final Friend labelled edge, and so computes the set of friends of the family.

The query evaluation problem for RPQs is called the RPQ Evaluation problem. As the semantics of an RPQs is a set of pairs of nodes, the input tuple is replaced by a pair of candidate nodes and the algorithm must answer `true` if there exists path between the two nodes labelled by the regular expression underlying the RPQ.

Definition 2.4.8 — The RPQ Evaluation Problem

Problem	RPQ-evaluation
Input	RPQ Q, graph G, two nodes $s, t \in N_G \times N_G$
Output	`true` if there exists a path p from s to t in G labelled by a word in Q, i.e. $s, t \in [\![Q]\!]_G$, `false` otherwise

No matter which variation one studies, whether one changes RPQs to a different language or applies other constraints on the path, the lower bound of the RPQ Evaluation problem will always be (at least) NLOGSPACE, as the problem which asks whether there exists a directed path between two input nodes is itself NLOGSPACE-complete (for a proof of this result, see for example [23]).

[23]: Sipser (1996), "Introduction to the Theory of Computation"

Theorem 2.4.9 — Complexity of RPQ Evaluation

The data complexity of the RPQ Evaluation problem is NLOGSPACE-complete [9] and its combined complexity is in PTime [12].

[9]: Barceló (2013), "Querying graph databases"

[12]: Wood (2012), "Query languages for graph databases"

One possible algorithm satisfying these bounds is the following. First, if the regular language is specified as a regular expression, construct an equivalent automaton $\mathscr{A}$ [6] (otherwise we call $\mathscr{A}$ the provided automa-

6: To avoid the potential exponential blow-up, the constructed automaton is allowed to be non-deterministic.

ton). Add a distinguished initial state q_s and a distinguished final state q_f to $\mathscr{A}$ and connect both states to the old initial and final states accordingly. Construct the product automaton $\mathscr{A} \times G$ of $\mathscr{A}$ and the graph G. For this, we see G as an automaton over Σ, i.e. the nodes become states, the edges become transitions specified by the label, the initial state is s and the unique final state is t. The states of $\mathscr{A} \times G$ are identified by all the possible pairs in which the first element is a state in $\mathscr{A}$ and the second is a state in G. There is a transition between two states (x, y) and (x', y') of $\mathscr{A} \times G$ if there is a transition from x to x' in $\mathscr{A}$ and from y to y' in the automaton obtained from G. Lastly check that the language of the resulting automaton $\mathscr{A} \times G$ is non-empty, i.e. contains at least one word. This construction preserves both the structures of G and of $\mathscr{A}$, ensuring that, if there is a word in the language, it corresponds to both a path in G and a word in $L(\mathscr{A})$.

A different popular construction consists in building the cross-product graph, instead of the cross-product automaton, and checking whether any node containing s can reach a node containing t. In this case, it is the automaton that is viewed as a graph, its states becoming nodes and its transitions becoming edges instead of the other way around. As finite automata non-emptiness checking and graph reachability are both in PTime, these two constructions are equivalent.

The RPQ Enumeration problem is defined analogously to the enumeration problem, once again with the tuples replaced by pairs of nodes.

Definition 2.4.10 — RPQ Enumeration Problem

Problem	RPQ-enumeration
Input	RPQ Q, graph G
Output	Enumerate all pairs of nodes s, t in G such that $(s, t) \in [\![Q]\!]_G$, without repetition

As for RPQ Evaluation, the complexity of RPQ Enumeration is directly linked to the problem of enumerating the words that belong to some regular language, which can be done in polynomial delay. This result was first proved by Erkki Mäkinen for Deterministic Finite Automata [41], then later extended to the non-deterministic context by Margareta Ackerman and Jeffrey Shallit [42].

[41]: Mäkinen (1997), "On Lexicographic Enumeration of Regular and Context-Free Languages"

[42]: Ackerman et al. (2009), "Efficient enumeration of words in regular languages"

Theorem 2.4.11

The RPQ Enumeration problem can be solved with output in polynomial delay [43].

[43]: Martens et al. (2018), "Evaluation and Enumeration Problems for Regular Path Queries"

The proof of the above Theorem, relies on the same construction as for the evaluation problem. Once the product-automaton is built, we can simply run the enumeration algorithm for finite automata.

The literature on RPQs is vast, varied and still on-going. The results presented in this section represent a fraction of the current knowledge. The fine-grained complexity of RPQ Evaluation and RPQ Enumeration has been studied in depth by Katrin Casel and Markus L. Schmid in [44]. Dario Colazzo and Carlo Sartiani devised a typing system which simplifies query optimisation [45]. Anil Pacaci, Angela Bonifati and M. Tamer Özsu have developed an algorithm for RPQ evaluation over streaming graphs [46]. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenz-

[44]: Casel et al. (2023), "Fine-Grained Complexity of Regular Path Queries"

[45]: Colazzo et al. (2015), "Typing regular path query languages for data graphs"

[46]: Pacaci et al. (2020), "Regular Path Query Evaluation on Streaming Graphs"

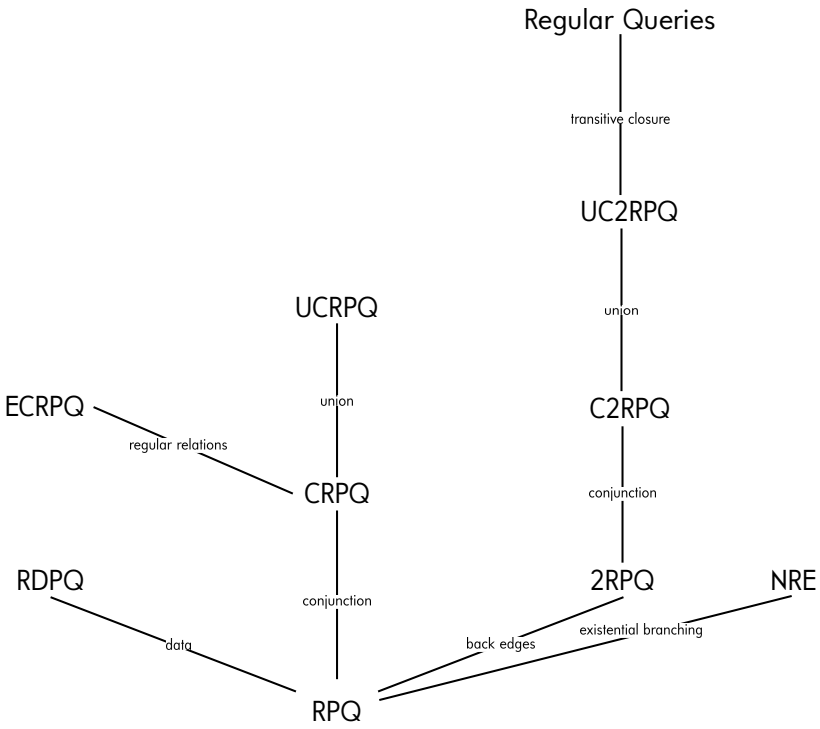


Figure 2.5: An illustration of how the different languages presented section 6 relate to one another

erini and Moshe. Y. Vardi study the containment problem for RPQs in [47] and the RPQ Evaluation problem restricted by views in [48].

[47]: Calvanese et al. (2003), "Reasoning on regular path queries"

[48]: Calvanese et al. (2000), "Answering Regular Path Queries Using Views"

Extensions of RPQs

RPQs can be extended in many ways, from conjunctions and unions, to nesting and data. In this subsection, we present the handful extensions to which we will compare the various languages of the next chapters. Figure 2.5 shows an overview of the 4 branches of extensions considered here.

Conjunction and Union The first extension we consider does not change the structure of the queries nor does it add to, or remove anything from the allowed regular languages. Rather, it is a way to compose multiple RPQs in order to build more complicated patterns. By adding conjunctions, we can get more information about the path by adding intermediate variables, connect multiple path fragments into one path-like structure, or even return several disconnected paths. Union introduces a form of choice: the endpoints of a path belong to the set of answers if the path between them satisfies at least one of the queries in the union.

Definition 2.4.12 — CRPQ and UCRPQ

Given a finite alphabet Σ, a Conjunctive Regular Path Query (CRPQ) is a conjunction of RPQs with a distinguished set of variables $\bar{z}$.

$$Q(\bar{z}) = \bigwedge_{1 \le i \le k} x_i \xrightarrow{e_i} y_i \qquad \text{where } \bar{z} \subseteq \bar{x} \cup \bar{y}$$

where $\bar{x} = \bigcup_{1 \le i \le k} \{x_i\}$ and $\bar{y} = \bigcup_{1 \le i \le k} \{y_i\}$

A Union of Conjunctive Regular Path Queries (UCRPQ) is a union of CRPQs which share the same set $\bar{z}$ of distinguished variables.

$$Q(\bar{z}) = \bigcup_{1 \leq i \leq k} Q_i(\bar{z}) \quad \text{where each } Q_i \text{ is a CRPQ.}$$

The semantics of a CRPQ $Q(\bar{z})$ evaluated on a graph database G is the set $\bar{n}$ of tuples of nodes of G such that there exists an assignment from the variables of Q to nodes in G that satisfies each atom of Q and maps elements of $\bar{z}$ to element of $\bar{n}$.

$$\begin{aligned}\llbracket Q(z_1, \dots, z_m) \rrbracket_G = \{(n_1, \dots, n_m) \in N_G^m \mid \exists \mu : (\bar{x} \cup \bar{y}) \to N_G, \\ \forall_{1 \leq i \leq k}, (\mu(x_i), \mu(y_i)) \in \left[\!\!\left[x_i \xrightarrow{e_i} y_i \right]\!\!\right]_G \wedge \\ \forall_{1 \leq j \leq m}, \exists u_j, \mu(z_j) = n_j\}\end{aligned}$$

The semantics of a UCRPQ $Q(\bar{z})$ evaluated on a graph database G is the union of the output for each atom of Q.

$$\llbracket Q(\bar{z}) \rrbracket_G = \bigcup_{1 \leq i \leq k} \llbracket Q_i(\bar{z}) \rrbracket_G$$

Example 2.4.13

The following CRPQ Q returns the pairs of people who are both Relatives and Friends with each other.

$$Q(x, y) := (x) \xrightarrow{\text{Relative}} (y) \wedge (x) \xrightarrow{\text{Friend}} (y)$$

When evaluated on the graph represented in Figure 2.3, the semantics of Q is the set $\{(p1, p4), (p2, p3), (p2, p4)\}$.

We call *CRPQ Evaluation* (resp. *UCRPQ Evaluation*) the problem of query evaluation where the query must be a CRPQ (resp. UCRPQ).

Theorem 2.4.14 — Complexity of CRPQ Evaluation

The data complexity of CRPQ Evaluation remains NLOGSPACE-complete [6, 9] and its combined complexity is NP-complete [11].

For data complexity, the NLOGSPACE algorithm consists of translating the graph intro a relational database and treating the CRPQ as a Conjunctive Query. Since query evaluation for conjunctive queries is in NLOGSPACE in data complexity [26], we get the desired result.

The hardness for combined complexity is a direct consequence of the NP-hardness of query evaluation for conjunctive queries over directed graphs [26]. The idea of the upper-bound algorithm is to guess the assignment for each variable, which requires space linear in the number of variables, and verify whether this candidate does indeed belong to the output, which can also be done in polynomial time.

[6]: Cruz et al. (1987), "A Graphical Query Language Supporting Recursion"
[9]: Barceló (2013), "Querying graph databases"

[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"

[26]: Chandra et al. (1977), "Optimal Implementation of Conjunctive Queries in Relational Data Bases"

Theorem 2.4.15 — Complexity of UCRPQ Evaluation

The complexity of UCRPQ Evaluation remains NLOGSPACE-complete in data complexity [6, 9] and NP-complete in combined complexity.

[6]: Cruz et al. (1987), "A Graphical Query Language Supporting Recursion"
[9]: Barceló (2013), "Querying graph databases"

Both results are of course a consequence of Theorem 2.4.14, as each CRPQ forming the UCRPQ can be checked independently.

Two way queries By extending regular expressions with a special symbol $^-$, representing the ability to traverse a directed edge backwards, i.e. from its target to its source, we obtain the class of 2-way Regular Path Queries. The popularity of this feature in practical language has made it a standard addition in theoretical languages as well.

Given a finite alphabet Σ, we define $\Sigma^\pm$ as the extension of Σ with extra symbols a^- for each $a \in \Sigma$. Given a graph database G, we define $G^\pm$, the *completion* of G, as the graph obtained from G by adding a new edge e^- for each directed edge e in G such that $\mathrm{src}(e^-) = \mathrm{tgt}(e)$, $\mathrm{tgt}(e^-) = \mathrm{src}(e)$, $\delta(e^-) = \delta(e)$ and $\lambda(e^-) = \{a^- \in \Sigma^\pm \mid a \in \lambda(e)\}$.

Definition 2.4.16 — 2RPQ

A Two Way RPQ (2RPQ) is an RPQ over $\Sigma^\pm$.

The semantics of a 2RPQ $Q : x \xrightarrow{L} y$ evaluated on a graph database G is the set of pairs of nodes in $G^\pm$ such that there exists a path labelled by a word in L between them.

Example 2.4.17

The following query Q returns all pairs of people x and y such that y is Friends with x. For example if we fix x to be Alice, we will get the set of all people who consider themselves to be a Friend of Alice.

$$Q(x, y) := (x) \xrightarrow{\text{Friend}^-} (y)$$

When evaluated on the graph represented in Figure 2.3, the semantics of Q is the set $\{(p1, p4), (p2, p1), (p3, p2), (p4, p2)\}$.

We call *2RPQ Evaluation* the problem of query evaluation where the query must be a 2RPQ. Unsurprisingly, the addition of this new feature has no impact on the complexity of query evaluation[7]

7: It does however complicate static analysis problems such as query containment.

Theorem 2.4.18 — Complexity of 2RPQ Evaluation

The complexity of 2RPQ Evaluation remains NLOGSPACE-complete in data complexity and in PTime in combined complexity [6, 9].

[6]: Cruz et al. (1987), "A Graphical Query Language Supporting Recursion"
[9]: Barceló (2013), "Querying graph databases"

As $G^\pm$ can be computed in linear time and logarithmic space in the size of G, both bounds persist from RPQ Evaluation.

As for RPQs, the class of 2RPQs closed under conjunction is called *Conjunctive 2 Way RPQ* (C2RPQ) and the class of C2RPQs closed under

union is called *Union of Conjunctive 2 Way RPQs* (UC2RPQ). The semantics of both classes are defined analogously to CRPQs and UCRPQs respectively. We call *C2RPQ Evaluation* (resp. *UC2RPQ Evaluation*) the problem of query evaluation where the query must be a C2RPQ (resp. UC2RPQ).

Theorem 2.4.19 — C2RPQ and UC2RPQ Evaluation

The complexity of both C2RPQ Evaluation and UC2RPQ Evaluation is NLOGSPACE-complete in data complexity and NP-complete in combined complexity [9, 11].

Again, the bounds are preserved from the problems of CRPQ Evaluation and UCRPQ Evaluation. The combined complexity can be lowered to PTime for three known subclasses. The first is that of queries with semantic tree width at most k (folklore, explicitly stated in e.g. [49]). Unfortunately, the membership problem for that class is in 2EXPSPACE [50]. The second class is that of acyclic UC2RPQs, queries for which the underlying graph is acyclic [11, 14]. Once again, the problem of deciding whether a UC2RPQ is acyclic is untractable, namely EXPSPACE-complete [51]. Relaxing the notion of acyclicity to semantic acyclicity, which is true of queries equivalent to some acyclic query, does retain the PTime query evaluation bound but, once more, testing membership remains EXPSPACE-complete [51].

Apart from query evaluation, these 2-way classes have been studied in the context of the query containment problem, which is known to be EXPSPACE-complete for UC2RPQs in the general case but can be solved in PSPACE for a particular class of well-behaved queries [13]. The related problem of "UC2RPQ boundedness", which asks, given a UC2RPQ whether it is equivalent to a union of UCQs, has also been shown to be EXPSPACE-complete [52].

[49]: Romero et al. (2017), "The homomorphism problem for regular graph patterns"

[50]: Figueira et al. (2023), "Approximation and Semantic Tree-Width of Conjunctive Regular Path Queries"

[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"

[14]: Barceló et al. (2012), "Relative Expressiveness of Nested Regular Expressions"

[51]: Barceló et al. (2016), "Semantic Acyclicity on Graph Databases"

[13]: Figueira (2020), "Containment of UC2RPQ: The Hard and Easy Cases"

[52]: Barceló et al. (2019), "Boundedness of Conjunctive Regular Path Queries"

Nested Regular Expressions Nested Regular Expressions were introduced by J. Pérez, M. Arenas and C. Guitierrez in [53] as a tool to study the RDF language "nSPARQL". As the name suggests, Nested Regular Expressions extend RPQs with a nesting operator []. When a query encounters such an operator, it branches off to check for the existence of a path labelled by the specified regular expression. Once the check is complete, the query goes back to the starting node of the branch and continues on with the rest of the path. We follow [10] for the definition.

[53]: Arenas et al. (2007), "An Extension of SPARQL for RDFS"

[10]: Libkin et al. (2016), "Querying Graphs with Data"

Definition 2.4.20 — NRE

A Nested Regular Expression (NRE) over a finite alphabet Σ is an expression over the extension of regular expressions over Σ with a nesting operator, denoted by [], and inverses, denoted by a^-. The formal syntax is as follows:

$$n := \epsilon \mid a \mid a^- \mid n \cdot n \mid n^* \mid n + n \mid [n] \qquad a \in \Sigma$$

The semantics of an NRE e evaluated over a graph database G is the set of pairs of nodes in G such that there exists a path labelled by a word in e between them. This relation is defined inductively as follows.

$$
\begin{aligned}
[\![\epsilon]\!]_G &= \{(v, v) \mid v \in N\} \\
[\![a]\!]_G &= \{(v, v') \mid \exists e \in E_d, \mathrm{src}(e) = v, \mathrm{tgt}(e) = v', a \in \mathrm{lab}(e)\} \\
[\![a^-]\!]_G &= \{(v, v') \mid \exists e \in E_d, \mathrm{tgt}(e) = v, \mathrm{src}(e) = v', a \in \mathrm{lab}(e)\} \\
[\![n \cdot n']\!]_G &= [\![n]\!]_G \circ [\![n']\!]_G \\
[\![n + n']\!]_G &= [\![n]\!]_G \cup [\![n']\!]_G \\
[\![n^*]\!]_G &= \text{the reflexive transitive closure of } [\![n]\!]_G \\
[\![[n]]\!]_G &= \{(v, v) \mid \exists v', (v, v') \in [\![n]\!]_G\}
\end{aligned}
$$

Example 2.4.21

The following NRE e returns all pairs of people linked by Relative edges such that each intermediate Relative has at least one Friend.

$$e := (\text{Relative} \cdot [\text{Friend}])^*$$

When evaluated on the graph represented in Figure 2.3, the semantics of the query defined by e is the set of all pairs of nodes in the graph except for those with `p3` as the second component (except for `(p3, p3)`) as it does not have any outgoing Friend edges.

We call *NRE Evaluation* the problem of query evaluation where the query must be an NRE.

Theorem 2.4.22 — Complexity of NRE Evaluation

The combined complexity of NRE Evaluation is in time linear in the size of the graph and the query [54].

[54]: Pérez et al. (2010), "nSPARQL: A navigational language for RDF"

The proof of Theorem 2.4.22 relies on the same algorithm as for RPQ Evaluation with an extra step prior to the construction of the product automaton, in which the nodes of the graph are annotated with the set of nested regular expressions to which the label of the path from the start node to this node belongs.

Although NRE Evaluation has the same complexity as RPQ Evaluation, and is easier (under the usual assumptions that PTime $\neq$ NP) than UC2RPQ Evaluation, the expressive power of NREs is higher than that of RPQs and incomparable with that of UC2RPQs.

Theorem 2.4.23 — Expressive power of NREs

Nested Regular Expressions are strictly more expressive than Regular Path Queries. In other words, there exists an NRE expression e such that there is no RPQ query Q for which $[\![e]\!]_G = [\![Q]\!]_G$ on all graphs G [14].
Nested Regular Expressions are incomparable with Conjunctive Two-way RPQs. In other words, there exists an NRE expression e such that there is no C2RPQ query Q for which $[\![e]\!]_G = [\![Q]\!]_G$ on all graphs G and vice versa [14].

[14]: Barceló et al. (2012), "Relative Expressiveness of Nested Regular Expressions"

The NRE of Example 2.4.21, $(\text{Relative} \cdot [\text{Friend}])^*$, is a good example of a query that is not expressible as an RPQ (or even a UC2RPQ). On

the other hand, the "triangle" query $x \xrightarrow{a} y, y \xrightarrow{a} z, z \xrightarrow{a} x$ requires the power of conjunction and cannot be expressed as an NRE.

Regular Queries Just like in the case of relational algebra, Datalog can be used to define the transitive closure of navigational languages. In fact, the most natural fragment of Datalog for graphs, in which the main restriction is that relations must have arity 2, extended with the power to check if a pair belongs to the transitive closure of a relation, corresponds to the transitive closure of UC2RPQs. This language, known as Regular Queries, was first studied in the Knowledge Representation community (under the name "nested positive 2RPQs") [55, 56], and adapted to the context of graph databases by J. L. Reutter, M. Romero and M. Y. Vardi in [57]. Intuitively, Regular Queries make it possible to talk about patterns in which two nodes can be connected by more than one edge and the main path can branch off to check for the existence of arbitrarily complicated sub-paths. Before formally defining Regular Queries, we first need to define the extension of non-recursive Datalog in which rules can refer to the transitive closure of a relation.

[55]: Bourhis et al. (2014), "How to Best Nest Regular Path Queries"
[56]: Bourhis et al. (2015), "Reasonable Highly Expressive Query Languages - IJCAI-15 Distinguished Paper (Honorary Mention)"
[57]: Reutter et al. (2017), "Regular Queries on Graph Databases"

An *extended Datalog rule* is an expression $R(\bar{u}) \leftarrow R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ where R is a relation name and each R_i is either a relation name or an expression $S^+(y, y')$ where S is a relation name. A pair of elements (u, u') belongs to $[\![S^+(y, y')]\!]$ if it is in the transitive closure of S.

An *extended Datalog* program is a finite set of extended Datalog rules.

Definition 2.4.24 — Regular Query [57]

Given a finite alphabet Σ, a Regular Query (RQ) is a non-recursive extended Datalog program over Σ with a *distinguished relation* `Ans` such that all relations (except possibly `Ans`) have arity 2.

Since we are in the graph database setting, each extensional relation name represents an edge label of our graph. Like for RPQs and their extensions, we build paths either in small steps by using a "pure" (as in without transitive closure) relation name, or in big steps by using the transitive closure construction.

Example 2.4.25

Assuming that we have relations Friend and Relative as in Example 2.1.2, a Regular Query can describe the set of pairs of nodes linked by a path in which each node is connected to the next by both a Friend edge and a Relative edge.

$$FR(x, y) \leftarrow Friend(x, y), Relative(x, y)$$
$$Ans(x, y) \leftarrow FR^+(x, y)$$

We call *RQ Evaluation* the problem of query evaluation where the query must be a Regular Query.

Theorem 2.4.26 — Complexity of RQ Evaluation

The data complexity of RQ Evaluation is NLOGSPACE-complete and its combined complexity is NP-complete [57].

[57]: Reutter et al. (2017), "Regular Queries on Graph Databases"

The proof of Theorem 2.4.26 relies on a translation to a fragment of Datalog called "linear" for which the complexity bounds were already known.

[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"

Extended CRPQs What if instead of talking about pairs of nodes, as all prior languages do, we want to reason on the entire paths? This is the goal of Extended CRPQs. Introduced in [11], ECRPQs add to CRPQs the ability to compare path properties, such as length, prefix, edit distance and so on. From an automata theory point of view, this corresponds to regular relations, which are recognized by automata capable of reading multiple tapes at the same time and outputting letters.

Given a finite alphabet Σ, we denote by $\Sigma_\perp$ the alphabet Σ extended with the symbol $\perp$. For $\bar{s} = (s_1, \dots, s_n)$ an n-tuple of strings over Σ, we denote by $[\bar{s}]$ the string where the i-th symbol is a tuple $(c_i, \dots, c_n)$ where each c_i is the i-th symbol of s_k, if the length of s_k is at least i, or $\perp$ otherwise. A *regular relation* S over Σ^* is a relation over Σ^* such that the set $\{[\bar{s}] \mid \bar{s} \in S\}$ is either accepted by an automaton over $\Sigma_\perp$ or, equivalently, can be defined by a regular expression over $\Sigma_\perp$.

Definition 2.4.27 — ECRPQ

Given a finite alphabet Σ, an Extended Conjunctive Regular Path Query (ECRPQ), is an expression of the form:

$$Q(\bar{z}, \bar{\chi}) = \bigcap_{1 \le i \le m} (x_i, \pi_i, y_i), \bigcap_{1 \le j \le t} R_j(\bar{\omega}_j)$$

where

- $m > 0, t \ge 0$
- each R_j is a regular expression that defines a regular relation over Σ
- $\bar{x} = (x_1, \dots, x_m)$ and $\bar{y} = (y_1, \dots y_m)$ are tuples of node variables and $\bar{\pi} = (\pi_1, \dots, \pi_t)$ is a tuple of distinct path variables
- $\{\bar{\omega}_1, \dots, \bar{\omega}_t\}$ are distinct tuples of path variables from $\bar{\pi}$ such that each $\bar{\omega}_j$ is of the same arity as R_j
- $\bar{z}$ is a tuple of node variables form $\bar{x} \cup \bar{y}$ and $\bar{\chi}$ is a tuple of path variables from $\bar{\pi}$

The semantics of an ECRPQ is the set of pairs consisting of a mapping from $\bar{z}$ to nodes of G together with a mapping from $\bar{\chi}$ to paths in G. Formally, we say that a pair $(\sigma(\bar{z}), \mu(\bar{\chi}))$ belongs to $[\![Q]\!]_G$ if

- for each $1 \le i \le m$, $\mu(\pi_1)$ is a path in G from $\sigma(x_i)$ to $\sigma(y_i)$, and
- for each $\bar{\omega}_j = (\pi_{j_1}, \dots, \pi_{j_k})$ the tuple of strings of labels of $\mu(\pi_{j_1}), \dots, \mu(\pi_{j_k})$ belong to the relation R_j.

Example 2.4.28

The following ECRPQ Q returns two paths of nodes connected by Friend edges of the same length.

$$Q(\varnothing, \{\pi_1, \pi_2\}) \leftarrow (x_1, \pi_1, y_1) \wedge (x_2, \pi_2, y_2) \wedge \texttt{eqFriend}(\pi_1, \pi_2)$$

where the regular relation `eqFriend` is defined by the regular expression $(F, F)^+$.
When evaluated on the graph represented in Figure 2.3, the semantics of Q is the set of mappings from π_1, π_2 to paths in the graph (we ignore the mappings to nodes as the set $\bar{z}$ is empty in this example). This set consists of all the possible pairs in $\{\text{path}(p_1, f_2, p_2), \text{path}(p_2, f_3, p_4), \text{path}(p_2, f_4, p_3), \text{path}(p_4, f_1, p_1)\}$ for paths of length 1, in $\{(p_1, f_2, p_2, f_4, p_3), \text{path}(p_1, f_2, p_2, f_3, p_4), \text{path}(p_2, f_3, p_4, f_1, p_1), (p_4, f_1, p_1, f_2, p_2)\}$ for paths of length 2, etc.

We call *ECRPQ Evaluation* the problem of query evaluation where the query must be an ECRPQ and the input tuple is a pair of mappings from $\bar{\chi}$ to paths.

Theorem 2.4.29 — Complexity of ECRPQ Evaluation

The data complexity of ECRPQ Evaluation is NLOGSPACE and its combined complexity is PSPACE-complete [11].

[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"

Once again, the product automaton construction is used to prove the above result. To deal with potentially infinite paths in the answers, the set of paths can be returned as an automaton recognizing the language which describes these paths.

In [58], Diego Figueira and Varun Ramanathan refine the combined complexity bounds by showing that, depending on how the "reachability" and "path testing" parts of the query interact, the complexity of ECRPQ Evaluation can fall within PTime, NP or PSPACE.

[58]: Figueira et al. (2022), "When is the Evaluation of Extended CRPQ Tractable?"

As expected, the addition of regular relations is significant and yields a strictly more expressive class of languages.

Theorem 2.4.30 — Expressive power of ECRPQs

Extended CRPQs are strictly more expressive than CRPQs. In other words, there exists an ECRPQ query Q such that there is no CRPQ query Q' for which $[\![Q]\!]_G = [\![Q']\!]_G$ on all graphs G [11] [7].

[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"

In [59], Pablo Barcelo, Diego Figueira and Leonid Libkin, show that it is not possible to extend ECRPQs further to rational relations without losing decidability, as the complexity of finding the intersection of a regular language and a rational relation is only decidable for the subsequence relation, but not for subword or suffix.

[59]: Barceló et al. (2013), "Graph Logics with Rational Relations"

Queries with memory The last branch of extensions of RPQs we consider are Regular Data Path Queries which, unlike all previous languages, can express conditions on the data values stored in the properties. In this language, the regular language of RPQs is replaced by a register automaton or regular expression with memory. Unless stated

7: This notation assumes a slightly different presentation of CRPQ semantics than the one defined above. For full compatibility with ECRPQ semantics, CRPQ semantics can also be defined as a pair of mappings, the first from $\bar{z}$ to nodes in G, and the second as the empty mapping.

otherwise, all following definitions and results come from [60] and assume a graph model, called *data graph*, in which only edges are labelled. Without loss of generality [61], we also assume that properties appear only on nodes and that each node has exactly one property.

[60]: Libkin et al. (2016), "Querying Graphs with Data"

[61]: Vrgoc (2014), "Querying graphs with data"

Introduced by M. Kaminski and N. Francez in [62], Register Automata (originally "Finite Memory Automata") extend finite automata with read and write memory cells, called *registers*, which can take values from an infinite alphabet. They operate on a variant of "data words" which contain two letters at each position, one from a finite alphabet (of labels for example), the other from an infinite alphabet (of data values for example). These words have been studied by the automata theory community in contexts such as semi-structured data (specifically XML) [63], timed automata [64] and extensions of first order logic [65]. In the context of graph databases, the data words are replaced with data paths in which the two values appear one after the other (instead of together) and begin and end in a value from the infinite alphabet. Formally, a *data path* over a finite alphabet Σ and an infinite domain $\mathcal{V}$ is a finite alternating sequence from $(\Sigma \cdot \mathcal{V})^*$ starting and ending with elements of $\mathcal{V}$. Given a path p in a graph, the data path corresponding to p is obtained by replacing each node with its data value from $\mathcal{V}$ and each edge with its label from Σ.

[62]: Kaminski et al. (1994), "Finite-Memory Automata"

[63]: Bojanczyk (2010), "Automata for Data Words and Data Trees"

[64]: Bouyer et al. (2003), "An algebraic approach to data languages and timed languages"

[65]: Bojanczyk et al. (2011), "Two-variable logic on data words"

When moving from one state to the next, a register automaton performs three actions: first, it compares the current letter with the one specified by the transition (just like finite automaton), second, it optionally compares the current data value with the one stored in a particular register, and third, it optionally writes the current data value to a register (or multiple registers, the expressive power remains the same). In what follows, we separate step 1 from steps 2 and 3 into two transition functions to account for the data path structure.

The class of register automata can be further separated into sub-classes described by the number of registers that the automaton has access to. A k-register automaton is an automaton which uses at most k registers. For ease of notation, we assume that each register is assigned a numeric identifier, between 1 and k, and refer to the contents of register i as d_i.

The comparison *condition* for data values is a boolean combination of equalities between variables and other data values defined by the following grammar:

$$c \;:=\; x_i^{=} \;\mid\; x_i^{\neq} \;\mid\; z^{=} \;\mid\; z^{\neq} \;\mid\; c \wedge c \;\mid\; c \vee c \;\mid\; \neg c$$

where $x_i \in \{x_1, \ldots, x_k\}$ is a variable referring to the contents of the i-th register and z is a data value from $\mathcal{V}$. We denote by $\mathscr{C}_k$ the set of all conditions on the k registers.

We denote by $\mathcal{V}_\perp$ the extension of $\mathcal{V}$ with the symbol $\perp$ meaning that the register is empty. Given a data value $d \in \mathcal{V}$, a tuple representing the current register configuration $\tau = (d_1, \ldots, d_k) \in \mathcal{V}_\perp^k$ and a condition c, we say that d and τ satisfy c, denoted by $d, \tau \vDash c$, inductively as follows:

- If $c = x_i^{=}$ then $d, \tau \vDash c$ if $d = d_i$
- If $c = x_i^{\neq}$ then $d, \tau \vDash c$ if $d \neq d_i$
- If $c = z^{=}$ then $d, \tau \vDash c$ if $d = z$
- If $c = z^{\neq}$ then $d, \tau \vDash c$ if $d \neq z$
- If $c = c_1 \vee c_2$ then $d, \tau \vDash c$ if $d, \tau \vDash c_1$ or $d, \tau \vDash c_2$
- If $c = c_1 \wedge c_2$ then $d, \tau \vDash c$ if $d, \tau \vDash c_1$ and $d, \tau \vDash c_2$
- If $c = \neg c'$ then $d, \tau \vDash c$ if $d, \tau \nvDash c'$

Definition 2.4.31 — Register Automaton

A k-Register Automaton (RA) over Σ and $\mathcal{V}$ is a tuple $(Q, q_0, F, \delta, \tau_0)$ where

- $Q = Q_w \cup Q_d$ is a union of disjoint sets of word states and data states;
- $\delta : (\delta_w, \delta_d)$ is a pair of transition relations:
  - $\delta_w : Q_w \times \Sigma \times Q_d$ is the word transition relation
  - $\delta_d : Q_d \times \mathscr{C}_k \times 2^{\{1,..,k\}} \times Q_w$ is the data transition relation
- $q_0 \in Q_d$ is the initial state
- $F \subseteq Q$ is a set of accepting states
- $\tau_0 \in \mathcal{V}_\perp^k$ is the initial register configuration

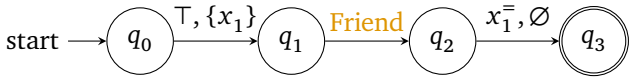
Let $p = v_0\ a_0\ \ldots\ v_{n-1}\ a_n$ be a data path over Σ and $\mathcal{V}$; where all v_is are from $\mathcal{V}$ and all a_is are from Σ. A *configuration* of $\mathscr{A}$ on p is a tuple (j, q, τ) where j is the current position of $\mathscr{A}$ in p, q is the current state and $\tau \in \mathcal{V}_\perp^k$ is the current content of the registers. The initial configuration is $(0, q_0, \tau_0)$ and any configuration with $q \in F$ is called final. The automaton can move from configuration $C = (j, q, \tau)$ to configuration $C' = (j+1, q', \tau')$ on p if one the following holds:

- $a_j \in \Sigma$, there is a transition $(q, a_j, q') \in \delta_w$ and $\tau' = \tau$
- $v_j \in \mathcal{V}$ and there is a transition $(q, c, I, q') \in \delta_d$ such that $v_j, \tau \models c$ and τ' is the same as τ except for the ith component, which is set to v_j whenever $i \in I$

A data path p is accepted by a register automaton $\mathscr{A}$ if there exists a sequence of configurations $C_0, \ldots, C_m$ such that C_0 is the initial configuration, C_m is a final configuration and, for all $1 \leq j \leq m$, the automaton can move from C_j to C_{j+1} by reading p. As for finite automata, we denote by $L(\mathscr{A})$ the set of data paths accepted by $\mathscr{A}$.

Example 2.4.32

The following is a 1-register automaton on $\Sigma = \{\text{Friend}, \text{Relative}\}$ and $\mathcal{V} = \mathbb{N}$.



This automaton accepts data paths of length 2, where the first letter is Friend and the two data values are equal.

By replacing the finite automaton in the definition of RPQs by a register automaton, we obtain the class of Regular Data Path Queries.

Definition 2.4.33 — Regular Data Path Query

Given a finite alphabet Σ and an infinite alphabet $\mathcal{V}$, a Regular Data Path Query (RDPQ) is an expression $Q = x \xrightarrow{\mathscr{A}} y$ where $\mathscr{A}$ is a register automaton over Σ and $\mathcal{V}$.

The semantics of an RDPQ evaluated on a graph G is the set of pairs of nodes in G such that there exists a data path labelled by a word recognized by $\mathscr{A}$ between them.

Just as finite automata have an equivalent declarative syntax, namely

regular expressions, the language of a register automaton can be described by a regular expression with memory, which extend regular expressions with a $\downarrow$ ("store into register") operator and conditions from $\mathscr{C}_k$.

Definition 2.4.34 — Regular Query with Memory

Given a finite alphabet Σ and a set of variables $x_1, \dots, x_k$, a *Regular Expression with Memory* (REM) is an expression obtained from the following grammar:

$$e \;:=\; \epsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow \bar{x}.e$$

where a is a letter in Σ, c is a condition in $\mathscr{C}_k$ and $\bar{x}$ is a tuple of variables from $x_1, \dots, x_k$.
A Regular Query with Memory (RQM) is an expression of the form $Q = x \xrightarrow{e} y$ where e is a regular expression with memory.

Given a REM e and a data path w, we say that w *satisfies* e if there exists two register configurations σ and σ' such that $(e, w, \sigma) \vdash \sigma'$ where the relation $\vdash$ is defined inductively as follows.

- $(\epsilon, w, \sigma) \vdash \sigma'$ iff $w = v$ for some $v \in \mathcal{V}$ and $\sigma' = \sigma$.
- $(a, w, \sigma) \vdash \sigma'$ iff $w = v_1 a v_2$ and $\sigma' = \sigma$
- $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ iff there is a splitting $w = w_1 \cdot w_2$ and a configuration σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$.
- $(e_1 + e_2, w, \sigma) \vdash \sigma'$ iff $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$.
- $(e^+, w, \sigma) \vdash \sigma'$ iff there is are a splitting $w = w_1 \cdots w_m$ and configurations $\sigma = \sigma_0, \cdots \sigma_m = \sigma'$ such that $(e_i, w_i, \sigma_{i-1}) \vdash \sigma_i$ for all $i \in \{1, \dots, m\}$.
- $(\downarrow \bar{x}.e, w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma_{\bar{x}=d}) \vdash \sigma'$ where $\sigma_{\bar{x}=d}$ is the configuration obtained from σ by setting all variables in $\bar{x}$ to the first value of w.
- $(e[c], w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma) \vdash \sigma'$ and $\sigma', d \vDash c$ where d is the last data value of w.

The semantics, or language, of a regular expression with memory e is the set of data paths that satisfy it, i.e. $\mathscr{L}(e) = \{w \mid \exists \sigma, (e, w, \bar{\perp}) \vdash \sigma\}$.

Example 2.4.35

The following RQM Q returns all pairs of nodes linked by a Friend edge and that have the same data value. Notice that the underlying REM is equivalent to the automaton of Example 2.4.32.

$$Q \;:=\; \downarrow x_1.(x) \xrightarrow{\text{Friend}} (y)[x_1^=]$$

When evaluated on the graph represented in Figure 2.3, the semantics of Q is the set $\{(p_1, p_2)\}$.

It is no surprise that REMs define the same class of languages as Register Automata.

Theorem 2.4.36 — Equivalence of RA and REMs

Register data path automata and regular expressions with memory define the same class of data path languages. [60]

[60]: Libkin et al. (2016), "Querying Graphs with Data"

The proof of this equivalence follows closely that of the equivalence of finite automata and regular expressions. For the automata to REM direction, the construction consists of merging the automaton's states while maintaining the transition information by putting increasingly more complex regular expressions with memory on the new transitions. In this way, the last remaining transition is labelled by the equivalent REM. As in the case of non-register automata, this can lead to an exponential blow-up. For the REM to automata direction, the automaton is constructed by induction on the form of the regular expression with memory, using standard techniques for the union, concatenation and Kleene star operations. Anytime the REM-specific operation $\downarrow$ is encountered, the current value is stored into a register and all conditions contained in $[c]$ operators are checked on the last transition of the sub-automaton.

Of course, this equivalence extends to the two query formalisms built on top of register automata and regular expressions with memory.

Corollary 2.4.37 — Equivalence of RDPQs and RQMs

For each regular data path query Q, there exists a regular query with memory Q' such that, for all data graphs D, $[\![Q]\!]_D = [\![Q']\!]_D$ and vice versa.

We call *RQM Evaluation* the problem of query evaluation where the query must be an RQM.

Theorem 2.4.38 — Complexity of RQM Evaluation

The data complexity of RQM Evaluation over data graphs is NLOGSPACE-complete and its combined complexity is PSPACE-complete [60].

[60]: Libkin et al. (2016), "Querying Graphs with Data"

As REMs can be efficiently translated into register automata, both upper bounds are obtained from the proof of the same statement for RDPQs. For data complexity, the algorithm is the same as for RPQs, i.e. guessing a word on the fly in the product automaton of the graph and the register automaton. For combined complexity, the algorithm remains the same with the extra need of space polynomial in the size of the query (as it is not fixed). For the lower-bounds, the NLOGSPACE-completeness comes directly from reachability on graphs, while the PSPACE-completeness is obtained by reduction to the problem of non-universality problem of finite automata, which asks, given an automaton $\mathscr{A}$, whether there exists a word that does not belong to $L(\mathscr{A})$.

[66]: Figueira et al. (2022), "Data Path Queries over Embedded Graph Databases"

The study of the impact of data values on pattern matching is further extended by Diego Figueira, Artur Jeż and Anthony W. Lin in [66] to operations from first-order infinite structures. Somewhat surprisingly, they show that for the real-closed field $\mathbb{R}_{\times,+}$ and the integer linear arithmetic $\mathbb{Z}_{LA}$ the data complexity of query evaluation remains in NLOGSPACE.

	RPQ	CRPQ/UCRPQ	2RPQ	C2RPQ/UC2RPQ
Data complexity	NLOGSPACE^C	NLOGSPACE^C	NLOGSPACE^C	NLOGSPACE^C
Combined complexity	PTime	NP^C	PTime	NP^C

	RQ	NRE	ECRPQ	RDPQ / RQM
Data complexity	NLOGSPACE^C	NLOGSPACE^C	NLOGSPACE^C	NLOGSPACE^C
Combined complexity	NP^C	PTime	PSPACE^C	PSPACE^C

Figure 2.6: Complexity of query evaluation for the languages presented in this section. As usual, ^{C} means the problem is complete for its class

Other extensions Many other navigational languages have been studied, some in the same context of graph databases, others for RDF or more generally semi-structured data. Already in 1987, Isabel F. Cruz, Alberto O. Mendelzon and Peter T. Wood introduced the language G, in which queries are essentially unions of RPQs with extra label variables [6]. The XML language XPath has been adapted to graphs by Leonid Libkin, Wim Martens and Domagoj Vrgoč in [67]. The problem of query evaluation for RPQs in the presence of a schema has been shown to be in EXPSPACE by Serge Abiteboul and Victor Vianu in [68]. The notion of time, for situations when data can become stale or invalid, and preference of a solution over another, have been investigated respectively by Marcelo Arenas, Pedro Bahamondes, Amir Aghasadeghi and Julia Stoyanovich in [69] and Gösta Grahne, Alex Thomo and William Wadge in [70].

[6]: Cruz et al. (1987), "A Graphical Query Language Supporting Recursion"

[67]: Libkin et al. (2013), "Querying graph databases with XPath"

[68]: Abiteboul et al. (1999), "Regular Path Queries with Constraints"

[69]: Arenas et al. (2022), "Temporal Regular Path Queries"

[70]: Grahne et al. (2007), "Preferentially Annotated Regular Path Queries"

2.5 Path semantics

All the query evaluation results of section 2.4 (except for ECRPQs) assume the *homomorphism semantics*, also called walk semantics, for paths, in which the results of queries are pairs of nodes and the path that connects them is forgotten. When the path component must be returned, the naïve approach is to compute the set of all paths that satisfy the query. This is called the *arbitrary path* semantics. Though it may seem like a desirable result, this semantic can lead to a difficult problem: infinite answers, makings tasks such as outputting the answer or counting the number of different paths near impossible. Indeed, if a cycle in the graph happens to match a pattern under Kleene star in the query, it can be traversed arbitrarily many times, creating a new answer each time. Nevertheless, figure 2.8 shows that 3 of the 10 most popular property graph database management systems do implement it and leave to the user the responsibility to limit the number of outputs in a reasonable way.

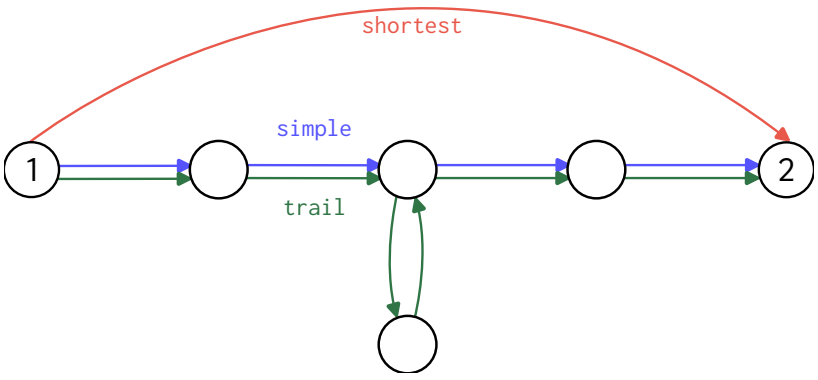


Figure 2.7: Illustration of the difference between shortest path, simple path and trail semantics for paths between nodes 1 and 2

Engine	Available path semantics	Score
Neo4j	Trail and shortest trail	44.46
ArangoDB	Shortest and arbitrary	3.32
Memgraph	Trail and shortest	3.02
Amazon Neptune	Trail	2.20
NebulaGraph	Arbitrary	2.14
TigerGraph	Shortest	1.83
DGraph	Shortest	1.45
AnzoGraph DB	Arbitrary [a]	0.23
AgensGraph	Shortest	0.20
Ultipa	Shortest	0.13

Figure 2.8: Available path semantics in Property Graph Database Management Systems ordered by DBEngines score. This score is a reflection of the popularity of a system, based on the number of results in search engines, frequency of searches on Google, number of related questions on technical websites, number of job offers, number of mentions in professional profiles and number of posts on X (formerly Twitter).

[a] only returns endpoints

The usual solution to output paths while avoiding the infinite answers problem is to consider a different semantic for the notion of path. The three main alternatives to arbitrary paths are: shortest path, simple path and trail. In the rest of this section we present these semantics and the main associated results for query evaluation and query enumeration.

Shortest paths

As the name implies, the *shortest path* semantics restricts paths to those that contain the least amount of edges among all the possible answers to the query. This definition has the advantage of being both simple to understand and to compute but it is often too restrictive and can discard relevant answers.

Given a graph G, an RPQ Q and two nodes u and v in G, we call single shortest path RPQ evaluation the problem of outputting a single shortest path between u and v that conforms to Q, and enumerate all shortest paths the problem of enumerating all such paths.

Theorem 2.5.1 — Complexity of shortest path evaluation

The single shortest path RPQ evaluation problem can be solved in linear time in the size of the graph and the query [71].

[71]: Vrgoč (2022), *Evaluating regular path queries under the all-shortest paths semantics*

The idea of the algorithm, presented in full in [71–73], is to run a shortest path finding algorithm, such as Breadth-First Search, on the product of the graph G and the automaton underlying the RPQ Q. As the shortest path finding algorithms run in time linear in the size of the considered graph, and the product graph can be constructed in linear size in the size of G and Q, we get the desired result.

[72]: Baier et al. (2017), "Evaluating Navigational RDF Queries over the Web"
[73]: Fionda et al. (2015), "NautiLOD: A Formal Language for the Web of Data Graph"

Theorem 2.5.2 — Complexity of shortest path enumeration

The enumerate all shortest paths problem can be solved with output in polynomial delay with a precomputation phase in linear time in the size of the graph and the query [71].

[71]: Vrgoč (2022), *Evaluating regular path queries under the all-shortest paths semantics*

This result comes from a simple modification of the algorithm of Theorem 2.5.1, extended to account for the need to remember which paths have already been output.

Simple paths

In the *simple path* semantics, a path must traverse each node at most once (usually this restriction is relaxed for the first and last nodes of the path, otherwise we talk about *acyclic* paths). This allows for paths longer than the shortest path but excludes paths that contain cycles. First introduced in the context of graph databases in [74], the simple path semantics has been extensively studied in the theoretical literature [75–81] but, as of May 2024, no graph database management system implements it (see figure 2.8) [8]. Unfortunately, this more flexible semantics comes with a complexity cost as the problem of query evaluation for simple paths is untractable.

Given a graph G, an RPQ Q and two nodes u and v in G, we call Regular Simple Path Query (RSPQ) the problem of RPQ Evaluation with the additional condition that the path must be simple.

Theorem 2.5.3 — Complexity of RSPQ

The Regular Simple Path Query problem is NP-complete for both data complexity and combined complexity [74].

The proofs of Theorem 2.5.3 rely on reductions from the *Even Path Problem*, which asks for a directed path of even length, and the *Disjoint Paths Problem*, which asks for a pair of directed paths (between two pairs of fixed nodes) that have no nodes in common. For the first reduction, it suffices to construct a graph G' identical to the input graph G except that all edges are labelled with the same letter, for example a. It is easy to see that finding a simple path labelled by $(aa)^*$ in G' is equivalent to finding a path of even length in G. For the second reduction, build G' the same way except for an additional edge labelled by a different letter, say b, between the destination endpoint of the first path and the source endpoint of the second path. With this construction, finding a single simple path between the first source endpoint and the second destination endpoint labelled by a^*ba^* is equivalent to finding two node-disjoint paths in G as the node-disjoint condition is preserved by the simple path semantics and the endpoints are preserved by the necessity to traverse the single b edge.

In [75], G. Bagan, A. Bonifati and B. Groz define the class of languages $\mathrm{SP}_{\mathrm{tract}}$ for which the Regular Simple Path Query problem is tractable, more specifically in NLOGSPACE. In [43], W. Martens and T. Trautner (now T. Popp), investigate the complexity of the Query Enumeration problem for simple paths constrained by RPQs. They show that this problem can be solved in polynomial delay whenever the language belongs to $\mathrm{SP}_{\mathrm{tract}}$.

Trails

The *trail* semantics is the dual of the simple path semantics for edges: a path is considered only if it traverses each edge at most once. This is the most lenient of the three semantics as it allows both paths longer than the shortest path and paths that go through a cycle once. Unlike for simple paths, the interest in trails from the academic community is very recent [43, 82–87], even though it is the most used in practice. The trail semantics is the default path semantics in Neo4j, which dominates the market of property graph database management systems by a very large margin, as witnessed by its db-engines score (see figure 2.8).

[74]: Mendelzon et al. (1989), "Finding Regular Simple Paths in Graph Databases"

[75]: Bagan et al. (2013), "A trichotomy for regular simple path queries on graphs"

[76]: Arenas et al. (2012), "Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard"

[77]: Losemann et al. (2013), "The complexity of regular expressions and property paths in SPARQL"

[78]: Barrett et al. (1998), "Formal Language Constrained Path Problems"

[79]: LaPaugh et al. (1984), "The even-path problem for graphs and digraphs"

[80]: Robertson et al. (1995), "Graph Minors .XIII. The Disjoint Paths Problem"

[81]: Nedev et al. (2000), "A Polynomial-Time Algorithm for Finding Regular Simple Paths in Outerplanar Graphs"

8: It is however very popular in RDF systems

[75]: Bagan et al. (2013), "A trichotomy for regular simple path queries on graphs"

[43]: Martens et al. (2018), "Evaluation and Enumeration Problems for Regular Path Queries"

[43]: Martens et al. (2018), "Evaluation and Enumeration Problems for Regular Path Queries"

[82]: Martens et al. (2019), "Bridging Theory and Practice with Query Log Analysis"

[83]: Martens et al. (2019), "Dichotomies for Evaluating Simple Regular Path Queries"

[84]: Martens et al. (2020), "A Trichotomy for Regular Trail Queries"

[85]: Martens et al. (2022), *Representing Paths in Graph Database Pattern Matching*

[86]: Martens et al. (2022), "The Complexity of Regular Trail and Simple Path Queries on Undirected Graphs"

[87]: Popp (2022), "Evaluation and Enumeration of Regular Simple Path and Trail Queries"

Unsurprisingly, like for simple paths, the problem of query evaluation for trails is untractable.

Given a graph G, an RPQ Q and two nodes u, v in G, we call Regular Trail Query (RTQ) the problem of RPQ Evaluation with the additional condition that the path must be a trail.

Theorem 2.5.4 — Complexity of RTQ

The Regular Trail Query problem is NP-complete in data complexity [87].

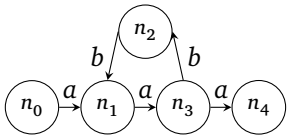
The proof is by reduction from the *Two-edge disjoint path* problem, which asks for a pair of directed paths (between two pairs of fixed nodes) that have no edges in common. The construction is the same as for the second reduction of the proof of Theorem 2.5.3. The edge-disjoint condition is then preserved by the trail semantics.

In [84], W. Martens, M. Niewerth and T. Trautner define T_{tract}, the subclass of languages for which the query evaluation problem is tractable (specifically NLOGSPACEcomplete) and the enumeration problem can be solved in polynomial delay.

[84]: Martens et al. (2020), "A Trichotomy for Regular Trail Queries"

Remark 2.5.5

Although trail and simple path semantics return more answers than shortest path for most queries, it is not always the case. For example, the semantics of $Q(x, y) = (x) \xrightarrow{a^+b^+a^+} (y)$ on the following graph G is empty for trail and simple path but not for shortest path, for which the path going through $n_0, n_1, n_3, n_2, n_1, n_3, n_4$ is an answer.

$G =$ 

In general, whenever $[\![Q]\!]_G \neq \emptyset$ for arbitrary path semantics then $[\![Q]\!]_G \neq \emptyset$ for shortest path semantics, which makes shortest path a reasonable candidate for both theoretical study and practical applications.

Other path semantics

Very few other path semantics have been studied in the theoretical literature. In 2006, Gösta Grahne and Alex Thomo proposed a Description Logic flavored semantics, called the "approximate semantics", in which labels are enhanced with real-world knowledge on related concepts. This requires less knowledge from the user on the internal representation and structure of the graph. Very recently C. David, N. Francis and V. Marsault have introduced the "Run-Based" semantics [88]. The core idea of this proposal is the separation of the regular expression into "atoms" which can be evaluated separately (under the simple path or trail semantics). This has the advantage of being less restrictive than all alternatives seen in this section, including trail, while remaining tractable, specifically NLOGSPACE-complete, for the query evaluation problem. At roughly the same time, Diego Figueira and Miguel Romero defined the "injective semantics" [89] which forces the assignment of

[88]: David et al. (2023), "Run-Based Semantics for RPQs"

[89]: Figueira et al. (2023), "Conjunctive Regular Path Queries under Injective Semantics"

variables to nodes in the graph to be injective. When this restriction is applied to the whole query, this semantics is equivalent to simple paths. When applied to "atoms" of the regular expression underlying the query, it allows the same node to be matched in different atoms of the query. Unlike the "Run-Based", the query evaluation problem for the "injective semantics" remains NP-complete in data complexity (trivially from the inclusion of simple path semantics).

References

[1] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685.

[4] Renzo Angles and Claudio Gutierrez. "Survey of graph database models". In: *ACM Comput. Surv.* 40.1 (2008), 1:1–1:39. DOI: 10.1145/1322432.1322433.

[6] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. "A Graphical Query Language Supporting Recursion". In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. Ed. by Umeshwar Dayal and Irving L. Traiger. ACM Press, 1987, pp. 323–330. DOI: 10.1145/38713.38749.

[9] Pablo Barceló. "Querying graph databases". In: *Principles of Database Systems (PODS)*. 2013, pp. 175–188.

[10] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. "Querying Graphs with Data". In: *J. ACM* 63.2 (2016), 14:1–14:53. DOI: 10.1145/2850413.

[11] Pablo Barceló et al. "Expressive languages for path queries over graph-structured data". In: *ACM Trans. Database Syst.* 37.4 (2012), 31:1–31:46.

[12] Peter T. Wood. "Query languages for graph databases". In: *SIGMOD Record* 41.1 (2012), pp. 50–60.

[13] Diego Figueira. "Containment of UC2RPQ: The Hard and Easy Cases". In: *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*. Ed. by Carsten Lutz and Jean Christoph Jung. Vol. 155. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 9:1–9:18. DOI: 10.4230/LIPICS.ICDT.2020.9.

[14] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. "Relative Expressiveness of Nested Regular Expressions". In: *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27-30, 2012*. 2012, pp. 180–195.

[19] Marcelo Arenas et al. *Database Theory*. Open source at https://github.com/pdm-book/community, 2022.

[20] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[21] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[22] Wilfrid Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[23] Michael Sipser. "Introduction to the Theory of Computation". In: *SIGACT News* 27.1 (1996), pp. 27–29. DOI: 10.1145/230514.571645.

[24] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[25] James Jones. "Undecidable diophantine equations". In: *Bulletin of the American Mathematical Society* 3.2 (1980), pp. 859–862.

[26] Ashok K. Chandra and Philip M. Merlin. "Optimal Implementation of Conjunctive Queries in Relational Data Bases". In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*. Ed. by John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison. ACM, 1977, pp. 77–90. DOI: 10.1145/800105.803397.

[27] Manuel Lima. *The book of trees : visualizing branches of knowledge*. New York : Princeton Architectural Press, 2014.

[28] openCypher. *Cypher Query Language Reference, Version 9*. 2017. URL: https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf.

[29] Alberto O. Mendelzon and Peter T. Wood. "Finding Regular Simple Paths in Graph Databases". In: *SIAM J. Comput.* 24.6 (1995), pp. 1235–1258.

[30] Béla Bollobás. *Modern Graph Theory*. Vol. 184. Springer Science & Business Media, 2013.

[31] Olaf Hartig et al. *RDF 1.2 Concepts and Abstract Syntax*. Mar. 2024. URL: https://www.w3.org/TR/2024/WD-rdf12-concepts-20240307/.

[32] Guus Schreiber and Yves Raimond. *RDF 1.1 Primer*. June 2014. URL: https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[33] Claudio Gutierrez, Carlos A. Hurtado, and Alberto O. Mendelzon. "Foundations of Semantic Web Databases". In: *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*. Ed. by Catriel Beeri and Alin Deutsch. ACM, 2004, pp. 95–106. DOI: 10.1145/1055558.1055573.

[34] Gavin Carothers and Eric Prud'hommeaux. *RDF 1.1 Turtle*. Feb. 2014. URL: https://www.w3.org/TR/2014/REC-turtle-20140225/.

[35] Dominik Tomaszuk and David Hyland-Wood. “RDF 1.1: Knowledge Representation and Data Integration Language for the Web”. In: *Symmetry* 12.1 (2020), p. 84. DOI: 10.3390/SYM12010084.

[36] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. “Reifying RDF: What Works Well With Wikidata?” In: *SSWS@ISWC*. 2015.

[37] Olaf Hartig. “RDF* and SPARQL*: An Alternative Approach to Annotate Statements in RDF”. In: *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 23rd - to - 25th, 2017*. Ed. by Nadeschda Nikitina et al. Vol. 1963. CEUR Workshop Proceedings. CEUR-WS.org, 2017.

[38] Renzo Angles et al. *PG-Schema: Schemas for Property Graphs*. 2022.

[39] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. “RDF and Property Graphs Interoperability: Status and Issues”. In: *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019*. Ed. by Aidan Hogan and Tova Milo. Vol. 2369. CEUR Workshop Proceedings. CEUR-WS.org, 2019.

[40] Serge Abiteboul and Victor Vianu. “Regular Path Queries with Constraints”. In: *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA*. Ed. by Alberto O. Mendelzon and Z. Meral Özsoyoglu. ACM Press, 1997, pp. 122–133. DOI: 10.1145/263661.263676.

[41] Erkki Mäkinen. “On Lexicographic Enumeration of Regular and Context-Free Languages”. In: *Acta Cybern.* 13.1 (1997), pp. 55–61.

[42] Margareta Ackerman and Jeffrey O. Shallit. “Efficient enumeration of words in regular languages”. In: *Theor. Comput. Sci.* 410.37 (2009), pp. 3461–3470. DOI: 10.1016/J.TCS.2009.03.018.

[43] Wim Martens and Tina Trautner. “Evaluation and Enumeration Problems for Regular Path Queries”. In: *International Conference on Database Theory (ICDT)*. Vol. 98. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 19:1–19:21.

[44] Katrin Casel and Markus L. Schmid. “Fine-Grained Complexity of Regular Path Queries”. In: *Log. Methods Comput. Sci.* 19.4 (2023). DOI: 10.46298/LMCS-19(4:15)2023.

[45] Dario Colazzo and Carlo Sartiani. “Typing regular path query languages for data graphs”. In: *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by James Cheney and Thomas Neumann. ACM, 2015, pp. 69–78. DOI: 10.1145/2815072.2815082.

[46] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. “Regular Path Query Evaluation on Streaming Graphs”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier et al. ACM, 2020, pp. 1415–1430. DOI: 10.1145/3318464.3389733.

[47] Diego Calvanese et al. “Reasoning on regular path queries”. In: *SIGMOD Rec.* 32.4 (2003), pp. 83–92. DOI: 10.1145/959060.959076.

[48] Diego Calvanese et al. “Answering Regular Path Queries Using Views”. In: *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. Ed. by David B. Lomet and Gerhard Weikum. IEEE Computer Society, 2000, pp. 389–398. DOI: 10.1109/ICDE.2000.839439.

[49] Miguel Romero, Pablo Barceló, and Moshe Y. Vardi. “The homomorphism problem for regular graph patterns”. In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017, pp. 1–12. DOI: 10.1109/LICS.2017.8005106.

[50] Diego Figueira and Rémi Morvan. “Approximation and Semantic Tree-Width of Conjunctive Regular Path Queries”. In: *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*. Ed. by Floris Geerts and Brecht Vandevoort. Vol. 255. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 15:1–15:19. DOI: 10.4230/LIPICS.ICDT.2023.15.

[51] Pablo Barceló, Miguel Romero, and Moshe Y. Vardi. “Semantic Acyclicity on Graph Databases”. In: *SIAM J. Comput.* 45.4 (2016), pp. 1339–1376. DOI: 10.1137/15M1034714.

[52] Pablo Barceló, Diego Figueira, and Miguel Romero. “Boundedness of Conjunctive Regular Path Queries”. In: *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*. Ed. by Christel Baier et al. Vol. 132. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 104:1–104:15. DOI: 10.4230/LIPICS.ICALP.2019.104.

[53] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. “An Extension of SPARQL for RDFS”. In: *Semantic Web, Ontologies and Databases, VLDB Workshop, SWDB-ODBIS 2007, Vienna, Austria, September 24, 2007, Revised Selected Papers*. Ed. by Vassilis Christophides, Martine Collard, and Claudio Gutier-

rez. Vol. 5005. Lecture Notes in Computer Science. Springer, 2007, pp. 1–20. DOI: 10.1007/978-3-540-70960-2_1.

[54] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "nSPARQL: A navigational language for RDF". In: *J. Web Semant.* 8.4 (2010), pp. 255–270. DOI: 10.1016/J.WEBSEM.2010.01.002.

[55] Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. "How to Best Nest Regular Path Queries". In: *Informal Proceedings of the 27th International Workshop on Description Logics, Vienna, Austria, July 17-20, 2014*. Ed. by Meghyn Bienvenu et al. Vol. 1193. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 404–415.

[56] Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. "Reasonable Highly Expressive Query Languages - IJCAI-15 Distinguished Paper (Honorary Mention)". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Qiang Yang and Michael J. Wooldridge. AAAI Press, 2015, pp. 2826–2832.

[57] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. "Regular Queries on Graph Databases". In: *Theory Comput. Syst.* 61.1 (2017), pp. 31–83. DOI: 10.1007/s00224-016-9676-2.

[58] Diego Figueira and Varun Ramanathan. "When is the Evaluation of Extended CRPQ Tractable?" In: *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid Libkin and Pablo Barceló. ACM, 2022, pp. 203–212. DOI: 10.1145/3517804.3524167.

[59] Pablo Barceló, Diego Figueira, and Leonid Libkin. "Graph Logics with Rational Relations". In: *Log. Methods Comput. Sci.* 9.3 (2013). DOI: 10.2168/LMCS-9(3:1)2013.

[60] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. "Querying Graphs with Data". In: *Journal of the ACM* 63.2 (2016), 14:1–14:53.

[61] Domagoj Vrgoc. "Querying graphs with data". PhD thesis. University of Edinburgh, UK, 2014.

[62] Michael Kaminski and Nissim Francez. "Finite-Memory Automata". In: *Theor. Comput. Sci.* 134.2 (1994), pp. 329–363. DOI: 10.1016/0304-3975(94)90242-9.

[63] Mikolaj Bojanczyk. "Automata for Data Words and Data Trees". In: *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*. Ed. by Christopher Lynch. Vol. 6. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 1–4. DOI: 10.4230/LIPICS.RTA.2010.1.

[64] Patricia Bouyer, Antoine Petit, and Denis Thérien. "An algebraic approach to data languages and timed languages". In: *Inf. Comput.* 182.2 (2003), pp. 137–162. DOI: 10.1016/S0890-5401(03)00038-5.

[65] Mikolaj Bojanczyk et al. "Two-variable logic on data words". In: *ACM Trans. Comput. Log.* 12.4 (2011), 27:1–27:26. DOI: 10.1145/1970398.1970403.

[66] Diego Figueira, Artur Jez, and Anthony W. Lin. "Data Path Queries over Embedded Graph Databases". In: *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid Libkin and Pablo Barceló. ACM, 2022, pp. 189–201. DOI: 10.1145/3517804.3524159.

[67] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. "Querying graph databases with XPath". In: *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. Ed. by Wang-Chiew Tan et al. ACM, 2013, pp. 129–140. DOI: 10.1145/2448496.2448513.

[68] Serge Abiteboul and Victor Vianu. "Regular Path Queries with Constraints". In: *J. Comput. Syst. Sci.* 58.3 (1999), pp. 428–452. DOI: 10.1006/JCSS.1999.1627.

[69] Marcelo Arenas et al. "Temporal Regular Path Queries". In: *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 2412–2425. DOI: 10.1109/ICDE53745.2022.00226.

[70] Gösta Grahne, Alex Thomo, and William W. Wadge. "Preferentially Annotated Regular Path Queries". In: *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings*. Ed. by Thomas Schwentick and Dan Suciu. Vol. 4353. Lecture Notes in Computer Science. Springer, 2007, pp. 314–328. DOI: 10.1007/11965893_22.

[71] Domagoj Vrgoč. *Evaluating regular path queries under the all-shortest paths semantics*. 2022.

[72] Jorge A. Baier et al. "Evaluating Navigational RDF Queries over the Web". In: *HT*. ACM, 2017, pp. 165–174. DOI: 10.1145/3078714.3078731.

[73] Valeria Fionda, Giuseppe Pirrò, and Claudio Gutierrez. "NautiLOD: A Formal Language for the Web of Data Graph". In: *ACM Trans. Web* 9.1 (2015), 5:1–5:43. DOI: 10.1145/2697393.

[74] Alberto O. Mendelzon and Peter T. Wood. “Finding Regular Simple Paths in Graph Databases”. In: *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*. 1989, pp. 185–193.

[75] Guillaume Bagan, Angela Bonifati, and Benoît Groz. “A trichotomy for regular simple path queries on graphs”. In: *Symposium on Principles of Database Systems (PODS)*. Ed. by Richard Hull and Wenfei Fan. ACM, 2013, pp. 261–272.

[76] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. “Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard”. In: *World Wide Web (WWW)*. 2012, pp. 629–638.

[77] Katja Losemann and Wim Martens. “The complexity of regular expressions and property paths in SPARQL”. In: *ACM Trans. Database Syst.* 38.4 (2013), p. 24.

[78] Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. “Formal Language Constrained Path Problems”. In: *Algorithm Theory - SWAT ’98, 6th Scandinavian Workshop on Algorithm Theory, Stockholm, Sweden, July, 8-10, 1998, Proceedings*. Ed. by Stefan Arnborg and Lars Ivansson. Vol. 1432. Lecture Notes in Computer Science. Springer, 1998, pp. 234–245. DOI: 10.1007/BFB0054371.

[79] Andrea S. LaPaugh and Christos H. Papadimitriou. “The even-path problem for graphs and digraphs”. In: *Networks* 14.4 (1984), pp. 507–513. DOI: 10.1002/NET.3230140403.

[80] Neil Robertson and Paul D. Seymour. “Graph Minors .XIII. The Disjoint Paths Problem”. In: *J. Comb. Theory, Ser. B* 63.1 (1995), pp. 65–110. DOI: 10.1006/JCTB.1995.1006.

[81] Zhivko Prodanov Nedev and Peter T. Wood. “A Polynomial-Time Algorithm for Finding Regular Simple Paths in Outerplanar Graphs”. In: *J. Algorithms* 35.2 (2000), pp. 235–259. DOI: 10.1006/JAGM.1999.1072.

[82] Wim Martens and Tina Trautner. “Bridging Theory and Practice with Query Log Analysis”. In: *SIGMOD Rec.* 48.1 (2019), pp. 6–13. DOI: 10.1145/3371316.3371319.

[83] Wim Martens and Tina Trautner. “Dichotomies for Evaluating Simple Regular Path Queries”. In: *ACM Trans. Database Syst.* 44.4 (2019), 16:1–16:46. DOI: 10.1145/3331446.

[84] Wim Martens, Matthias Niewerth, and Tina Trautner. “A Trichotomy for Regular Trail Queries”. In: *International Symposium on Theoretical Aspects of Computer Science, (STACS)*. 2020, 7:1–7:16.

[85] Wim Martens et al. *Representing Paths in Graph Database Pattern Matching*. 2022.

[86] Wim Martens and Tina Popp. “The Complexity of Regular Trail and Simple Path Queries on Undirected Graphs”. In: *PODS ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid Libkin and Pablo Barceló. ACM, 2022, pp. 165–174. DOI: 10.1145/3517804.3524149.

[87] Tina Popp. “Evaluation and Enumeration of Regular Simple Path and Trail Queries”. PhD thesis. University of Bayreuth, Germany, 2022.

[88] Claire David, Nadime Francis, and Victor Marsault. “Run-Based Semantics for RPQs”. In: *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*. Ed. by Pierre Marquis, Tran Cao Son, and Gabriele Kern-Isberner. 2023, pp. 178–187. DOI: 10.24963/KR.2023/18.

[89] Diego Figueira and Miguel Romero. “Conjunctive Regular Path Queries under Injective Semantics”. In: *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*. Ed. by Floris Geerts, Hung Q. Ngo, and Stavros Sintos. ACM, 2023, pp. 231–240. DOI: 10.1145/3584372.3588664.

3 What is GQL?

This chapter is joint work with N. Francis, A. Gheebrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund and D. Vrgoč and was published under the title "A Researcher's Digest of GQL" at ICDT 2023. The content has been adjusted to reflect the changes since the paper was first written, namely the publication of both standards.

Graph databases' widespread use happened without them having their lingua franca, which is the role that SQL is playing for relational databases. The landscape of graph languages — at least at first sight — is very varied. Neo4j has its own language called Cypher [15], which is also implemented in other products, including SAP HANA and Amazon Neptune. Oracle introduced its language PGQL [90]; TigerGraph has GSQL [17], and several products use the non-declarative graph traversal language Gremlin [91]. However, upon a closer examination, one discovers that declarative languages are more like different dialects of the same language rather than different languages altogether. This led to a proposal to define a new unifying standard for a Graph Query Language (GQL) [92]. The proposal was given a go-ahead in 2019, and since then was taken up by the same committee that produces and maintains the SQL Standard. It is known as ISO/IEC JTC1 SC32 WG3 within the International Organization for Standardization, or ISO.

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

[90]: Rest et al. (2016), "PGQL: a property graph query language"

[17]: Deutsch et al. (2020), "Aggregation Support for Modern Graph Analytics in TigerGraph"

[91]: Rodriguez (2015), "The Gremlin graph traversal machine and language"

[92]: Wikipedia contributors (2020), *GQL Graph Query Language*

In fact, this committee has developed two projects in parallel:

- *SQL/PGQ*, a new Part 16 of the SQL Standard, that defines querying graphs specified as views over a relational schema, published in June 2023.
- *GQL*, a standalone language for querying property graphs, published in April 2024.

The language of the Standard is hardly of the kind that the research community is accustomed to. It consists of a grammar for the constructs, supplemented with syntax and semantic rules, the latter written in natural language describing an algorithm for computing the result of a particular operation (essentially a mix of prose and pseudocode). Such descriptions are long, far from formal definitions suitable for initiating research in the area, and often prone to misinterpretation. To researchers, such a text is therefore much like a 500+ page legal document, instead of a workable definition that helps them understand the essence of the language.

This motivates the goal of this chapter: *to distill, in a form accessible to the database research community, the principal elements of the forthcoming GQL Standard, and provide their formal semantics.*

The idea of finding calculi underlying programming languages and providing their formal semantics is mainstream in the programming languages field. Recently we saw it extended to database query languages, specifically to core fragments of SQL [93–95] and Cypher [15]. The present chapter follows this trend. It provides a significant simplification of the GQL Standard, which at the same time covers its key features, and yet is sufficiently simple to provide its formal semantics, thereby enabling its further study and opening up new avenues of research on graph query languages.

[93]: Chu et al. (2017), "HoTTSQL: proving query rewrites with univalent SQL semantics"

[94]: Guagliardo et al. (2017), "A Formal Semantics of SQL Queries, Its Validation, and Applications"

[95]: Benzaken et al. (2019), "A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra"

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

We do not follow GQL letter to letter to simplify some of the idiosyn-

crasies of a real-life language to better highlight its essential features. Queries presented here are close to the features of the language. They come with a formal grammar that is a fragment of GQL's grammar, and a formal semantics, that is suitable as a starting point of new research in graph query languages. This work focuses on *read-only GQL queries*, to which we will simply refer as GQL queries. That is, we do not yet consider data updates.

Previous Academic Work on GQL The two graph languages currently standardized — GQL and SQL/PGQ — share their *pattern matching* facilities, which constitute the key part of any graph language. These were described in [96], by a group that included members of ISO's Standard group, as well as members of LDBC's Formal Semantics Working Group (FSWG), whose goal was to analyze and formalize the design of the language. This is the next installment in the effort to distill PGQ and GQL standards for the research community.

[96]: Deutsch et al. (2022), "Graph Pattern Matching in GQL and SQL/PGQ"

Apart from this recent work on GQL, we note that academic foundations already influenced its design process. As seen in GQL's influence graph [97], the language draws inspiration from regular path queries [6, 29], STRUQL [98], GXPath [60], and regular queries [57].

[97]: (2023), *GQL Influence Graph*

[6]: Cruz et al. (1987), "A Graphical Query Language Supporting Recursion"

[29]: Mendelzon et al. (1995), "Finding Regular Simple Paths in Graph Databases"

[98]: Fernandez et al. (1997), "A Query Language for a Web-Site Management System"

[60]: Libkin et al. (2016), "Querying Graphs with Data"

[57]: Reutter et al. (2017), "Regular Queries on Graph Databases"

3.1 GQL by Example

In this section we give a high-level description of GQL queries and their evaluation. The graph database model used by GQL is simply a collection of one or more property graphs. As an illustration, Figure 3.1 is a graph database consisting of two property graphs: the Fraud graph has information about bank transactions that are to be investigated for fraud, and the Social graph has information about people's social activities such as membership in a yacht club. Notice that these two graphs have a non-empty intersection: the nodes for Jay and Mike belong to both graphs, but they are seen in a different way and therefore have different labels and properties. In Fraud, the nodes have label `Account` and properties `owner` and `isBlocked`, indicating the status of the account. In Social, these nodes have label `Person` and property `name`.

We start with a simple query that looks for large (over $1M) transfers into a blocked account, and reports owners of accounts involved in such transfers:

```
1 USE Fraud
2 MATCH (x) -[z:Transfer WHERE z.amount>1000000]->
3       (y WHERE y.isBlocked=true)
4 RETURN x.owner AS sender, y.owner AS recipient
```

The reader familiar with Cypher will parse this query easily; it roughly follows Cypher's ascii-art syntax for expressing patterns, and also permits checking conditions on properties inside patterns. Essentially, the pattern in lines 2 and 3 asks for nodes `x` and `y` that are connected with an edge `z` that is labeled with `Transfer`. Furthermore, the `amount` property of `z` should exceed one million and the `isBlocked` property of *y* should be `true`. Such patterns, called *path patterns* in GQL, are the main building block of GQL queries, and they roughly correspond to RPQs.

Note also that the query is preceded by a `USE` clause stating explicitly in which graph matches are sought. When evaluating a query, GQL keeps track of

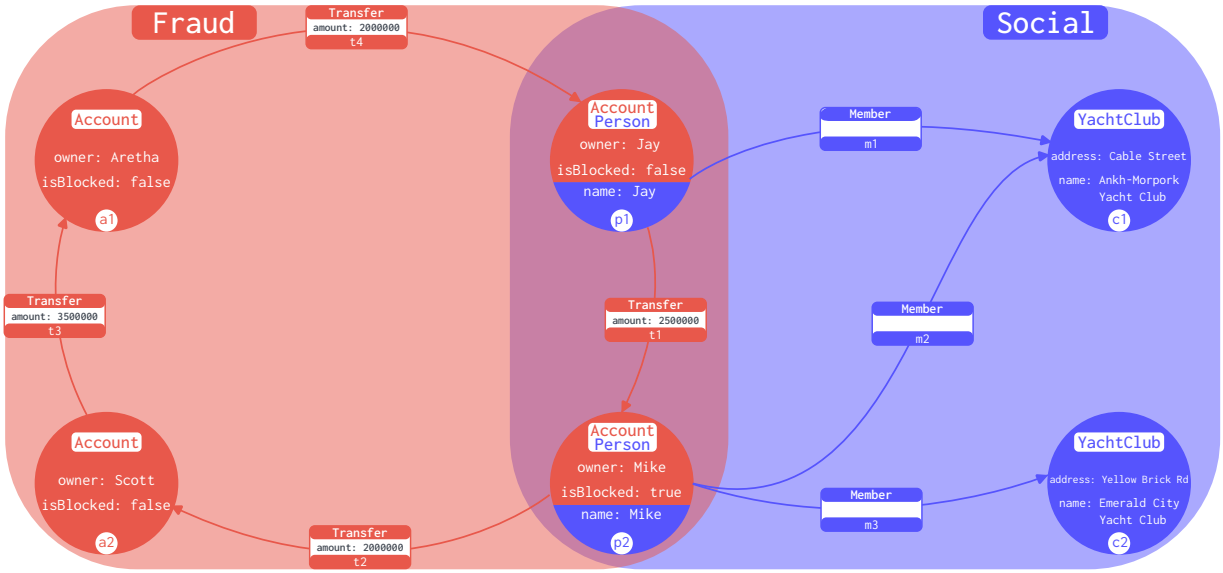


Figure 3.1: A database with graphs `Fraud` and `Social`.

- the *working graph*, which is the current graph in the database on which we do pattern matching and
- the *working table*, which contains intermediate results of the query, up to the current evaluation point.

Intuitively, the working table is a collection of records that gets passed from one part of the query to another in order to compute the final result. Thus, while GQL is a graph query language, it uses tables to represent intermediate and end-results of queries. In Section 3.3, we also discuss a third ingredient that GQL keeps track of, namely the working record.

Coming back to our sample query, in the first line we write `USE Fraud`, which turns the `Fraud` graph into our working graph. In lines 2 and 3, we have our path pattern, preceded by the keyword `MATCH`. This clause is the main workhorse of GQL, and it tells us to do the matching of the pattern onto the working graph. When evaluating our query over the database from Figure 3.1, after executing lines 2 and 3 of the query, we will be left with the following working table:

x	y	z
p1	p2	t1

Continuing in line 4, the working table is modified by keeping only the `owner` attribute of the nodes `x` and `y`, while renaming them, and the following is returned to the user:

sender	recipient
Jay	Mike

(3.1)

We next extend this query by checking for such transfers where both account owners are members of the same yacht club, reporting this time the address for the yacht club to send investigators to.

```
1 USE Fraud {
2     MATCH (x) -[z:Transfer WHERE z.amount>1000000]->
3           (y WHERE y.isBlocked=true)
4     RETURN x.owner AS sender, y.owner AS recipient
```

```
5       THEN
6       USE Social
7       MATCH (x1) -[:Member]-> (z1:YachtClub),
8           (y1) -[:Member]-> (z1:YachtClub)
9       FILTER sender=x1.name AND recipient=y1.name
10      RETURN z1.address AS clubAddress
11  }
```

Here lines 1–4 repeat the previous query. The keyword `THEN` is used to pipe the result of this query to the following subquery. While the curly braces extend the scope of `USE Fraud` beyond `THEN`, in line 6 we switch the working graph to `Social` in order to match the pattern:
`(x1)-[:Member]->(z1:YachtClub), (y1)-[:Member]->(z1:YachtClub)`

This pattern consists of two path patterns, separated by a comma. In GQL, the comma performs a join on the results of the two path patterns. From a theoretical point of view, it brings us in the realm of conjunctive (two-way) regular path queries. In GQL, such patterns are called *graph patterns*. When this pattern is evaluated over the `Social` graph, we obtain the following (fresh) working table:

x1	y1	z1
p1	p2	c1
p1	p1	c1
p2	p2	c1
p2	p2	c2

(3.2)

this time with variables `x1`, `y1`, and `z1`. After evaluating the pattern, the `MATCH` statement makes the natural join of table (3.2) with table (3.1), leading to

sender	recipient	x1	y1	z1
Jay	Mike	p1	p2	c1
Jay	Mike	p1	p1	c1
Jay	Mike	p2	p2	c1
Jay	Mike	p2	p2	c2

In this case, this will be the Cartesian product since the two working tables have no variables in common. The `FILTER` condition in line 9 selects only the first row of the latter table. The `RETURN` statement in line 10 tells us to keep only the `address` attribute of `z1`, renamed as `clubAddress`, resulting in:

clubAddress
Cable Street

This is also where our query ends, and the working table contains all the results to our query.

1: Notice that the query uses Cypher's ascii-art `( )` for nodes in the subexpressions `(x)`, `(y)`, and `()`, but also uses `( )` for indicating the subexpression over which `{1,}` is applied.

The examples we have seen thus far illustrate only a limited part of GQL since their variables only bind to single nodes or edges. Next, we show what happens to variables that can bind to *lists* and paths. Concerning lists, a query[1] such as

```
USE Fraud
MATCH TRAIL (x) ((y)-[:Transfer]->()){1,} (x)
RETURN x AS source, y AS moneyTrail
```

would return the following table.

source	moneyTrail
p1	list(p1, p2, a2, a1)
p2	list(p2, a2, a1, p1)
a2	list(a2, a1, p1, p2)
a1	list(a1, p1, p2, a2)

Here, the variable y is bound to a *list of nodes*. The four outputs all describe the same trail, which is the only Transfer-cycle in the graph, but the bindings use different start nodes for x and therefore also order the nodes in the lists for y differently. Concerning paths, the query

```
USE Fraud
MATCH TRAIL p = (x) (-[:Transfer]->()){1,} (x)
RETURN x AS source, p AS path
```

would return the following table.

source	path
p1	path(p1, t1, p2, t2, a2, t3, a1, t4, p1)
p2	path(p2, t2, a2, t3, a1, t4, p1, t1, p2)
a2	path(a2, t3, a1, t4, p1, t1, p2, t2, a2)
a1	path(a1, t4, p1, t1, p2, t2, a2, t3, a1)

The output is similar but this time we have the entire path instead of the list of nodes in each answer. We note that property graphs can have multiple edges with the same end-nodes, so the list of nodes in a path is not sufficient to determine the path.

3.2 Syntax of GQL

The full syntax of GQL queries is given in Figure 3.2 with $\mathbb{G}$ a set of property graphs, and the following pairwise disjoint countable sets: $\mathscr{L}$ of labels, $\mathscr{K}$ of keys, Const of value constants with a designated value null, and Vars of variables.

While somewhat intimidating at a first glance, the grammar can be roughly divided into four parts:

- *path patterns*, which mimic regular path queries [29, 99] (see Chapter 2 for an in-depth description), but have additional features such as two-way navigation and conditioning;
- *graph patterns*, which generalize conjunctive two-way regular path queries [100] with the ability to return different types of paths;
- *queries*, which allow us to manipulate the results of graph patterns and combine their evaluation over different graphs in the database; and
- *expressions and conditions*, which allow filtering results obtained in previous three parts of GQL.

[29]: Mendelzon et al. (1995), "Finding Regular Simple Paths in Graph Databases"
[99]: Mendelzon et al. (1996), "Querying the World Wide Web"
[100]: Calvanese et al. (2000), "Containment of Conjunctive Regular Path Queries with Inverse"

Of course, each of these parts has many specific features. For instance, path patterns allow using descriptors, which bind a node/edge to a variable, test its label or more complex conditions (e.g. amount is greater than 1000000). Simple node/edge patterns can be combined into regular expressions, by using concatenation, union or repetitions. Graph patterns, on the other hand, allow specifying the subset of matched paths that is to be returned, or joining path patterns into more complex queries. Finally, clauses/queries themselves allow us to manipulate results obtained from graph patterns, much like what is possible in the relational. Complex features such as iteration over the returned

elements, passing the results to another subquery, and changing the evaluation graph, are also supported.

Well-Formed Queries

The syntax of path patterns defined in Figure 3.2 is permissive as it allows expressions that do not type-check. For example, `(x)-[x]->()` is syntactically permitted even though it equates a node variable with an edge variable. Other patterns would provide great expressive power, such as the graph pattern `()-[y]->{0,}(), ()-[y]->{0,}*()`, which implicitly joins on lists.

We will introduce in Chapter 4 a type system operating on a subset of the patterns described in Figure 3.2. Its goal is to ensure that GQL path patterns and graph patterns do not exhibit the pathological behavior illustrated above. Here, we will only describe the resulting syntactic restrictions informally.

Each variable is given a *type* τ from the set $\mathbb{T}$ defined by the following grammar.

$$\tau ::= \mathsf{Node} \mid \mathsf{Edge} \mid \mathsf{Path} \mid \mathsf{Maybe}(\tau) \mid \mathsf{Group}(\tau)$$

The three atomic types are used for variables returning nodes, edges, and paths, respectively. The type constructor Maybe is used for variables occurring on one side of a disjunction only, while Group is used for variables occurring under repetition, whose bindings are grouped together. As variables in pattern matching are never bound to data values, we do not need the usual types like integers or strings here.

Types are computed in a bottom-up fashion as follows. Variables appearing in node patterns (resp. in edge patterns, resp. as names of path patterns) are of type Node (resp. Edge, resp. Path). Variables appearing on one side of a disjunction with type τ but not the other are of type $\mathsf{Maybe}(\tau)$. Variables appearing under a repetition with type τ are of type $\mathsf{Group}(\tau)$ higher-up in the syntax tree of the expression. Consider the pattern `(-[x]-> | -[y]->){0,}`. The type of x is Edge in `-[x]->`, while it is Maybe(Edge) in `-[`x`]-> | -[`y`]->`, and Group(Maybe(Edge)) in `(-[x]-> | -[y]->){0,}`.

A variable x appearing in a path/graph pattern ξ is called:

- a *singleton variable* if its type is Node or Edge with respect to ξ
- a *conditional variable* if its type is $\mathsf{Maybe}(\tau)$ for some type τ;
- a *group variable* if its type is $\mathsf{Group}(\tau)$ for some type τ;
- a *path variable* if its type is Path.

PATH PATTERN For $x \in \mathrm{Vars}$, $\ell \in \mathscr{L}$, $0 \le n \le m \in \mathbb{N}$:

(*descriptor*)	$\delta := x\ :\ell\ \texttt{WHERE}\ \theta$	x, $:\ell$, and `WHERE` θ are optional
(*path pattern*)	$\pi := (\delta)$	(*node pattern*)
	$\mid -[\delta]- > \mid < -[\delta]- \mid \sim[\delta]\sim$	(*edge pattern*)
	$\mid \pi\,\pi$	(*concatenation*)
	$\mid \pi \mid \pi$	(*union*)
	$\mid \pi\ \texttt{WHERE}\ \theta$	(*conditioning*)
	$\mid \pi\{n,m\}$	(*bounded repetition*)
	$\mid \pi\{n,\}$	(*unbounded repetition*)

EXPRESSION and **CONDITION** For $x \in \mathrm{Vars}$, $\ell \in \mathscr{L}$, $a \in \mathscr{K}$, $c \in \mathrm{Const}$:

(*expression*)	$\chi := x \mid x.a \mid c$
(*condition*)	$\theta := \chi = \chi \mid \chi < \chi \mid \chi\ \texttt{IS NULL}$
	$\mid x : \ell \mid \texttt{EXISTS}\ \{\mathsf{Q}\}$
	$\mid \theta\ \texttt{OR}\ \theta \mid \theta\ \texttt{AND}\ \theta \mid \texttt{NOT}\ \theta$

GRAPH PATTERN For $x \in \mathrm{Vars}$:

(*path mode*)	$\mu := (\texttt{ALL} \mid \texttt{ANY})\ [\texttt{SHORTEST}]\ [\texttt{TRAIL} \mid \texttt{ACYCLIC}]$
(*graph pattern*)	$\Pi := \mu\ [x =]\ \pi \mid \Pi, \Pi$

CLAUSE and **QUERY** For $k \ge 0$, $\ell \ge 1$, and $x, y, x_1, \ldots, x_k \in \mathrm{Vars}$, and $G \in \mathbb{G}$:

(*clause*)	$\mathsf{C} := \texttt{MATCH}\ \Pi$
	$\mid \texttt{LET}\ x = \chi$
	$\mid \texttt{FOR}\ x\ \texttt{IN}\ y$
	$\mid \texttt{FILTER}\ \theta$
(*linear query*)	$\mathsf{L} := \texttt{USE}\ G\ \mathsf{L}$
	$\mid \mathsf{C}\ \mathsf{L}$
	$\mid \texttt{RETURN}\ \chi_1\ \texttt{AS}\ x_1, \ldots, \chi_k\ \texttt{AS}\ x_k$
(*query*)	$\mathsf{Q} := \mathsf{L}$
	$\mid \texttt{USE}\ G\ \{\mathsf{Q}_1\ \texttt{THEN}\ \mathsf{Q}_2\ \cdots\ \texttt{THEN}\ \mathsf{Q}_\ell\}$
	$\mid \mathsf{Q}\ \texttt{INTERSECT}\ \mathsf{Q} \mid \mathsf{Q}\ \texttt{UNION}\ \mathsf{Q} \mid \mathsf{Q}\ \texttt{EXCEPT}\ \mathsf{Q}$

Figure 3.2: Syntax of GQL

Here is a non-exhaustive list [2] of the conditions a pattern must meet in order for its semantics to be defined. A pattern ξ is *well-formed* if

1. Every variable appearing in a pattern ξ has one and only one type w.r.t. ξ.
2. In concatenation and join, variables appearing in both operands are singleton variables with respect to each operand.
3. In a conditioned path pattern π `WHERE` θ, every variable appearing in θ must have a type w.r.t. π.
4. In a graph pattern of the form $\mu\ \pi$ or $\mu\ x\ =\ \pi$ such that μ is `ALL` (which is possible since all of `SHORTEST`, `TRAIL`, and `ACYCLIC` are optional), π must contain no unbounded repetition, to avoid potentially infinite outputs.
5. For every repeated pattern $\pi\{n,m\}$ or $\pi\{n,\}$, the *minimum path length* $\|\pi\|_{\min}$ of π, defined below, is positive. This avoids applying repetitions to paths that do not match an edge.

$$
\begin{aligned}
&\|\nu\|_{\min} = 0\\
&\|\eta\|_{\min} = 1\\
&\|\pi\{n,\}\|_{\min} = \|\pi\{n,m\}\|_{\min} = n \cdot \|\pi\|_{\min}\\
&\|\pi \texttt{ WHERE } \theta\|_{\min} = \|\pi\|_{\min}\\
&\|\pi_1 \mid \pi_2\|_{\min} = \min(\|\pi_1\|_{\min}, \|\pi_2\|_{\min})\\
&\|\pi_1\ \pi_2\|_{\min} = \|\pi_1\|_{\min} + \|\pi_2\|_{\min}
\end{aligned}
$$

Note that the local nature of types is important in item 2: implicit joins are allowed under repetitions, as in `( (a)-[]->(b)-[]->(a)-[]-> ){1,}`. Moreover, item 1 implies the existence of a schema, defined as follows:

Definition 3.2.1 Pattern schema

A *schema* of a well-formed pattern ξ is a function

$$\mathrm{sch}(\xi) : \mathrm{var}(\xi) \to \mathbb{T}$$

where $\mathrm{var}(\xi)$ is the set of variables appearing in ξ.

We will assume these syntactic restrictions to be in place when defining the semantics of GQL queries in Section 3.3. Moreover, we define the semantics only when the computation goes as expected, that is, when it satisfies preconditions we state explicitly. For instance, we will assume that a variable is bound before being used, that we never run into clashes in variable names, and that if a specific type is expected for an operation, then the value will have that type at runtime. Some of the preconditions could be checked syntactically, at the cost of a tedious type system. Some of the preconditions cannot be checked before run-time because they depend on the data stored in the database. Deciding how to treat those cases (static analysis, runtime exceptions, implicit casts) is outside the scope of this thesis. In some cases, the GQL standard describes how they should be treated, in others, they are implementation-dependent.

3.3 Semantics

In this section we present the formal semantics of GQL. At a high level, when evaluating a query, GQL keeps track of three things: (i) the working graph , which is the property graph we are using to match our patterns

2: For the full set of rules, please see the type system presented in Chapter 4 (from page 75).

currently; (ii) the *working table*, that stores the information computed thus far; and (iii) the *working record*, which contains the tuple of the result we are currently using. In this section we provide mathematical abstractions for each of these concepts in order to define the semantics of GQL. We start by setting the preliminary definitions, and then move to defining the semantics for each portion of the language, as specified in Figure 3.2.

3.3.1 Preliminaries

We use the standard definition of property graph and graph database introduced in Chapter 2 Section 2.3.

As GQL patterns allow variables to bind to multiple elements, we introduce a formal notion of lists.

Definition 3.3.1 — List

We use the notation $\mathrm{list}(v_1, \dots, v_n)$ to denote the list containing the objects $v_1, \dots, v_n$ in this order. Lists can be empty, in which case we write $\mathrm{list}()$. We use Lists to denote the set of all lists with elements in $\mathcal{N} \cup \mathcal{E}_\mathrm{d} \cup \mathcal{E}_\mathrm{u}$.

To define the formal semantics we use bindings which specify how variables are matched to values $\mathbb{V}$ of the input graph database. Intuitively, a binding is a mathematical formalization of the concept of a working record in GQL. Formally, we set $\mathbb{V}$ as the union $\mathrm{Const} \cup \mathcal{N} \cup \mathcal{E}_\mathrm{d} \cup \mathcal{E}_\mathrm{u} \cup \mathrm{Paths} \cup \mathrm{Lists}$.

Definition 3.3.2 — Binding

A binding μ is a partial function $\mu : \mathrm{Vars} \to \mathbb{V}$ whose domain $\mathrm{dom}(\mu)$ is finite. We denote bindings μ explicitly by $(x_1 \mapsto v_1, \dots, x_n \mapsto v_n)$ where $x_1, \dots, x_n$ are variables in $\mathrm{dom}(\mu)$, $v_1, \dots, v_n$ are values in $\mathbb{V}$, and for every i it holds that $\mu(x_i) = v_i$.

Note that the domains of bindings are not ordered, hence for instance $(a_1 \mapsto v_1, a_2 \mapsto v_2) = (a_2 \mapsto v_2, a_1 \mapsto v_1)$. The *empty binding*, that is, the binding with an empty domain, is denoted by $()$.

Definition 3.3.3 — Compatibility of bindings

Two bindings μ_1, μ_2 are said to be *compatible*, denoted by $\mu_1 \sim \mu_2$, if they agree on their shared variables, that is, for every $x \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$ it holds that $\mu_1(x) = \mu_2(x)$.

If μ_1 and μ_2 are compatible, we define their *join* $\mu_1 \bowtie \mu_2$ as expected, that is $\mathrm{dom}(\mu_1 \bowtie \mu_2) = \mathrm{dom}(\mu_1) \cup \mathrm{dom}(\mu_2)$ and $(\mu_1 \bowtie \mu_2)(x) = \mu_1(x)$ whenever $x \in \mathrm{dom}(\mu_1) \setminus \mathrm{dom}(\mu_2)$, and $(\mu_1 \bowtie \mu_2)(x) = \mu_2(x)$ whenever $x \in \mathrm{dom}(\mu_2)$.

We remark here that our definition allows joins on variables that are bound to paths or lists. However, as we will see, the syntactic restrictions on queries limit this feature significantly.

3.3.2 Semantics of Path Patterns

We start by defining the semantics of path patterns. For the remainder of this subsection, we consider a fixed property graph

$$G = \langle N^G, E_{\text{d}}^G, E_{\text{u}}^G, \text{lab}^G, \text{endpoints}^G, \text{src}^G, \text{tgt}^G, \text{prop}^G \rangle.$$

Moreover, we assume that all queries are well-formed and all patterns considered are restricted syntactically as described in Section 3.2. The semantics $[\![\pi]\!]_G$ of a pattern π is a set of pairs (p, μ) where μ a binding, and p is a path in G. In $[\![\pi]\!]_G$, G denotes the working graph in GQL parlance (specified by the keyword `USE`), and the pairs (p, μ) model what is computed over this working graph.

Semantics of Node and Edge Patterns

$$[\![(\,)]\!]_G = \left\{ (n, ()) \;\middle|\; n \in N^G \right\} \quad [\![(x)]\!]_G = \left\{ (n, (x \mapsto n)) \;\middle|\; n \in N^G \right\}$$
$$[\![(:\ell)]\!]_G = \left\{ (n, ()) \;\middle|\; n \in N^G, \ell \in \text{lab}^G(n) \right\}$$

Other cases are treated by moving the label and conditions outside of the node pattern, i.e `(`x`:`ℓ `WHERE` θ`)` becomes `(`x`)` `WHERE` `(`x`:`ℓ `AND` θ`)`.

$$[\![\texttt{-[]->}]\!]_G = \left\{ (\text{path}(\text{src}^G(e), e, \text{tgt}^G(e)), ()) \;\middle|\; e \in {E_d}^G \right\}$$
$$[\![\texttt{-[}x\texttt{]->}]\!]_G = \left\{ (\text{path}(\text{src}^G(e), e, \text{tgt}^G(e)), (x \mapsto e)) \;\middle|\; e \in {E_d}^G \right\}$$
$$[\![\texttt{-[:}\ell\texttt{]->}]\!]_G = \left\{ (\text{path}(\text{src}^G(e), e, \text{tgt}^G(e)), ()) \;\middle|\; e \in {E_d}^G, \ell \in \text{lab}^G(e) \right\}$$

Other cases of the forward edge patterns are treated by moving the label and conditions outside of the edge pattern, just as for node patterns. Backward edge patterns and undirected edge patterns are treated similarly, with the base cases given below.

$$[\![\texttt{<-[]-}]\!]_G = \left\{ (\text{path}(\text{tgt}^G(e), e, \text{src}^G(e)), ()) \;\middle|\; e \in {E_d}^G \right\}$$
$$[\![\texttt{~[]~}]\!]_G = \left\{ \begin{array}{l} (\text{path}(u_1, e, u_2), ()), \\ (\text{path}(u_2, e, u_1), ()) \end{array} \;\middle|\; \begin{array}{l} e \in {E_u}^G \\ \{u_1, u_2\} = \text{endpoints}^G(e) \end{array} \right\}$$

Semantics of Concatenation, Union, and Conditioning

$$[\![\pi_1\, \pi_2]\!]_G = \left\{ (p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \;\middle|\; \begin{array}{l} (p_i, \mu_i) \in [\![\pi_i]\!]_G \text{ for } i = 1, 2 \\ p_1 \text{ and } p_2 \text{ concatenate} \\ \mu_1 \sim \mu_2 \end{array} \right\}$$

Note that since $\pi_1\, \pi_2$ is assumed to be well-formed, all variables shared by π_1 and π_2 are singleton variables (Condition 2 in Section 3.2). In other words, implicit joins over group and optional variables are disallowed; the same remark will also apply for the semantics of joins.

Remark 3.3.4

Consider the pattern
`(x) (-[:Transfer]->()-[:Transfer]->(x)){1,}`
This pattern is disallowed in GQL because the leftmost x is a singleton variable, whereas the rightmost x is a group variable. In GQL philosophy, the leftmost x will be bound to a node and the rightmost x will be bound to a list of nodes, which is a type mismatch.

$$[\![\pi_1 \mid \pi_2]\!]_G = \left\{ (p, \mu \cup \mu') \;\middle|\; (p,\mu) \in [\![\pi_1]\!]_G \cup [\![\pi_2]\!]_G \right\}$$

where μ' maps every variable in $\mathrm{var}(\pi_1 \mid \pi_2) \setminus \mathrm{dom}(\mu)$ to null. (Recall that var maps a pattern to the set of variables appearing in it.)

$$[\![\pi \texttt{ WHERE } \theta]\!]_G = \left\{ (p,\mu) \in [\![\pi]\!]_G \;\middle|\; [\![\theta]\!]^{\mu}_G = \mathrm{true} \right\}$$

Semantics of Repetition

$$[\![\pi\{n,m\}]\!]_G = \bigcup_{i=n}^{m} [\![\pi]\!]^i_G$$
$$[\![\pi\{n,\}]\!]_G = \bigcup_{i=n}^{\infty} [\![\pi]\!]^i_G$$

Above, for a pattern π and a natural number $i \geq 0$, we use $[\![\pi]\!]^i_G$ to denote the i-th power of $[\![\pi]\!]_G$, which we define as

$$[\![\pi]\!]^0_G = \{\, (\mathrm{path}(u), \mu) \mid u \text{ is a node in } G \,\}$$

where μ binds each variable in $\mathrm{dom}(\mathrm{sch}(\pi))$ to list(), i.e. the empty-list value; and

$$\forall i > 0 \quad [\![\pi]\!]^i_G = \left\{ (p_1 \cdot \ldots \cdot p_i, \mu') \;\middle|\; \begin{array}{l} (p_1,\mu_1), \ldots, (p_n,\mu_i) \in [\![\pi]\!]_G \\ p_1, \ldots, p_i \text{ concatenate} \end{array} \right\}$$

where μ' binds each variable in $\mathrm{dom}(\mathrm{sch}(\pi))$ to $\mathrm{list}(\mu_1(x), \ldots, \mu_i(x))$. Recall that sch is defined in Section 3.2.

Remark 3.3.5

Since $\pi\{n,\}$ is assumed to be well-formed, it holds $\|\pi\|_{\min} \geq 1$. A simple induction then yields that each p_i in the definition above has positive length. A second induction then yields that, given a path p, there are finitely many bindings μ such that $(p,\mu) \in [\![\pi\{n,m\}]\!]_G$. This fact is crucial to have a finite output in the end.

For instance, consider a graph with a single node u and no edges, and the pattern `(a){0,}` which is not well-formed (the minimal path length of `()` is 0). For every i, the set $[\![\texttt{(a)}]\!]^i_G$ contains $(\mathrm{path}(u), \mu_i)$ where $\mu_i = (a \mapsto \mathrm{list}(\underbrace{u, \ldots, u}_{i \text{ times}}))$; hence the union in the definition of $[\![\pi\{n,\}]\!]_G$ above would not only yield an infinite number of elements, but all of

them would be associated to the same path. As a result a graph pattern such as `ALL SHORTEST (a){0,}` would have infinitely many results.

3.3.3 Semantics of Graph Patterns

We now define the semantics of graph patterns. We first fully define atomic graph patterns and then define their joins.

$$[\![x = \pi]\!]_G = \{(p, \mu \cup \{x \mapsto p\}) \mid (p, \mu) \in [\![\pi]\!]_G\}$$

In the following we denote by $\tilde{\pi}$ a graph pattern that never uses the "," operator, hence it is of the form $\mu\ x{=}\ \pi$, where μ is a path mode, x is a variable, π is a path pattern, and "x=" is optional.

$$[\![\texttt{TRAIL}\ \pi]\!]_G = \{\ (p,\mu) \in [\![\pi]\!]_G \mid \text{no edge occurs more than once in } p\ \}$$
$$[\![\texttt{ACYCLIC}\ \pi]\!]_G = \{\ (p,\mu) \in [\![\pi]\!]_G \mid \text{no node occurs more than once in } p\ \}$$
$$[\![\texttt{SHORTEST}\ \tilde{\pi}]\!]_G = \left\{ (p,\mu) \in [\![\tilde{\pi}]\!]_G \;\middle|\; \mathrm{len}(p) = \min\left\{ \mathrm{len}(p') \;\middle|\; \begin{array}{l} (p',\mu') \in [\![\tilde{\pi}]\!]_G \\ \mathrm{src}(p') = \mathrm{src}(p) \\ \mathrm{tgt}(p') = \mathrm{tgt}(p) \end{array} \right\} \right\}$$
$$[\![\texttt{ALL}\ \tilde{\pi}]\!]_G = [\![\tilde{\pi}]\!]_G$$
$$[\![\texttt{ANY}\ \tilde{\pi}]\!]_G = \bigcup_{(s,t)\in X} \{\mathrm{any}(\{\ (p,\mu) \in [\![\tilde{\pi}]\!]_G \mid \mathrm{endpoints}(p) = (s,t)\ \})\}$$

3: any need not be deterministic.

where $X = \{\ (\mathrm{src}(p), \mathrm{tgt}(p)) \mid (p,\mu) \in [\![\tilde{\pi}]\!]_G\}$ and any is a procedure that arbitrarily returns one element from a set [3].

$$[\![\Pi_1, \Pi_2]\!] = \left\{ (\bar{p}_1 \times \bar{p}_2, \mu_1 \bowtie \mu_2) \;\middle|\; \begin{array}{l} (\bar{p}_1,\mu_1) \in [\![\Pi_1]\!]_G, (\bar{p}_1,\mu_2) \in [\![\Pi_2]\!]_G \\ \text{and } \mu_1 \sim \mu_2 \end{array} \right\}$$

Here, $\bar{p}_1 = (p_1^1, p_1^2, \ldots, p_1^k)$ and $\bar{p}_2 = (p_2^1, p_2^2, \ldots, p_2^l)$ are tuples of paths, and $\bar{p}_1 \times \bar{p}_2$ stands for $(p_1^1, p_1^2, \ldots, p_1^k, p_2^1, p_2^2, \ldots, p_2^l)$. Just as it is the case of concatenation, since $\Pi_1,\ \Pi_2$ is well-formed, implicit joins can occur over singleton variables only.

3.3.4 Semantics of Conditions and Expressions

The semantics $[\![\chi]\!]_G^\mu$ of an expression χ is an element in $\mathbb{V}$ that is computed with respect to a binding μ and a graph G. Intuitively, variables in χ are evaluated with μ and we use G to access the properties of an element. It is formally defined as follows.

$$[\![c]\!]_G^\mu = c$$
$$[\![x]\!]_G^\mu = \mu(x) \qquad \text{for}$$
$$[\![x.a]\!]_G^\mu = \begin{cases} \mathrm{prop}^G(\mu(x), a) & \text{if } (\mu(x), a) \in \mathrm{dom}(\mathrm{prop}^G) \\ \mathrm{null} & \text{else if } \mu(x) \in (\mathcal{N} \cup \mathcal{E}_\mathrm{d} \cup \mathcal{E}_\mathrm{u}) \end{cases}$$

where $c \in \mathrm{Const}$, $x \in \mathrm{dom}(\mu)$ and $a \in \mathcal{K}$.

Remark 3.3.6

Recall that different graphs may share nodes and edges. Hence the condition $(\mu(x), a) \in \mathrm{dom}(\mathrm{prop}^G)$, above, does imply that $\mu(x)$ is a node or an edge in G, but does **not** imply that it was matched in G.

The semantics $[\![\theta]\!]^\mu_G$ of a condition θ is an element in $\{\mathrm{true}, \mathrm{false}, \mathrm{null}\}$ that is evaluated with respect to a binding μ and a graph G, and is defined as follows:

$$[\![\chi_1 = \chi_2]\!]^\mu_G = \begin{cases} \mathrm{null} & \text{if } [\![\chi_1]\!]^\mu_G = \mathrm{null} \text{ or } [\![\chi_2]\!]^\mu_G = \mathrm{null} \\ \mathrm{true} & \text{if } [\![\chi_1]\!]^\mu_G = [\![\chi_2]\!]^\mu_G \neq \mathrm{null} \\ \mathrm{false} & \text{otherwise} \end{cases}$$

$$[\![\chi_1 < \chi_2]\!]^\mu_G = \begin{cases} \mathrm{null} & \text{if } [\![\chi_1]\!]^\mu_G = \mathrm{null} \text{ or } [\![\chi_2]\!]^\mu_G = \mathrm{null} \\ \mathrm{true} & \text{else if } [\![\chi_1]\!]^\mu_G < [\![\chi_2]\!]^\mu_G \\ \mathrm{false} & \text{otherwise} \end{cases}$$

$$[\![\chi \texttt{ IS NULL}]\!]^\mu_G = \begin{cases} \mathrm{true} & \text{if } [\![\chi]\!]^\mu_G = \mathrm{null} \\ \mathrm{false} & \text{otherwise} \end{cases}$$

$$[\![\chi : \ell]\!]^\mu_G = \begin{cases} \mathrm{true} & \text{if } [\![\chi]\!]^\mu_G \in N^G \cup {E_u}^G \cup {E_d}^G \text{ and } \ell \in \mathrm{lab}^G([\![\chi]\!]^\mu_G) \\ \mathrm{false} & \text{else if } [\![\chi]\!]^\mu_G \in \mathcal{N} \cup \mathcal{E}_\mathrm{d} \cup \mathcal{E}_\mathrm{u} \end{cases}$$

$$[\![\theta_1 \texttt{ AND } \theta_2]\!]^\mu_G = [\![\theta_1]\!]^\mu_G \wedge [\![\theta_2]\!]^\mu_G \quad (*)$$

$$[\![\theta_1 \texttt{ OR } \theta_2]\!]^\mu_G = [\![\theta_1]\!]^\mu_G \vee [\![\theta_2]\!]^\mu_G \quad (*)$$

$$[\![\texttt{NOT } \theta]\!]^\mu_G = \neg\, [\![\theta]\!]^\mu_G \quad (*)$$

$$[\![\texttt{EXISTS\{ Q \}}]\!]^\mu_G = \begin{cases} \mathrm{true} & \text{if } [\![\mathrm{Q}]\!]_G(\{\mu\}) \text{ is not empty} \\ \mathrm{false} & \text{otherwise} \end{cases}$$

(∗): Operators ∧, ∨, and ¬ are defined as in SQL three-valued logic, e.g. null ∨ true = true while null ∧ true = null.

3.3.5 Semantics of Queries

Clauses and queries are interpreted as functions that operate on tables. These tables are our abstraction of GQL's working tables.

Definition 3.3.7 — Table

A table T is a set of bindings that have the same domains, referred to as $\mathrm{dom}(T)$.

Note that tables do not have schemas: two different bindings in a table might associate a variable to values of incompatible types.

Semantics of Clauses The semantics $[\![\mathrm{C}]\!]_G$ of a clause C is a function that maps tables into tables, and is parametrized by a graph G. Patterns, conditions and expression in a clause are evaluated with respect to G.

$$[\![\texttt{MATCH } \Pi]\!]_G(T) = \bigcup_{\mu \in T} \{\mu \bowtie \mu' \mid (p, \mu') \in [\![\Pi]\!]_G,\ \mu \sim \mu'\}$$

Note that if Π uses a variable that already occurs in $\mathrm{dom}(T)$, a join is performed. Unlike in the case of path patterns and graph patterns, this join can involve variables bound to lists or paths. While this is not problematic mathematically, it is disallowed in GQL.

If $x \notin \mathrm{dom}(T)$, then

$$[\![\texttt{LET}\ x = \chi]\!]_G(T) = \bigcup_{\mu \in T} \left\{ \mu \bowtie \left(x \mapsto [\![\chi]\!]^{\mu}_G \right) \right\}$$
$$[\![\texttt{FILTER}\ \theta]\!]_G(T) = \bigcup_{\mu \in T} \{ \mu \mid [\![\theta]\!]^{\mu}_G = \mathrm{true} \} .$$

4: Note that null is treated just as list()

If $x \notin \mathrm{dom}(T)$ and, for every $\mu \in T$, $\mu(y)$ is a list or null,[4] then

$$[\![\texttt{FOR}\ x\ \texttt{IN}\ y]\!]_G(T) = \bigcup_{\mu \in T} \{ \mu \bowtie (x \mapsto v) \mid v \in \mu(y) \} .$$

Semantics of Linear Queries

$$[\![\texttt{USE}\ G'\ \mathsf{L}]\!]_G(T) = [\![\mathsf{L}]\!]_{G'}(T)$$
$$[\![C\ \mathsf{L}]\!]_G(T) = [\![\mathsf{L}]\!]_G([\![C]\!]_G(T))$$
$$[\![\texttt{RETURN}\ \chi_1\ \texttt{AS}\ x_1, \ldots, \chi_\ell\ \texttt{AS}\ x_\ell]\!]_G = \bigcup_{\mu \in T} \left\{ \begin{pmatrix} x_1 \mapsto [\![\chi_1]\!]^{\mu}_G, \ldots, \\ x_\ell \mapsto [\![\chi_\ell]\!]^{\mu}_G \end{pmatrix} \right\}$$

Semantics of Queries The *output* of a query Q is defined as

$$\mathrm{Output}(\mathsf{Q}) = [\![\mathsf{Q}]\!]_G(\{()\}) ,$$

where $\{()\}$ is the unit table that consists of the empty binding, and G is the default graph in D. We define the semantics of queries recursively as follows.

$$[\![\texttt{USE}\ G'\ \{\mathsf{Q}_1\ \texttt{THEN}\ \mathsf{Q}_2\ \cdots\ \texttt{THEN}\ \mathsf{Q}_k\}]\!]_G(T) = [\![\mathsf{Q}_k]\!]_{G'} \circ \cdots \circ [\![\mathsf{Q}_1]\!]_{G'}(T)$$

If $\mathrm{dom}\left([\![\mathsf{Q}_1]\!]_G(T)\right) = \mathrm{dom}\left([\![\mathsf{Q}_2]\!]_G(T)\right)$, then we let

$$[\![\mathsf{Q}_1\ \texttt{INTERSECT}\ \mathsf{Q}_2]\!]_G(T) = [\![\mathsf{Q}_1]\!]_G(T) \cap [\![\mathsf{Q}_2]\!]_G(T)$$
$$[\![\mathsf{Q}_1\ \texttt{UNION}\ \mathsf{Q}_2]\!]_G(T) = [\![\mathsf{Q}_1]\!]_G(T) \cup [\![\mathsf{Q}_2]\!]_G(T)$$
$$[\![\mathsf{Q}_1\ \texttt{EXCEPT}\ \mathsf{Q}_2]\!]_G(T) = [\![\mathsf{Q}_1]\!]_G(T) \setminus [\![\mathsf{Q}_2]\!]_G(T)$$

3.4 A Few Known Discrepancies with the GQL Standard

In pursuing the goal of introducing the key features of GQL to the research community, we inevitably had to make decisions that resulted in discrepancies between our presentation and the 500+ pages of the forthcoming Standard. In this section, we discuss a non-exhaustive list of differences between the actual GQL Standard and our digest. To start with, in all our formal development we assumed that queries are given by their syntax trees, which result from parsing them. Hence we completely omitted such parsing-related aspects as parentheses, operator precedence etc. Also we note that many GQL features, even those described here, are optional, and not every implementation is obliged to have them all.

The remaining discrepancies are divided into three main categories: syntactic restrictions (Section 3.4.1), query evaluation (Section 3.4.2), and missing features (Section 3.4.3).

3.4.1 User-Friendly Syntactic Restrictions

The GQL Standard imposes restrictions on the syntax that aim at preventing unexpected behavior, and that we generally did not describe. Two such examples are given below.

First, consider the two following queries

$$Q_1 = \texttt{MATCH}\ \mu\ \texttt{x=-[]->*}$$
$$Q_2 = \texttt{MATCH}\ \mu\ \texttt{x=-[]->*()}$$

for some path mode μ (it does not matter which one). According to our semantics, both return one binding, namely $(x \mapsto \mathtt{path}(u))$, for each node u in the graph; however, Q_1 is syntactically forbidden in the GQL Standard because no node pattern occurs.

Another interesting syntactic restriction concerns *strict interior* variables under path modes, such as c in the following:

```
MATCH ANY (:Person) -[]->* (c:Account) -[]->* (:Person),
      ANY (:Person) -[]->* (c:Account) -[]->* (:Person)
```

The `ANY` path modes are evaluated independently, and before the implicit join on variable c. Then, the node bound to the variable c by either path pattern is arbitrary, and joining on them is very likely to fail. This situation was not deemed user-friendly by the Committee, and therefore precluded.

3.4.2 Query Evaluation

Bag semantics

For simplicity, we described GQL as if it was following set semantics but, in reality, GQL uses bags just like Cypher and SQL. In order to define clauses and queries under bag semantics, small changes are needed:

- tables should be defined as bags, rather than sets, of bindings;
- unions ($\cup$) over the elements of a table should be additive bag unions ($\uplus$); and
- set comprehensions should be replaced with bag comprehensions.

As an example, if we denote bags with double curly braces, then the semantics of `RETURN` is

$$[\![\texttt{RETURN}\ \chi_1\ \texttt{AS}\ x_1, \ldots, \chi_\ell\ \texttt{AS}\ x_\ell]\!]_G(T) = \biguplus_{\mu \in T} \left\{\!\!\left\{ \begin{pmatrix} x_1 \mapsto [\![\chi_1]\!]^\mu_G, \ldots, \\ x_\ell \mapsto [\![\chi_\ell]\!]^\mu_G \end{pmatrix} \right\}\!\!\right\}$$

Note that GQL partially eliminates duplicates during pattern matching, which is reflected here by the semantics of graph patterns: $[\![\Pi]\!]_G$ is a *set* of path/binding pairs, while $[\![\texttt{MATCH}\ \Pi]\!]_G$ returns a *bag* of bindings by projecting out the paths (see the definition of $[\![\texttt{MATCH}\ \Pi]\!]_G$ in Section 3.3.5). Hence, different ways to compute the same path/binding pair will only contribute to one copy of the binding in the output of $[\![\texttt{MATCH}\ \Pi]\!]_G$. It is still possible to get multiple copies of some binding in the output, but these come from pairs with different paths.

Partial deduplication is an effort to unify the multiplicities of queries that express *the same* pattern in different ways. To see this, consider the queries

Q_1: `MATCH (a:Person)-[]->(b WHERE b:Person OR b:Account)`
Q_2: `MATCH (a:Person)-[]->(b:Person) | (a)-[]->(b:Account)`

and the path $\mathtt{path}(v_1, e_1, v_2)$ matched by either of them with the binding $\mu_1 = (a \to v_1, b \mapsto v_2)$, where v_2 bears *both* labels `Person` and `Account`. As the disjunction in Q_1 is expressed using a Boolean condition, this query always returns a single copy of μ_1. In Q_2, however, the disjunction is expressed with a union (`|`) of patterns; thus, if the semantics of `|` were defined as a bag-union, the query would return two copies of μ_1.

Finally, as in SQL, the operations **`INTERSECT`**, **`UNION`**, and **`EXCEPT`** remove duplicates in GQL, while the variants **`INTERSECT ALL`**, **`UNION ALL`**, and **`EXCEPT ALL`** do not.

Path bindings

[96]: Deutsch et al. (2022), "Graph Pattern Matching in GQL and SQL/PGQ"

In a nutshell, a *path binding* is a path where each element may be annotated with variables, and it is inconsistent as soon as two different elements have the same annotation (see [96] for details). Thus, a path binding defines a single path/binding pair, whereas a path/binding pair can define several path bindings. In GQL Standard, pattern matching computes a set of consistent path bindings, while our semantics computes a set of path/binding pairs, and the results are bags formed by projecting away paths. Consequently, our semantics might sometimes return fewer results than GQL's, but the difference only affects multiplicity. For example, consider **`MATCH`** `()-[]->(a) | (a)-[]->()` on a graph with a single node u and a single (looping) edge. According to our semantics, only one copy of $(a \mapsto u)$ is returned, while two occurrences of it are returned according to GQL Standard.

Postponed evaluation of conditions.

In our treatment of the language, the semantics of the following query is undefined:

```
MATCH ALL SHORTEST -[x]->
( ()-[y]->() WHERE x.amount < y.amount ){10,10}
```

5: This is orthogonal to left-to-right evaluation: `-[x]->` could be placed on the right instead.

Indeed, when the condition **`WHERE`** `x.amount < y.amount` is evaluated, the variable `x` is not yet bound, as `-[x]->` occurs in a different branch of the query's syntax tree. In GQL Standard, however, the above query is legal, because the evaluation of **`WHERE`** conditions is postponed for as long as possible.[5] While the meaning of the query is clear, its evaluation is non-trivial. The context of each condition (here, `y` is bound to ten successive edge ids) must be recorded, because it will be different when the evaluation occurs. Note that the evaluation of conditions must occur before the evaluation of **`SHORTEST`**, hence queries like

```
MATCH -[x]->,
ALL SHORTEST ( -[y]-> WHERE x.amount < y.amount ){10,10}
```

are not allowed in GQL.

Referencing the input table in conditions during pattern matching

In our semantics, the input table is not passed on to pattern matching, so one cannot refer to variables from it in `WHERE` conditions. As an example, the semantics of `LET x=42 MATCH (a WHERE a.amount=x)` is undefined.

3.4.3 Missing Features

Syntactic sugar

The GQL Standard includes a lot of syntactic sugar that we disregarded. For instance, several other types of edge patterns exist, such as `-[δ]-`, which matches edges regardless of their direction. Another example is the possibility of using `*` and `+` as shorthands for `{0,}` and `{1,}`.

Complex label expressions

We only allow a single label in descriptors, but the GQL Standard allows complex *label expressions*, as in `MATCH (a:YachtClub|(Person&!Account))`. Using `WHERE`, this could be rewritten as
`MATCH (a WHERE a:YachtClub OR (a:Person AND NOT a:Account))`

Label expressions can also use the special atom "`%`" to check the nonemptiness of the label set. For example, `MATCH (a:%)` matches nodes with at least one label and `MATCH (a:!%)` matches nodes with no labels. Note that "`%`" cannot be used to define a regular expression of labels, unlike its usage in the `LIKE` expressions of SQL.

Complex path modes

GQL allows more complex path modes than described here. Recall that `SHORTEST` partitions matched paths by endpoints and returns the shortest paths for each pair of endpoints. `SHORTEST` k `GROUPS` generalizes this: for each pair of endpoints, it returns all paths of length at most i_k, where $i_1 < i_2 < \cdots < i_k$ are the k smallest lengths of paths between these endpoints. `SHORTEST k PATHS` returns k shortest paths for each pair of endpoints.

Another mode present in GQL is `SIMPLE`: it is similar to `ACYCLIC` but allows the first and the last node on a path to be the same, i.e., a simple cycle. There is also the keyword `WALK` to explicitly indicate the absence of a path mode.

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"
[101]: Francis et al. (2018), *Formal Semantics of the Language Cypher*

GQL's `TRAIL` differs from Cypher's *trail semantics* [15, 101]. The latter corresponds to GQL's *match mode* `DIFFERENT EDGES`, which is omitted in this digest. Indeed, Cypher's requirement that all matched edges must be different operates at the level of graph patterns, whereas GQL's `TRAIL` operates at the level of path patterns. Hence, while the GQL query `MATCH TRAIL ()-[e1]->(), TRAIL ()-[e2]->()` will return bindings in which `e1` and `e2` are equal, `MATCH ()-[e1]>(), ()-[e2]->()` in Cypher would not; the latter behavior is captured by the GQL query

```
MATCH DIFFERENT EDGES ()-[e1]>(), ()-[e2]->().
```

Finally, we only use path modes at the beginning of path patterns. GQL's rules are more involved, in that they allow `TRAIL` and `ACYCLIC` to be used inside patterns.

Projection clauses

The GQL Standard includes several clauses similar to `RETURN`, such as `YIELD`, `PROJECT`, and `SELECT`. We ignored these because, although they are not allowed at the same positions in queries, they can be simulated by simple rewritings in terms of `RETURN`.

Combination of queries

In addition to set operations (`UNION`, etc.) and bag operations (`UNION ALL`, etc.), queries could be of the form Q_1 `OTHERWISE` Q_2. Its semantics is as follows: $[\![Q_1 \texttt{ OTHERWISE } Q_2]\!](T)$ equals $[\![Q_1]\!](T)$ if table $[\![Q_1]\!](T)$ is non-empty, otherwise it equals $[\![Q_2]\!](T)$.

Aggregation

The GQL Standard features two kinds of aggregation. The first one, much like `GROUP BY` in SQL, groups together bindings under which the evaluation of an expression produces the same value, then an aggregate value is computed for each group. The second kind aggregates along matched paths to compute a value, both during and after pattern matching. Computing the length of a path is a typical example; one can have more complex aggregates, such as the sum of the values `n.amount` for each node `n` in the path. This is similar to `reduce` in Cypher. The use of this feature in pattern matching requires strong syntactic restrictions for query evaluation to be decidable [102].

[102]: Francis et al. (2023), "GPC: A Pattern Calculus for Property Graphs"

Subqueries

GQL has a facility to run subqueries through the `CALL` Q clause, the semantics of which is roughly as follows: for each binding μ in the input table, $[\![Q]\!]_G(\{\mu\})$ is evaluated in a sub-process, and the resulting table is left-joined with the current working table. An important detail is that `CALL` can only expand bindings. It cannot remove columns from the input table nor change the values in them. The existence of read-only columns matters in clauses like `RETURN`, which cannot therefore be treated with our semantics as is. In GQL, this is handled with a notion of working record.

Note also that `CALL` Q will make nondeterminism much harder to detect if updates happen in Q. Tables are unordered sets (or bags) but in an update clause each binding causes changes in the graph (see next item) and so it can modify the evaluation of the clause for the next binding. In such cases, inconsistent changes may be detected [103].

[103]: Green et al. (2019), "Updating Graph Databases with Cypher"

Updates

Graph database updates in GQL are outside the scope of this thesis. They will work similarly to Cypher updates [103], by using clauses that can add and remove elements (`INSERT` and `DELETE`), or modify elements' attributes (`SET` and `REMOVE`). Therefore, pattern matching

[103]: Green et al. (2019), "Updating Graph Databases with Cypher"

and updates can be mixed together and result in bulk updates to the graph based on its contents, as in the example below:

```
MATCH  (a:Account)
INSERT (p:Person)
SET     p.name = a.owner
INSERT (p)-[:Owns]->(a)
REMOVE  a.owner
```

3.5 What the Future Holds

In this chapter we have summarized the key elements of GQL, the new standard graph query language.At the time when the first version of the SQL Standard was produced, many key elements of relational theory were already in place. For GQL, the standardization work is well ahead of the academic developments it should ideally be based upon. In what follows, we bring to the attention of the community several directions of academic work that will facilitate the development of graph query languages and their standardization.

Expressiveness and complexity

For relational query languages, the database research community has uncovered a rich landscape of fragments (conjunctive queries, positive queries, and queries with inequalities are some very well studied examples) and extensions (for example, adding counting and aggregation, or adding recursion as in many instantiations of Datalog), see [19, 20]. For these, we understand the trade-off between their expressiveness and the complexity of query evaluation. Here we have described a basic language for graphs, essentially the core of GQL, akin to relational algebra and calculus. Now we need to develop its theory, starting with understanding expressiveness and complexity and their trade-offs, in a way similar to what we know about relational databases. For the pattern matching facilities of GQL, shared with SQL/PGQ, some early results are available [102].

[19]: Arenas et al. (2022), *Database Theory*
[20]: Abiteboul et al. (1995), *Foundations of Databases*

[102]: Francis et al. (2023), "GPC: A Pattern Calculus for Property Graphs"

Query processing and optimization

Query processing and optimization is a central area in relational database research that needs yet to be developed for GQL. In a more theoretical level, the basis for understanding optimization is query equivalence and containment. We know a thing or two about containment for (conjunctive) regular path queries [104, 105] and extensions with data [106] but not for queries that resemble the real-life language. Moving to more practical aspects, one needs efficient and practical algorithms and data structures for processing graph queries in GQL, whether in a native system, or a relational implementation. Of course there is significant work in this direction [71, 72, 85, 107–110] but it needs to be adjusted to languages that could dominate the practical landscape for decades.

[104]: Calvanese et al. (2003), "Reasoning on regular path queries"
[105]: Figueira et al. (2020), "Containment of Simple Conjunctive Regular Path Queries"
[106]: Kostylev et al. (2018), "Containment of queries for graphs with data"
[71]: Vrgoč (2022), *Evaluating regular path queries under the all-shortest paths semantics*
[72]: Baier et al. (2017), "Evaluating Navigational RDF Queries over the Web"
[85]: Martens et al. (2022), *Representing Paths in Graph Database Pattern Matching*
[107]: Yakovets et al. (2016), "Query Planning for Evaluating SPARQL Property Paths"
[108]: Gubichev et al. (2013), "Sparqling Kleene: Fast property paths in RDF-3X"
[109]: Nguyen et al. (2015), "Join Processing for Graph Patterns: An Old Dog with New Tricks"
[110]: Hogan et al. (2019), "A Worst-Case Optimal Join Algorithm for SPARQL"

Design decisions and alternatives

We explained in Remark 3.3.4 how GQL currently forbids concatenating patterns that contain different kinds of variables. Notice, however,

that this current state reflects a design decision and it may be interesting to explore other avenues for graph query languages. For instance, one could consider a semantics in which both occurrences of x in Remark 3.3.4 should be bound to single nodes. Under such a semantics, the pattern would essentially perform a join on the even nodes of the path and would match "flower" shaped paths centered around node x, consisting of Transfer-loops of length two. Alternatively, one could consider a semantics in which, as soon as x occurs as a group variable, all occurrences of x are considered to be group variable occurrences. In this case, the query would match Transfer-paths of even length and bind x to the list of "even" nodes on such paths. In line with this work would be the study of an automaton model with group variables that would allow classical evaluation and automata-theoretic constructions such as the product, determinization, etc.

Since GQL is a complex language, there are many such places in which fundamental research can either help to validate the current design decisions or propose alternatives.

Updates

We have concentrated on the read-only part of the languages and have not touched updates. Designing a proper update language is not a simple task: in Cypher, for example, the initial design exhibited a multitude of problems [103]. GQL largely follows Cypher, which means its updates and transaction processing facilities need to be designed with care and subjected to the same research scrutiny as their relational counterpart.

[103]: Green et al. (2019), "Updating Graph Databases with Cypher"

Graph-to-graph queries

GQL, as its precursors including Cypher, is a very good tool for turning graphs into relations. The ever reappearing issue in the field of graph languages is how to design a graph-to-graph language whose queries output graphs. Queries are then composable: a query can be applied to the output of a previous one. We also regain such basic concepts as views and subqueries, taken for granted in relational databases, but very limited in the current graph database landscape.

Metadata

Looking into the future, we need to have a good schema language for graphs, and see how it interacts with graph query languages. Some efforts in this direction have already been made: the PG-Keys proposal introduces keys for property graphs [111] and more recently proposed PG-Schema [38] specifies a schema language for property graphs that should lead to future schema standards. As these are formulated, much theory needs to be developed, for example semantic query optimization, as well as incremental validation of schemas and constraints following work for relational and semistructured data [112, 113].

[111]: Angles et al. (2021), "PG-Keys: Keys for Property Graphs"

[38]: Angles et al. (2022), *PG-Schema: Schemas for Property Graphs*

[112]: Gupta et al. (1999), *Materialized Views: Techniques, Implementations, and Applications*

[113]: Barbosa et al. (2004), "Efficient Incremental Validation of XML Documents"

References

[6] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. "A Graphical Query Language Supporting Recursion". In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. Ed. by Umeshwar Dayal and Irving L. Traiger. ACM Press, 1987, pp. 323–330. DOI: 10.1145/38713.38749.

[15] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

[17] Alin Deutsch et al. "Aggregation Support for Modern Graph Analytics in TigerGraph". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2020, pp. 377–392. DOI: 10.1145/3318464.3386144.

[19] Marcelo Arenas et al. *Database Theory*. Open source at https://github.com/pdm-book/community, 2022.

[20] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[29] Alberto O. Mendelzon and Peter T. Wood. "Finding Regular Simple Paths in Graph Databases". In: *SIAM J. Comput.* 24.6 (1995), pp. 1235–1258.

[38] Renzo Angles et al. *PG-Schema: Schemas for Property Graphs*. 2022.

[57] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. "Regular Queries on Graph Databases". In: *Theory Comput. Syst.* 61.1 (2017), pp. 31–83. DOI: 10.1007/s00224-016-9676-2.

[60] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. "Querying Graphs with Data". In: *Journal of the ACM* 63.2 (2016), 14:1–14:53.

[71] Domagoj Vrgoč. *Evaluating regular path queries under the all-shortest paths semantics*. 2022.

[72] Jorge A. Baier et al. "Evaluating Navigational RDF Queries over the Web". In: *HT*. ACM, 2017, pp. 165–174. DOI: 10.1145/3078714.3078731.

[85] Wim Martens et al. *Representing Paths in Graph Database Pattern Matching*. 2022.

[90] Oskar van Rest et al. "PGQL: a property graph query language". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. 2016, pp. 1–6.

[91] Marko A. Rodriguez. "The Gremlin graph traversal machine and language". In: *DBPL*. ACM, 2015, pp. 1–10.

[92] Wikipedia contributors. *GQL Graph Query Language*. 2020. URL: https://en.wikipedia.org/wiki/GQL_Graph_Query_Language.

[93] Shumo Chu et al. "HoTTSQL: proving query rewrites with univalent SQL semantics". In: *PLDI*. ACM, 2017, pp. 510–524. DOI: 10.1145/3062341.3062348.

[94] Paolo Guagliardo and Leonid Libkin. "A Formal Semantics of SQL Queries, Its Validation, and Applications". In: *Proc. VLDB Endow.* 11.1 (2017), pp. 27–39. DOI: 10.14778/3151113.3151116.

[95] Véronique Benzaken and Evelyne Contejean. "A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra". In: *CPP*. ACM, 2019, pp. 249–261. DOI: 10.1145/3293880.3294107.

[96] Alin Deutsch et al. "Graph Pattern Matching in GQL and SQL/PGQ". In: *SIGMOD*. ACM, 2022, pp. 1–12.

[97] *GQL Influence Graph*. https://www.gqlstandards.org/existing-languages. Accessed: 2023-01-17. 2023.

[98] Mary F. Fernandez et al. "A Query Language for a Web-Site Management System". In: *SIGMOD Rec.* 26.3 (1997), pp. 4–11. DOI: 10.1145/262762.262763.

[99] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. "Querying the World Wide Web". In: *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*. IEEE Computer Society, 1996, pp. 80–91. DOI: 10.1109/PDIS.1996.568671.

[100] Diego Calvanese et al. "Containment of Conjunctive Regular Path Queries with Inverse". In: *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000*. 2000, pp. 176–185.

[101] Nadime Francis et al. *Formal Semantics of the Language Cypher*. 2018.

[102] Nadime Francis et al. "GPC: A Pattern Calculus for Property Graphs". In: *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA,*

USA, June 18-23, 2023. Ed. by Floris Geerts, Hung Q. Ngo, and Stavros Sintos. ACM, 2023, pp. 241–250. DOI: 10.1145/3584372.3588662.

[103] Alastair Green et al. "Updating Graph Databases with Cypher". In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2242–2253.

[104] Diego Calvanese et al. "Reasoning on regular path queries". In: *SIGMOD Record* 32.4 (2003), pp. 83–92.

[105] Diego Figueira et al. "Containment of Simple Conjunctive Regular Path Queries". In: *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 2020, pp. 371–380.

[106] Egor V. Kostylev, Juan L. Reutter, and Domagoj Vrgoc. "Containment of queries for graphs with data". In: *J. Comput. Syst. Sci.* 92 (2018), pp. 65–91. DOI: 10.1016/j.jcss.2017.09.005.

[107] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. "Query Planning for Evaluating SPARQL Property Paths". In: *SIGMOD Conference*. ACM, 2016, pp. 1875–1889. DOI: 10.1145/2882903.2882944.

[108] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. "Sparqling Kleene: Fast property paths in RDF-3X". In: *GRADES*. CWI/ACM, 2013. DOI: 10.1145/2484425.2484443.

[109] Dung T. Nguyen et al. "Join Processing for Graph Patterns: An Old Dog with New Tricks". In: *GRADES*. ACM, 2015, 2:1–2:8. DOI: 10.1145/2764947.2764948.

[110] Aidan Hogan et al. "A Worst-Case Optimal Join Algorithm for SPARQL". In: *ISWC (1)*. Vol. 11778. Lecture Notes in Computer Science. Springer, 2019, pp. 258–275. DOI: 10.1007/978-3-030-30793-6_15.

[111] Renzo Angles et al. "PG-Keys: Keys for Property Graphs". In: *SIGMOD '21: International Conference on Management of Data*. ACM, 2021, pp. 2423–2436.

[112] A. Gupta and I.S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[113] Denilson Barbosa et al. "Efficient Incremental Validation of XML Documents". In: *ICDE*. IEEE Computer Society, 2004, pp. 671–682. DOI: 10.1109/ICDE.2004.1320036.

4 The Graph Pattern Calculus

This chapter is joint work with N. Francis, A. Gheebrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund and D. Vrgoč and was published under the title "GPC: A Pattern Calculus for Property Graphs" at PODS 2023.

In the previous chapter, we introduced and explained the two new standard graph query languages GQL and SQL/PGQ, and the pattern matching they share. In this chapter, we proceed with the next step of formalization by defining a calculus which captures a smaller fragment of the languages but is a much better tool for theoretical study.

The development of GQL as a query language standard is rather different from SQL. The latter came out of a well-researched relational theory; relational calculus led to its declarative approach, while relational algebra formed the foundation of RDBMS implementations. But while GQL is "inspired" by the key developments of database research, they are not represented directly in the language, which itself was designed by an industry consortium. The GQL committee lists[1] three main academic influences: regular path queries [6], Graph XPath [60], and regular queries on graphs [57]. While these provided important initial orientation, the GQL development is much more in line with industry-level languages such as Cypher, GSQL, and PGQL.

1: `https://www.GQLstandards.org/existing-languages`

[6]: Cruz et al. (1987), "A Graphical Query Language Supporting Recursion"

[60]: Libkin et al. (2016), "Querying Graphs with Data"

[57]: Reutter et al. (2017), "Regular Queries on Graph Databases"

The theory of graph query languages on the other hand produced a multitude of languages based on RPQs: CRPQs, UCRPQs, 2(UC)RPQs, ECRPQs, just to name a few (see [9, 114] for many more). However, none of them can play the role of relational calculus with respect to the development of GQL, as they do not capture its key features with respect to both navigation and handling data.

[9]: Barceló (2013), "Querying graph databases"

[114]: Angles et al. (2017), "Foundations of Modern Query Languages for Graph Databases"

The goal of this chapter is to produce that missing piece, a theoretical language that underlies GQL and SQL/PGQ pattern matching and can be studied in the same way as the RPQ family has been studied over decades.

Such a calculus should judiciously choose the key features leaving others for extensions. Think again about the SQL/relational calculus analogy. The latter does not have bag semantics, nulls, aggregates, full typing, and many other SQL features. We adopt a similar approach here. Our goal is to introduce a calculus that captures the *essence* of GQL pattern matching, without every single feature present. Similarly to relational calculus, we settle for the core set-semantics fragment without nulls or aggregates, each of which can be added as extensions.

What is new? We now outline the main differences between the currently existing theoretical languages and real-life pattern matching in property graphs that we need to capture.

- Much of the theoretical literature relies on an overly simplified model of graph databases as edge-labeled graphs; this is common in the study of RPQs and extensions [9]. Typical patterns in languages like Cypher combine graph navigation with querying data held in nodes and edges. With data values, the theoretical model of choice is data graphs [60]: in those, each node carries

[9]: Barceló (2013), "Querying graph databases"

[60]: Libkin et al. (2016), "Querying Graphs with Data"

a single data value, similarly to data trees studied extensively in connection with tree-structured data [115]. Property graphs are yet more complex, as each edge or node can carry an *arbitrary collection of key-value pairs*.

[115]: Bojanczyk et al. (2006), "Two-variable logic on data trees and XML reasoning"

- GQL patterns bind variables in different ways. As an example, consider a pattern (in our notation, to be introduced in Section 4.1) $(x:a) \xrightarrow{e:b} (y:a)$ that looks for b-labeled edges between two a-labeled elements. While producing a match, it binds variables x and y to the start and end-nodes of the edge, and e to the edge itself. If, on the other hand, we look at $(x:a) \xrightarrow{e:b}{}^{1..\infty} (y:a)$, then we match paths of length 1 or more (indicated by $1..\infty$) from x to y. In this case e gets bound not to an edge, but rather to a *list of edges*. This has immediate implications on conditions in which such a binding can be used.
- Property graphs are multigraphs (there can be multiple edges between two endpoint nodes), pseudographs (there can be an edge looping from a node to itself), and mixed, or partially directed graphs (as an edge can be directed or undirected). This means edges can be traversed in different directions, or traversals can be indicated as having no direction at all.
- To ensure the number of paths returned is finite, real-life languages put additional restrictions on paths, such as insisting that they be trails (no repeated edges, as in Cypher), simple (no repeated nodes), or shortest (as in G-Core [18]). Typically in research literature one considers each one of those semantics separately, but GQL permits mixing them.

[18]: Angles et al. (2018), "G-CORE: A Core for Future Graph Query Languages"

- As in Cypher, GQL allows to match paths and output them. In theoretical languages this feature is rather an exception [11, 18].
- Finally, one can apply conditions to filter matched paths. For example, after matching $(x:a) \xrightarrow{e:b}{}^{1..\infty} (y:a)$, one can apply a condition $x.k = y.k$ stating that property k of both x and y is the same. Notice that we cannot talk similarly about properties of e, as they are bound to a list.

[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"
[18]: Angles et al. (2018), "G-CORE: A Core for Future Graph Query Languages"

Other features of patterns are those in regular languages: concatenation, disjunction, repetition; they can be applied on top of already existing patterns, similarly to [57].

[57]: Reutter et al. (2017), "Regular Queries on Graph Databases"

The main contribution of this chapter is the Graph Pattern-matching Calculus GPC that captures all the key features of GQL and SQL/PGQ pattern matching. Its syntax is described in Section 4.1. To ensure the well-definedness of its expressions, the calculus comes with a *type system*, given in Section 4.2. We give a formal semantics of GPC in Section 4.3, and provide a number of basic results on the complexity of the language, and its relationship with classical theoretical formalisms in Section 4.4. In Section 4.5 we outline some possible extensions and describe two concrete examples where theoretical studies of the language had a direct impact on the Standard as it was being written.

4.1 Pattern calculus

First note that throughout this Chapter we will use the standard definition of property graph and graph database as introduced in Chapter 2 Section 2.3.

The basic building blocks of GPC are node and edge (or arrow) patterns.

BASICS For $x, y \in \text{Vars}$, $\ell \in \mathscr{L}$, $a, b \in \mathscr{K}$, $c \in \text{Const}$:

$$
\begin{array}{rcll}
d & ::= & x \mid :\ell \mid x:\ell & \text{descriptor} \\
\leftrightarrow & ::= & \rightarrow \mid \leftarrow \mid - & \text{direction} \\
\theta & ::= & x.a = c \mid x.a = y.b & \text{condition} \\
 & & \mid \theta \vee \theta \mid \neg\theta & \\
\rho & ::= & \text{simple} \mid \text{trail} \mid \text{shortest} & \text{restrictor} \\
 & & \mid \text{shortest simple} \mid \text{shortest trail} &
\end{array}
$$

PATTERNS For $0 \le n \le m \le \infty$:

$$
\begin{array}{rcll}
\psi & ::= & () \mid (d) & \text{node pattern} \\
 & & \mid \leftrightarrow \mid \overset{d}{\leftrightarrow} & \text{edge pattern} \\
 & & \mid \psi + \psi & \text{union} \\
 & & \mid \psi\,\psi & \text{concatenation} \\
 & & \mid \psi_{\langle\theta\rangle} & \text{conditioning} \\
 & & \mid \psi^{n..m} & \text{repetition}
\end{array}
$$

QUERIES For $x \in \text{Vars}$:

$$
\begin{array}{rcll}
Q & ::= & \rho\,\psi \mid x = \rho\,\psi & \text{path pattern} \\
 & & \mid Q, Q & \text{join}
\end{array}
$$

Figure 4.1: GPC expressions

Node patterns are of the form $(x:\ell)$. Here x is a variable, and $:\ell$ specifies the node label. The brackets "(" and ")" are mandatory, and signify that we are talking about a node. Both the variable x, and the label specification $:\ell$ are optional, and can be omitted. This way, the simplest node pattern is $()$, matching any node in the graph. The presence of a variable means that it gets bound[2]; the presence of a label ℓ means that only ℓ-nodes are matched.

Edge patterns are of the form $\overset{x:\ell}{\leftrightarrow}$, where again x and ℓ are a variable and an edge label, respectively, and $\leftrightarrow$ is one of the allowed directions: $\rightarrow$ (*forward*), $\leftarrow$ (*backward*), $-$ (*undirected*). Both x and $:\ell$ can be omitted. In the case when they are present, the variable x gets bound to the matching edge, and $:\ell$ constrains the allowed edge labels.

The full grammar of GPC is given in Fig. 4.1. Here:

- d specifies *node* and *edge descriptors*; these may include a variable to which that graph element is bound, and its label.
- $\leftrightarrow$ specifies edge directions: forward, backward, or undirected.
- θ defines conditions: atomic ones compare property values held in nodes or edges to one another or to constants, and conditions are closed under Boolean connectives.
- ρ specifies restrictors on paths to ensure a finite result set; paths can be restricted to be simple (no repeated nodes), trail (no repeated edges), or shortest, which can be optionally combined with simple or trail.
- ψ defines patterns: the atomic ones are node and edge patterns, which have an optional descriptor and, for the latter, a mandatory direction; patterns are then built from these using concatenation

2: We use the same definition of bindings and lists as in Chapter 3 throughout this Chapter

(denoted by juxtaposition), union (+), conditioning (akin to selection in relational algebra), and repetition of the form $n..m$, meaning that the pattern is repeated between n and m times. Note that the $0..\infty$ is precisely the Kleene star.

Q defines a *query*: a non-empty list of optionally named ($x = \rho\psi$) path patterns, each qualified by a restrictor.

When we write patterns, we disambiguate with square brackets, and the lower operator takes precedence. For instance, $\psi\psi'_{\langle\theta\rangle} + \psi''$ is equivalent to $[\psi[\psi'_{\langle\theta\rangle}]] + \psi''$. By an *expression* of GPC we shall mean a pattern or a query.

Examples The formal semantics is presented in Section 4.3. Next we illustrate how GPC operates with several examples. Each path pattern is matched to a path; such a path could be a single node, an edge (with endpoints included), or a more complex path. For example, the pattern

$$(x_1 : A) \xrightarrow{y_1} (x_2 : B) \xleftarrow{y_2} (x_3 : C) \xrightarrow{y_3} (x_1)$$

matches a path from an A-node to itself via B- and C-nodes with the first and third edges going forward and the second edge going backward. Notice that this pattern introduces an implicit join over the endpoints of the path by repeating the variable x_1.

The pattern $(x : A) \rightarrow (z : B) \left[\leftarrow (u : C) + ( )\right]$ is an *optional pattern*[3], a feature present in many languages such as SPARQL and Cypher. It matches an edge from an A-node to a B-node, binding x and z to its endpoints, and if the B-node has an incoming edge from a C-node, binds u to its source. This pattern can be seen as the disjunction $\psi_1 + \psi_2$ where $\psi_1 = (x : A) \rightarrow (z : B) \leftarrow (u : C)$ and $\psi_2 = (x : A) \rightarrow (z : B)\ ( )$. In ψ_2, the concatenated $( )$ must match the same node as z and thus it has no effect on the pattern; this accounts for the case where the node bound to z has no incoming edge from a C-node.

3: Note that we use square brackets for grouping, since $( )$ defines a node pattern.

The pattern $(x : A) \xrightarrow{y}{}^{1..\infty} (z : B)$ looks for paths of arbitrary positive length from an A-node to a B-node. It uses variable y to bind edges encountered on this path. Unlike the bindings for x and z, which are unique nodes, there may well be multiple edges encountered on the path between them, and thus y needs to be bound to a complex object encoding all such edges on a path. Intuitively, in this case y binds to the list of edges on a matching path. However, in general the binding for such variables, which we call group variables, is more complex. Indeed, consider a pattern $\psi^{n..m}$ and a particular match of this pattern in which ψ is repeated k times, $n \leq k \leq m$. These k repetitions of ψ are matched by paths $p_1, \ldots, p_k$, and thus for every variable used in ψ we need to record not only which elements of $p_1 \cdots p_k$ it binds to, but also in which paths p_i these elements occur. Thus, in general, group variables will be bound to lists of (path, graph element) pairs.

The pattern $\left[(x : A) \xrightarrow{y}{}^{1..\infty} (z : B)\right]_{\langle x.a = z.a\rangle}$ is an example of a conditioned pattern; here the condition ensures that the value of property a is the same at the endpoints of the path. Note that conditions cannot compare nodes or edges, only their properties.

A pattern cannot be used by itself as a query; for example, in the pattern $u = \left[(x : A) \xrightarrow{y}{}^{1..\infty} (z : B)\right]$, the variable u can be bound to infinitely many paths. Indeed, if there is a loop on some path from x to z, it can be traversed arbitrarily many times, while still satisfying the condition of the pattern. To deal with this, every pattern in a query is compulsorily

$$\frac{}{(x) \vdash x : \text{Node}} \qquad \frac{}{(x : \ell) \vdash x : \text{Node}} \qquad \frac{}{\overset{x}{\Leftrightarrow} \vdash x : \text{Edge}} \qquad \frac{}{\overset{x:\ell}{\Longleftrightarrow} \vdash x : \text{Edge}} \qquad \frac{x \notin \text{var}(\psi)}{x = \rho\, \psi \vdash x : \text{Path}}$$

$$\frac{\psi \vdash z : \tau}{\psi^{n..m} \vdash z : \text{Group}(\tau)} \qquad \frac{\psi \vdash z : \tau}{\rho\, \psi \vdash z : \tau} \qquad \frac{\psi \vdash z : \tau \quad z \neq x}{x = \rho\, \psi \vdash z : \tau}$$

$$\frac{\psi \vdash x : \tau \quad \tau \in \{\text{Node}, \text{Edge}\}}{\psi \vdash x.a = c : \text{Bool}} \qquad \frac{\psi \vdash x : \tau \quad \psi \vdash y : \tau' \quad \tau, \tau' \in \{\text{Node}, \text{Edge}\}}{\psi \vdash x.a = y.b : \text{Bool}}$$

$$\frac{\psi \vdash \theta : \text{Bool} \quad \psi \vdash \theta' : \text{Bool}}{\psi \vdash \theta \wedge \theta' : \text{Bool}} \qquad \frac{\psi \vdash \theta : \text{Bool} \quad \psi \vdash \theta' : \text{Bool}}{\psi \vdash \theta \vee \theta' : \text{Bool}} \qquad \frac{\psi \vdash \theta : \text{Bool}}{\psi \vdash \neg\theta : \text{Bool}} \qquad \frac{\psi \vdash \theta : \text{Bool} \quad \psi \vdash z : \tau}{\psi_{\langle\theta\rangle} \vdash z : \tau}$$

$$\frac{\psi_1 \vdash z : \tau \quad \psi_2 \vdash z : \tau}{\psi_1 + \psi_2 \vdash z : \tau} \qquad \frac{\psi_1 \vdash z : \tau \quad \psi_2 \vdash z : \text{Maybe}(\tau)}{\psi_1 + \psi_2 \vdash z : \text{Maybe}(\tau)} \qquad \frac{\psi_1 \vdash z : \text{Maybe}(\tau) \quad \psi_2 \vdash z : \tau}{\psi_1 + \psi_2 \vdash z : \text{Maybe}(\tau)}$$

$$\frac{\psi_1 \vdash z : \tau \quad z \notin \text{var}(\psi_2)}{\psi_1 + \psi_2 \vdash z : \tau?} \qquad \frac{\psi_2 \vdash z : \tau \quad z \notin \text{var}(\psi_1)}{\psi_1 + \psi_2 \vdash z : \tau?}$$

$$\frac{\psi_1 \vdash z : \tau \quad \psi_2 \vdash z : \tau \quad \tau \in \{\text{Node}, \text{Edge}\}}{\psi_1\, \psi_2 \vdash z : \tau} \qquad \frac{\psi_1 \vdash z : \tau \quad z \notin \text{var}(\psi_2)}{\psi_1\, \psi_2 \vdash z : \tau} \qquad \frac{\psi_2 \vdash z : \tau \quad z \notin \text{var}(\psi_1)}{\psi_1\, \psi_2 \vdash z : \tau}$$

$$\frac{Q_1 \vdash z : \tau \quad Q_2 \vdash z : \tau \quad \tau \in \{\text{Node}, \text{Edge}\}}{Q_1, Q_2 \vdash z : \tau} \qquad \frac{Q_1 \vdash z : \tau \quad z \notin \text{var}(Q_2)}{Q_1, Q_2 \vdash z : \tau} \qquad \frac{Q_2 \vdash z : \tau \quad z \notin \text{var}(Q_1)}{Q_1, Q_2 \vdash z : \tau}$$

Figure 4.2: Typing rules for the GPC type system.

preceded by a restrictor, e.g., $u = \text{trail}\,[(x : A) \xrightarrow{y}{}^{1..\infty} (z : B)]$. Then only trails, of which there are finitely many, that satisfy the conditions of ψ will be returned as values of variable u.

The necessity of type rules The calculus defined in Fig. 4.1 is very permissive and allows expressions that do not type-check. For example, $(x) \xrightarrow{x} ( )$ is syntactically permitted even though it equates a node variable with an edge variable. As another example, adding the condition $x.a = y.a$ to the pattern $(x : A) \xrightarrow{y}{}^{1..\infty} (z : B)$ seen above would result in comparing a singleton with a list of pairs. The type system introduced next eliminates such mismatches.

4.2 Type System

The goal of the type system is to ensure that GPC expressions do not exhibit the pathological behavior explained at the end of the previous section.

The set Types of *types* used to type variables is defined by the following grammar

$$\tau ::= \mathsf{Node} \mid \mathsf{Edge} \mid \mathsf{Path} \mid \mathsf{Maybe}(\tau) \mid \mathsf{Group}(\tau).$$

The three atomic types are used for variables returning nodes, edges, and paths, respectively. The type constructor Maybe is used for variables occurring on one side of a disjunction only, while Group is used for variables occurring under repetition, whose bindings are grouped together. As variables in GPC are never bound to property values, we do not need the usual types like integers or strings. However, to eliminate references to unbound variables, we do need to type conditions (such as for $\langle x.a = y.a \rangle$ in the example at the end of Section 4.1); we use an additional type Bool for that.

Typing statements are of the form $\xi \vdash x : \tau$ stating that in expression ξ (a pattern or a query), we can derive that variable x has type τ, and $\xi \vdash \theta : \mathsf{Bool}$, stating that a condition is correctly typed as a Boolean value under the typing of other variables.

The typing rules are presented in Figure 4.2. Here, $\mathsf{var}(\xi)$ stands for the set of variables used in expression ξ.

For a type τ we let $\tau? = \tau$ if $\tau = \mathsf{Maybe}(\tau')$ for some τ' and $\tau? = \mathsf{Maybe}(\tau)$ otherwise.

The first five rules state that variables in node/edge patterns, and variables naming paths, are typed accordingly. The next four rules say that variables of group type appear in repetition patterns, and that restrictors and path naming do not affect typing.

The next two lines deal with typing conditions: property values of singletons can be compared for equality; correctly typed conditions do not affect the typing of variables in a pattern; and conditions are closed under Boolean connectives.

The following two lines deal with the optional type $\mathsf{Maybe}(\tau)$. It is bound to a variable z in a disjunction $\psi_1 + \psi_2$ if in one of the patterns z is of type τ and in the other z is either not present or of type $\mathsf{Maybe}(\tau)$.

Derivation rules for concatenation $\psi_1 \psi_2$ and join Q_1, Q_2 are similar: a variable is allowed to appear in both expressions only if it is typed as a node or an edge in both, or it inherits its type from one when it does not appear in the other.

Definition 4.2.1 — Well-typed expressions

An expression is well-typed if for every variable used in it, its type can be derived according to the typing rules.

A well-typed expression assigns a unique type to every variable appearing in it, and only to such variables.

Proposition 4.2.2

For every well-typed expression ξ, variable x, and types τ, τ',
- $\xi \vdash x : \tau$ implies $x \in \mathsf{var}(\xi)$;
- $\xi \vdash x : \tau$ and $\xi \vdash x : \tau'$ imply that $\tau = \tau'$.

Proof. The first item holds because a variable appears in the conclusion of an inference rule only if it appears in one of its premises, or explicitly in the expression. The second item holds because all inference rules have mutually exclusive premises. □

Definition 4.2.3

Let Θ denote a binary operator from GPC (Fig. 4.1). We say that Θ is *associative* (resp. *commutative*) with respect to the type system if the condition (4.1) (resp. (4.2)) below holds for all expressions ξ_1, ξ_2, ξ_3, types τ, and variables x:

$$(\xi_1 \,\Theta\, \xi_2) \,\Theta\, \xi_3 \vdash x : \tau \iff \xi_1 \,\Theta\, (\xi_2 \,\Theta\, \xi_3) \vdash x : \tau\,, \tag{4.1}$$

$$\xi_1 \,\Theta\, \xi_2 \vdash x : \tau \iff \xi_2 \,\Theta\, \xi_1 \vdash x : \tau\,. \tag{4.2}$$

The definition for the binary join on queries is the same.

Proposition 4.2.4

- Union, concatenation and join are associative and commutative with respect to the type system.
- There is no expression ξ, variable x, and type τ such that $\xi \vdash x : \mathsf{Maybe}(\mathsf{Maybe}(\tau))$.

A *schema* σ is a partial function from variables Vars to types Types, with a finite domain. With each well-typed expression ξ we can naturally associate a schema $\mathsf{sch}(\xi)$, induced by the types derived from ξ. It is defined formally below and it is well-defined by Proposition 4.2.2.

Definition 4.2.5 — Schema

Given a well-typed expression ξ, the schema of ξ, written $\mathsf{sch}(\xi)$, is the schema that maps each variable $x \in \mathsf{var}(\xi)$ to the unique type τ such that $\xi \vdash x : \tau$.
A variable x in $\mathsf{var}(\xi)$ is called

- a *singleton variable* if $\mathsf{sch}(\xi)(x) \in \{\mathsf{Node}, \mathsf{Edge}\}$;
- a *conditional variable* if $\mathsf{sch}(\xi)(x) = \mathsf{Maybe}(\tau)$ for some τ;
- a *group variable* if $\mathsf{sch}(\xi)(x) = \mathsf{Group}(\tau)$ for some τ;
- a *path variable* if $\mathsf{sch}(\xi)(x) = \mathsf{Path}$.

Remark 4.2.6 — Schema compositionality

It is easily checked that the function sch is compositional, in the sense that $\mathsf{sch}(\xi_1 \,\Theta\, \xi_2)$, for some binary operator Θ can be computed by a function that depends only on Θ and takes as arguments $\mathsf{sch}(\xi_1)$ and $\mathsf{sch}(\xi_2)$ (and likewise for unary operators).

4.3 Semantics

We begin by defining values, which is what can be returned by a query. Since GPC returns references to graph elements, not the data they bear, elements of Const are not values.

Definition 4.3.1 GPC Values

Given a type $\tau \in \mathrm{Types}$, the set $\mathcal{V}_\tau$ of values of type τ is defined inductively as follows

- $\mathcal{V}_{\mathrm{Node}} = \mathcal{N}$, $\mathcal{V}_{\mathrm{Edge}} = \mathcal{E}_{\mathrm{d}} \cup \mathcal{E}_{\mathrm{u}}$, $\mathcal{V}_{\mathrm{Path}} = \mathrm{Paths}$;
- $\mathcal{V}_{\mathrm{Maybe}()} = \mathcal{V}_\tau \cup \{\mathrm{Nothing}\}$ for a special value Nothing;
- $\mathcal{V}_{\mathrm{Group}()}$ is the set of all composite values of the form

$$\mathrm{list}\big((p_1, v_1), \ldots, (p_n, v_n)\big)$$

where $n \geq 0$ and $p_i \in \mathcal{V}_{\mathrm{Path}}$ and $v_i \in \mathcal{V}_\tau$ for all $i \in [1, n]$.

The set of all values is $\mathcal{V} = \bigcup_{\tau \in \mathrm{Types}} \mathcal{V}_\tau$.

As in Chapter 3, the semantics of GPC is defined in terms of *bindings* from variables to values. Recall that a binding μ is a partial function from Vars to $\mathcal{V}$, with finite domain. We write $\mu_\emptyset$ for the empty binding; that is, a binding that binds no variables. The values bound to variables of a well-typed pattern or query should respect its schema: a binding μ *conforms* to a schema σ if $\mathrm{dom}(\mu) = \mathrm{dom}(\sigma)$ and $\mu(x) \in \mathcal{V}_{\sigma(x)}$ for all $x \in \mathrm{dom}(\mu)$.

We say that two bindings μ and μ' *unify* if $\mu(x) = \mu'(x)$ for all $x \in \mathrm{dom}(\mu) \cap \mathrm{dom}(\mu')$. In that case, we define their *unification* $\mu \cup \mu'$ by setting $(\mu \cup \mu')(x) = \mu(x)$ if $x \in \mathrm{dom}(\mu)$ and $(\mu \cup \mu')(x) = \mu'(x)$ otherwise, for all $x \in \mathrm{dom}(\mu) \cup \mathrm{dom}(\mu')$. If S is a family of bindings that pairwise unify, then their unification is associative, and we write it as $\bigcup S = \bigcup_{\mu \in S} \mu$.

The semantics of a well-typed GPC expression ξ on a property graph G is a pair $\big(\mathrm{sch}(\xi), [\![\xi]\!]_G\big)$, where $\mathrm{sch}(\xi)$ is the schema of ξ (see Section 4.2), and $[\![\xi]\!]_G$ is the set of answers to ξ on G.

An *answer* to ξ on G is a pair $(\bar{p}, \mu)$, where $\bar{p}$ is a tuple of paths in G, and μ is a binding that conforms to $\mathrm{sch}(\xi)$. If ξ is a pattern, $\bar{p}$ consists of a single path p, in which case we simply write p instead of (p). If ξ is a query, $\bar{p}$ contains one path for each joined pattern.

By Remark 4.2.6, $\mathrm{sch}(\xi)$ can be computed compositionally independently of $[\![\xi]\!]_G$. In what follows, we shall define $[\![\xi]\!]_G$ using $\mathrm{sch}(\xi)$ and $[\![\xi']\!]_G$ for direct subexpressions ξ' of ξ. Hence, the semantics of expressions (patterns and queries), i.e., the function $\xi \mapsto \big(\mathrm{sch}(\xi), [\![\xi]\!]_G\big)$, is compositional.

For the remainder of this section, we consider a fixed property graph $G = \langle N, E_{\mathrm{d}}, E_{\mathrm{u}}, \lambda, \mathrm{endpoints}, \mathrm{src}, \mathrm{tgt}, \delta \rangle$.

Semantics of atomic patterns

For the sake of brevity, here we write atomic patterns as if all components were present, but still allow the possibility that some of them may be absent. Hence, $(x : \ell)$ subsumes the cases (x), $(: \ell)$, and $()$.

$$[\![(x : \ell)]\!]_G = \{\, (\mathrm{path}(n), \mu) \mid n \in N,\ \ell \in \lambda(n) \text{ if } \ell \text{ is present} \,\}$$

where $\mu = \{x \mapsto n\}$ if x is present, and $\mu = \mu_\varnothing$ otherwise.

$$\left[\!\left[\xrightarrow{x:\ell}\right]\!\right]_G = \left\{ (\mathrm{path}(u_1, e, u_2), \mu') \;\middle|\; \begin{array}{l} e \in E_d, \\ u_1 = \mathrm{src}(e),\ u_2 = \mathrm{tgt}(e), \\ \ell \in \lambda(e) \text{ if } \ell \text{ is present} \end{array} \right\}$$

$$\left[\!\left[\xleftarrow{x:\ell}\right]\!\right]_G = \left\{ (\mathrm{path}(u_2, e, u_1), \mu') \;\middle|\; \begin{array}{l} e \in E_d, \\ u_1 = \mathrm{src}(e),\ u_2 = \mathrm{tgt}(e), \\ \ell \in \lambda(e) \text{ if } \ell \text{ is present} \end{array} \right\}$$

$$\left[\!\left[\overset{x:\ell}{---}\right]\!\right]_G = \left\{ (\mathrm{path}(u_1, e, u_2), \mu') \;\middle|\; \begin{array}{l} e \in E_u, \\ \mathrm{endpoints}(e) = \{u_1, u_2\}, \\ \ell \in \lambda(e) \text{ if } \ell \text{ is present} \end{array} \right\}$$

where $\mu' = \{x \mapsto e\}$ if x is present, and $\mu' = \mu_\varnothing$ otherwise. Observe that the pattern $\overset{x:\ell}{---}$ returns both $\mathrm{path}(u_1, e, u_2)$ and $\mathrm{path}(u_2, e, u_1)$ if $\mathrm{endpoints}(e) = \{u_1, u_2\}$ with $u_1 \neq u_2$, but only one of them if $u_1 = u_2$ since both paths are the same.

Semantics of concatenation

$$[\![\psi_1\, \psi_2]\!]_G = \left\{ (p_1 \cdot p_2, \mu_1 \cup \mu_2) \;\middle|\; \begin{array}{l} (p_i, \mu_i) \in [\![\psi_i]\!]_G \text{ for } i = 1, 2, \\ p_1 \text{ and } p_2 \text{ concatenate}, \\ \mu_1 \text{ and } \mu_2 \text{ unify} \end{array} \right\}$$

The typing system ensures that all variables shared by ψ_1 and ψ_2 are singleton variables (otherwise $\psi_1\psi_2$ would not be well-typed). In other words, implicit joins over group and optional variables are disallowed (path variables do not occur in patterns at all).

Semantics of union

$$[\![\psi_1 + \psi_2]\!]_G = \{\, (p, \mu \cup \mu') \mid (p, \mu) \in [\![\psi_1]\!]_G \cup [\![\psi_2]\!]_G \,\}$$

where μ' maps every variable in $\mathrm{dom}\big(\mathrm{sch}(\psi_1 + \psi_2)\big) \setminus \mathrm{dom}(\mu)$ to Nothing. Note that here we rely on $\mathrm{sch}(\psi_1 + \psi_2)$.

Semantics of conditioned patterns

$$\left[\!\left[\psi_{\langle\theta\rangle}\right]\!\right]_G = \{\, (p, \mu) \in [\![\psi]\!]_G \mid \mu \vDash \theta \,\}$$

where $\mu \vDash \theta$ is defined inductively as follows:

- $\mu \vDash (x.a = c)$ iff $\delta\big(\mu(x), a\big)$ is defined and equal to c;
- $\mu \vDash (x.a = y.b)$ iff $\delta(\mu(x), a)$ is defined, $\delta\big(\mu(y), b\big)$ is defined and $\delta(\mu(x), a) = \delta\big(\mu(y), b\big)$;
- $\mu \vDash (\theta_1 \wedge \theta_2)$ iff $\mu \vDash \theta_1$ and $\mu \vDash \theta_2$;
- $\mu \vDash (\theta_1 \vee \theta_2)$ iff $\mu \vDash \theta_1$ or $\mu \vDash \theta_2$;
- $\mu \vDash (\neg\theta)$ iff $\mu \nvDash \theta$.

Semantics of repeated patterns

$$[\![\psi^{n..m}]\!]_G = \bigcup_{i=n}^{m} [\![\psi]\!]_G^i$$

Above, for a pattern ψ and a natural number $n \geq 0$, we use $[\![\psi]\!]_G^n$ to denote the n-th power of $[\![\psi]\!]_G$, defined as follows. We let

$$[\![\psi]\!]_G^0 = \{\, (\mathsf{path}(u), \mu) \mid u \text{ is a node in } G \,\}$$

where μ is the binding that maps each variable in $\mathsf{dom}(\mathsf{sch}(\psi))$ to $\mathsf{list}()$, the empty composite value. For $n > 0$, we let

$$[\![\psi]\!]_G^n = \left\{ (p, \mu) \;\middle|\; \begin{array}{l} (p_1, \mu_1), \ldots, (p_n, \mu_n) \in [\![\psi]\!]_G \\ p = p_1 \cdot \ldots \cdot p_n \\ \mu = \mathbf{collect}\big((p_1, \mu_1), \ldots, (p_n, \mu_n)\big) \end{array} \right\}$$

where $\mathbf{collect}[(p_1, \mu_1), \ldots, (p_n, \mu_n)]$ is a binding defined, and discussed, below. In the case that ψ does not have any variables, **collect** simply returns a function with empty domain. If ψ does contain variables, then each such variable is mapped to a list. (As such, nesting of patterns of the form $\psi^{n..m}$ leads to nesting of lists.)

There are several ways to define **collect** and obtain a sound semantics. In all cases, **collect** takes as input any number of path/binding pairs $(p_1, \mu_1), \ldots, (p_n, \mu_n)$, such that $n > 0$ and $p_1, \ldots, p_n$ concatenate to a path $p = p_1 \cdots p_n$. Furthermore, by our inductive definition of the semantics, it will always be the case that $\mu_1, \ldots, \mu_n$ all have the same domain, that we denote by D. Then, $\mathbf{collect}\big((p_1, \mu_1), \ldots, (p_n, \mu_n)\big)$ is a binding that maps every $x \in D$ to a $\mathsf{list}((p'_1, v_1), \ldots, (p'_\ell, v_\ell))$ where each p'_i is a portion of the matched path (they collectively satisfy $p'_1 \cdots p'_\ell = p_1 \cdots p_n$) and v_i is the value associated to x for that portion.

If all p_i's have a positive length (i.e., have at least one edge), **collect** is simply defined as follows.

$$\forall x \in D \quad \mathbf{collect}\big((p_1, \mu_1), \ldots, (p_n, \mu_n)\big)(x) = \mathsf{list}((p_1, \mu_1(x)), \ldots, (p_n, \mu_n(x))) \tag{4.3}$$

Although **collect** is still well defined by (4.3) if some of the p_i's have length 0, the above definition may lead to infinite query results. To avoid this, we outline three different approaches.

Approach 1: Syntactic restrictions We add a syntactic restriction that prevents the case from ever appearing: pattern $\psi^{n..m}$ is forbidden if pattern ψ may match an edgeless path. The latter is defined inductively: every edge pattern is allowed; if ψ is allowed then so are $\psi_{\langle\theta\rangle}$, $\psi^{n..m}$, $\psi\psi'$ and $\psi'\psi$ for every condition θ, $n > 0$, and pattern ψ'; if ψ_1 and ψ_2 are allowed then so is $\psi_1 + \psi_2$. This is the solution adopted by the GQL standard: the minimum path length of ψ must be positive. The drawback of this solution is that it rules out syntactically some patterns for which (4.3) would result in a well-defined finite semantics.

Approach 2: Run-time restriction As an alternative, the precondition for **collect** well-definedness can be checked at run-time, i.e., it is only defined if all p_i's have a positive length. While not imposing any additional restrictions, this approach has a drawback that ψ may have some result while $\psi^{1..1}$ has none, for some pattern ψ.

Approach 3: Grouping edgeless paths To overcome problems with the first two approaches, we propose a more general semantics of **collect**

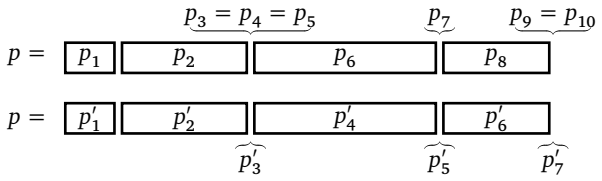


Figure 4.3: Refactorization of a path $p = p_1 p_2 \cdots p_{10}$ as $p = p'_1 p'_2 \cdots p'_7$ by grouping consecutive edgeless factors

that groups together consecutive edgeless paths from $p_1, \cdots, p_n$. If no such paths exist, either due to syntactic restriction or ruling them out at run-time, the result of this approach coincides with (4.3); thus this approach subsumes the other two.

We define $p'_1, \cdots, p'_\ell$ as a coarser factorization of $p_1 \cdots p_n$: each p'_i is the concatenation of successive (p_j)'s, in which consecutive edgeless paths are grouped together, as shown in Figure 4.3. Formally, the p'_is are defined as the unique path sequence such that there exists $i_1 < i_2 < \cdots < i_{\ell+1}$ (delineating the boundaries of $p'_1, \cdots, p'_\ell$) with $i_1 = 1$, $i_{\ell+1} = n + 1$ and satisfying the following.

$$\begin{aligned}
\forall k \in \{1, \dots, \ell\} \quad & p'_k = p_{i_k} \cdots p_{i_{k+1}-1} \\
\forall k \in \{1, \dots, \ell - 1\} \quad & \mathsf{len}(p_{i_k}) \neq 0 \vee \mathsf{len}(p_{i_{k+1}}) \neq 0 \\
\forall k \in \{1, \dots, \ell\} \quad & \begin{cases} \text{either} & i_{k+1} = i_k + 1 \text{ and } \mathsf{len}(p_{i_k}) \neq 0 \\ \text{or} & \forall i\,, i_k \leq i < i_{k+1},\ \mathsf{len}(p_i) = 0 \end{cases}
\end{aligned}$$

The binding $\mathbf{collect}\big((p_1, \mu_1), \dots, (p_n, \mu_n)\big)$ is defined only if

$$\forall k \in \{1, \dots, \ell\} \quad \mu_{i_k}, \dots, \mu_{(i_{k+1}-1)} \text{ pairwise unify}$$

Then their unification is denoted by μ'_k and **collect** is defined by

$$\forall x \in D \quad \mathbf{collect}\big((p_1, \mu_1), \dots, (p_n, \mu_n)\big)(x) = \mathsf{list}\big((p'_1, \mu'_1(x)), \cdots, (p'_\ell, \mu'_\ell(x))\big)$$

Remark 4.3.2

For the purpose of **collect**, one could use a weaker definition for unification that would allow μ and μ' to unify if, or every $x \in \mathrm{dom}(\mu) \cap \mathrm{dom}(\mu')$, any of the following holds: $\mu(x) = \mathsf{Nothing}$, $\mu'(x) = \mathsf{Nothing}$ or $\mu(x) = \mu'(x)$. This would allow even more combinations than the definition above.

Semantics of queries

$$[\![\mathsf{trail}\ \psi]\!]_G = \left\{ (p, \mu) \in [\![\psi]\!]_G \;\middle|\; \begin{array}{l} \text{no edge occurs more than} \\ \text{once in } p \end{array} \right\}$$

$$[\![\mathsf{simple}\ \psi]\!]_G = \left\{ (p, \mu) \in [\![\psi]\!]_G \;\middle|\; \begin{array}{l} \text{no node occurs more than} \\ \text{once in } p \end{array} \right\}$$

$$[\![\mathsf{shortest}\ \xi]\!]_G = \left\{ (p, \mu) \in [\![\xi]\!]_G \;\middle|\; \mathsf{len}(p) = \min \left\{ \mathsf{len}(p') \;\middle|\; \begin{array}{l} (p', \mu') \in [\![\xi]\!]_G \\ \mathsf{src}(p') = \mathsf{src}(p) \\ \mathsf{tgt}(p') = \mathsf{tgt}(p) \end{array} \right\} \right\}$$

where ξ is ψ, trail ψ or simple ψ, for some pattern ψ. We then define:

$$[\![x = \rho\psi]\!]_G = \{(p, \mu \cup \{x \mapsto p\}) \mid (p, \mu) \in [\![\rho\ \psi]\!]_G\}$$
$$[\![Q_1, Q_2]\!]_G = \left\{ (\bar{p}_1 \times \bar{p}_2, \mu_1 \cup \mu_2) \;\middle|\; \begin{array}{l} (\bar{p}_i, \mu_i) \in [\![Q_i]\!]_G \text{ for } i = 1, 2 \\ \mu_1 \text{ and } \mu_2 \text{ unify} \end{array} \right\}$$

Here, $\bar{p}_1 = (p_1^1, p_1^2, \ldots, p_1^k)$ and $\bar{p}_2 = (p_2^1, p_2^2, \ldots, p_2^l)$ are tuples of paths, and $\bar{p}_1 \times \bar{p}_2$ stands for $(p_1^1, p_1^2, \ldots, p_1^k, p_2^1, p_2^2, \ldots, p_2^l)$. Note that $\bar{p}_i$ is a single path when Q_i does not contain the join operator. Moreover, like for concatenation, the typing system guarantees that Q_1 and Q_2 are only joined over singleton variables, but not over path, group, or conditional variables.

In the results below we assume the third approach to the definition of **collect** as subsuming the other two. One may verify, by routine inspection, that the semantics is consistent with the typing system.

Proposition 4.3.3

For every well-typed expression ξ and every $(\bar{p}, \mu)$ in $[\![\xi]\!]_G$, all paths in $\bar{p}$ belong to $\mathrm{Paths}(G)$ and μ conforms to $\mathrm{sch}(\xi)$.

Even though the set $\mathrm{Paths}(G)$ may be infinite, syntactic restrictions ensure finiteness of output.

Theorem 4.3.4

$[\![Q]\!]_G$ is finite for each query Q and graph G.

Proof. We will treat only the case when Q is $\rho\psi$; other cases follow from this case or are straightforward.

Let us first show that the set $P = \{\, p \mid \exists \mu\ (p, \mu) \in [\![Q]\!]_G \,\}$ is finite. If ρ is one of trail, simple, shortest trail or shortest simple, the claim holds since there are finitely many trails and simple paths in a graph. The last case, that is $\rho =$ shortest, follows from the fact that for all nodes s and t the set

$$P_{(s,t)} = \{\, p \in P \mid p \text{ starts in } s \text{ and ends in } t \,\}$$

is finite. Indeed, all paths in $P_{(s,t)}$ have the same length, and there are finitely many paths of a given length in a graph.

To finish this proof, we now show the following lemma stating that for each path $p \in P$, there are finitely many μ such that $(p, \mu) \in [\![Q]\!]_G$.

Lemma 4.3.5

Let G be a graph, ψ be a well-typed pattern. For every path p of G, there are finitely many μ such that $(p, \mu) \in [\![\psi]\!]_G$.

By induction over the structure of ψ. The lemma is obviously true for all base cases, and all inductive cases are easy except for a repeated pattern of the form $\psi^{h..\infty}$, for some h.

$$[\![\psi^{h..\infty}]\!]_G = \bigcup_{i=h}^{\infty} [\![\psi]\!]_G^i \,.$$

The proof amounts to showing that there exists a bound B such that

$$\forall n > B,\ \forall \mu, (p,\mu) \in [\![\psi]\!]^n_G \quad (p,\mu) \in [\![\psi]\!]^{n-1}_G \,. \tag{4.4}$$

We fix $B = (L+1)(M+1)$ where $L = \mathsf{len}(p)$ and $M = \max\{\, \mathrm{card}\{\, \mu \mid (\mathrm{path}(u),\mu) \in [\![\psi]\!]_G \,\} \mid u \text{ is a node in } p \,\}$. Note that M is well defined since by IH, $\{\, \mu \mid (\mathrm{path}(u),\mu) \in [\![\psi]\!]_G \,\}$ is finite for every node u.

Let n and μ be as in (4.4). Since $(p,\mu) \in [\![\psi]\!]^n_G$, there are paths $p_1, \dots, p_n$ and bindings $\mu_1, \dots, \mu_n$ such that $p = p_1 p_2 \cdots p_n$ and $\mu = \mathbf{collect}((p_1,\mu_1), \dots, (p_k,\mu_n))$.

We let ℓ and the sequences $(i_k)_{1\le k<\ell}$ and $(\mu'_k)_{1\le k<\ell}$ be defined as on page 81. Note that ℓ is at most $2L+1$; that the number of paths ψ'_k such that $\mathsf{len}(\psi'_k) = 0$ is at most $L+1$; and that the number of paths ψ'_k such that $\mathsf{len}(\psi'_k) \neq 0$ is at most L, in which case $(i_{k+1} - i_k) = 1$. Since $n > (L+1)(M+1)$, it follows that

$$\sum_{\substack{k \\ 1\le k<\ell \\ \mathsf{len}(p'_k)=0}} (i_{k+1} - i_k) = n - \sum_{\substack{k \\ 1\le k<\ell \\ \mathsf{len}(p'_k)\neq 0}} (i_{k+1} - i_k) > (L+1)(M+1) - L$$
$$> (L+1)M + 1$$

hence there exists k such that $\mathsf{len}(p'_k) = 0$ and $(i_{k+1} - i_k) > M$. From the definition of M, it holds $M \ge \mathrm{card}\left\{\, \mu \;\middle|\; (p'_k,\mu) \in [\![\psi]\!]_G \,\right\}$, hence by the pigeon-hole principle, there are i, j such that $i_k \le i < j < i_{k+1}$ and $\mu_i = \mu_j$. It follows that

$$\bigcup_{\substack{x \\ i_k \le x < i_{k+1}}} \mu_x = \bigcup_{\substack{x \\ i_k \le x < i_{k+1} \\ x \neq j}} \mu_x \,.$$

Note that the left part above is the definition of μ'_k, hence $(p,\mu) \in [\![\psi]\!]^{n-1}_G$ as

$$\mathbf{collect}((p_1,\mu_1), \dots, (p_n,\mu_n)) = \mathbf{collect}((p_1,\mu_1), \dots, (p_{j-1},\mu_{j-1}), (p_{j+1},\mu_{j+1}), \dots, (p_n,\mu_n)) \qquad \square$$

4.4 Expressivity and Complexity

Expressive power. First, we look at the expressive power of GPC. For this, we will compare GPC with main graph query languages considered in the research literature. Specifically, we compare GPC with regular path queries (RPQs) [6, 29], and their two-way extension, 2RPQs [100], C2RPQs, which close 2RPQs under conjunctions [8, 100], and their unions, UC2RPQs [8]. An interesting class is also that of nested regular expressions (NREs) [14], where along a path conforming to a regular language, we can test if there is an outgoing path conforming to another regular expression, as in PDL, or XPath. Finally, we consider the class of regular queries (RQs) [57], which subsumes all the aforementioned classes as explained in Chapter 2.

In order to compare with the aforementioned languages, we consider a simple extension of GPC with projection and union, reflecting the fact that the pattern matching mechanism we formalize will be a sublanguage of a fully-fledged query language like GQL or SQL/PGQ. A GPC+

[6]: Cruz et al. (1987), "A Graphical Query Language Supporting Recursion"
[29]: Mendelzon et al. (1995), "Finding Regular Simple Paths in Graph Databases"
[100]: Calvanese et al. (2000), "Containment of Conjunctive Regular Path Queries with Inverse"
[8]: Consens et al. (1990), "GraphLog: a Visual Formalism for Real Life Recursion"
[100]: Calvanese et al. (2000), "Containment of Conjunctive Regular Path Queries with Inverse"
[8]: Consens et al. (1990), "GraphLog: a Visual Formalism for Real Life Recursion"
[14]: Barceló et al. (2012), "Relative Expressiveness of Nested Regular Expressions"
[57]: Reutter et al. (2017), "Regular Queries on Graph Databases"

query is a set of rules

$$Ans(\bar{x}) :- Q_1; \quad Ans(\bar{x}) :- Q_2; \quad \dots \quad Ans(\bar{x}) :- Q_k$$

where Q_i is a GPC query such that $\bar{x} \subseteq \mathrm{var}(Q_i)$ for all i. The *semantics* of such a on graph G is

$$[\![Q_1]\!]_G^{\bar{x}} \cup [\![Q_2]\!]_G^{\bar{x}} \cup \cdots \cup [\![Q_k]\!]_G^{\bar{x}}$$

where $[\![Q_i]\!]_G^{\bar{x}} = \{\, \mu(\bar{x}) \mid \exists p\,(p, \mu) \in [\![Q_i]\!]_G \}$ for all i. Notice that in our definition we allow unions only at the top level in order to combine results of queries whose arity is higher than binary. Binary unions are already covered at the level of GPC patterns, and can be arbitrarily nested inside iterations.

Theorem 4.4.1

GPC+ can express all of the following:

- Unions of Conjunctive Two-way Regular Path Queries (UC2RPQs);
- Nested Regular Expressions (NREs);
- Regular Queries.

Proof. For 2RPQs, note that these are explicitly present in the syntax of GPC patterns, and projecting on the endpoints gives us an equivalent expression. C2RPQs and their unions are then handled by the conjunction of GPC queries, and unions in GPC+, respectively. The case of NREs is a bit more interesting, and it contains the blueprint for regular queries. To illustrate the main ideas, consider the nested regular expression $(a[b^+]c)^+$, which looks for paths of the form $(ac)^n$, where after traversing an a, we also check the existence of a nonempty path labelled with bs. An equivalent GPC+ query is $Ans(x, y) :- Q$ where Q is the GPC query

$$\text{shortest } (x)[\xrightarrow{:a} (z) \xrightarrow{:b}{}^{1..\infty} () \xleftarrow{}{}^{1..\infty} (z) \xrightarrow{:c}]^{1..\infty} (y)\,.$$

Basically, we introduce a fresh variable z which binds to the node from which we need to find a nonempty b-labelled path to an anonymous node, and then we return to the same node, thus allowing us to encode an answer inside of a single path. Since we only care about the endpoints in all of these query classes, the restrictor shortest is enough. This idea is then applied inductively in order to capture regular queries.

Because regular queries work in a simpler graph model in which there are no properties and undirected edges, we will need neither conditioned patterns nor undirected edge patterns. In fact, that data model also does not include node labels, but it is straightforward to include them, which we do.

Consider a Datalog program defining a regular query. We first rewrite the program in such a way that that all user predicates except for the answer predicate *Ans* are binary and are defined with rules whose bodies are connected. Towards this goal, we first eliminate all occurrences of user defined predicates in non transitive atoms, by substituting their definitions exhaustively. User predicates not used in the body of any rule should be also exhaustively removed. Now, each remaining non-answer user predicate is only used in transitive atoms and it remains to elim-

inate rules whose bodies are disconnected. Consider such a predicate that has among its defining rules a disconnected rule

$$P(x_1, x_2) :- P_1(y_1, z_1), P_2(y_2, z_2), \dots, P_k(y_k, z_k)$$

where $y_1, \dots y_k$ and $z_1, \dots z_k$ are variables from the set $\{x_1, \dots, x_m\}$ for some $m \geq 2$. Note that components that are disconnected from both x_1 and x_2 are essentially global Boolean side conditions: if we want to use the rule, we must check that they can be matched somewhere, but if we use the rule multiple times, there is no need to check it again. We shall now consider two cases, depending on whether x_1 and x_2 are in the same component or not.

Suppose first that x_1 and x_2 are in different connected components of the body of the rule above. Let us remove this rule from the program (keeping the other rules for P) and add a new rule

$$\dot{P}(x_1, x_2) :- P_1(y_1, z_1), P_2(y_2, z_2), \dots, P_k(y_k, z_k)$$

where $\dot{P}$ is a fresh predicate.

If we now replace each occurrence of $P^+(x, y)$ with

$$P^+(x, x'), \dot{P}(x', y'), P^+(y', y) \quad \text{or} \quad \dot{P}(x, y'), P^+(y', y) \quad \text{or}$$
$$P^+(x, x'), \dot{P}(x', y) \quad \text{or} \quad \dot{P}(x, y) \quad \text{or} \quad P^+(x, y)$$

and then eliminate $\dot{P}$ by substituting its definition, we obtain an equivalent program. The reason why it is equivalent is that when traversing the graph with P, there is no need to use the disconnected rule more than once. Indeed, whenever we have a sequence of nodes $u_1, u_2, \dots, u_t$ such that $P(u_i, u_{i+1})$ holds for all $i < t$ under the original definition of P, we also have that $P(u_i, u_{i+1})$ or $\dot{P}(u_i, u_{i+1})$ for all $i < t$ under the new definition of P. If the latter holds for more than one value of i, let $i_{\min}$ be the minimal and $i_{\max}$ the maximal of those values. We have $i_{\min} < i_{\max}$. It follows that $P(u_{i_{\min}}, u_{i_{\max}+1})$ holds and we can remove $u_{i_{\min}+1}, \dots, u_{i_{\max}}$ from the sequence. Traverses that use the disconnected rule at most once are captured by the modified program.

The remaining possibility is that x_1 and x_2 are in the same connected component of the body of the rule above. This time we also eliminate the disconnected rule from the definition of P (keeping the other rules for P), but additionally we define two new predicates, $\dot{P}$ and $\ddot{P}$. The definition of $\dot{P}$ is obtained from the original definition of P by taking all rules for P but replacing P in the head with $\dot{P}$, except for the rule with disconnected components, in which we remove all connected components that contain neither x_1 nor x_2; these components are collected in a single rule defining a fresh predicate $\ddot{P}(z, z)$ where z is an arbitrary variable used in any of these components. Now, each transitive atom $P^+(x, y)$ can be replaced with $P^+(x, y)$ or $\dot{P}^+(x, y), \ddot{P}(z, z)$ where z is a fresh variable, and $\ddot{P}$ can be eliminated by substituting its definition. The purpose of $P^+(x, y)$ is to maintain all derivations of P that don't use the disconnected rule that we eliminated. The replacement with $\dot{P}^+(x, y), \ddot{P}(z, z)$ is correct because when traversing the graph with P it is enough to match the disconnected components of the considered rule once, as this match can be reused whenever the rule is applied. Indeed, consider a sequence of nodes $u_1, u_2, \dots, u_t$ such that $\dot{P}(u_i, u_{i+1})$ for all $i < t$ and $\ddot{P}(u)$ for some node u. If the modified rule is never used, we have $P(u_i, u_{i+1})$ for all $i < t$ under the old definition of P. If

the modified rule is used, then we can extend each match of its body to the remaining connected components of the original rule's body by using the match of $\ddot{P}$ at node u.

This way we can eliminate all disconnected rules from the definitions of predicates used in transitive atoms. As the only remaining user predicate is the answer predicate, we are done.

For technical convenience, let us further modify the resulting program so that for each user predicate P,

- either P is defined with a single rule of the from

  $$P(x,x) :- A(x) \quad \text{or} \quad P(x,y) :- a(x,y) \quad \text{or} \quad P(x,y) :- R^+(x,y)$$

  where A is a node label, a an edge label, and R a user predicate,
- or P is defined by rules whose bodies are conjunctions of binary user predicates (without transitive closure).

Note that this can be done without introducing rules with disconnected bodies, so it does not break the first stage of the preprocessing. We are now ready to construct an equivalent GPC+ query.

First, for each binary non-answer user predicate P in the program, we construct by structural induction a GPC pattern ψ_P such that, for each graph G, we have that $P(u,u')$ holds in G iff $(u,u') \in [\![(x)\psi_P(y)]\!]_G^{x,y}$ where x and y are fresh variables not used in ψ_P. The base cases are predicates defined with a single rule of the form

$$P(x,x) :- A(x) \quad \text{or} \quad P'(x,y) :- a(x,y)\,;$$

for such predicates we use

$$\psi_P = (:A) \quad \text{and} \quad \psi_{P'} = \xrightarrow{:a}\,.$$

For predicates defined with a single rule of the form

$$P(x,y) :- R^+(x,y)$$

we take the pattern

$$\psi_P = (\psi_R)^{1..\infty}\,.$$

Finally, consider a non-answer predicate P defined with n rules using only binary user predicates (without transitive closure). We shall construct path patterns $\psi_{P,i}$ (using fresh variables) capturing the rules defining P and then let

$$\psi_P = \psi_{P,1} + \psi_{P,2} + \cdots + \psi_{P,n}\,.$$

Consider the ith rule defining P, say

$$P(x_1,x_2) :- P_1(y_1,z_1), P_2(y_2,z_2), \ldots, P_k(y_k,z_k)$$

where $y_1, \ldots y_k$ and $z_1 \ldots z_k$ are variables from the set $\{x_1,\ldots,x_m\}$ for some $m \geq 2$. Let $\psi_{P_1},\ldots,\psi_{P_k}$ be the path patterns obtained inductively for predicates $P_1,\ldots,P_k$, each using fresh variables. We define $\psi_{P,i}$ as

$$\begin{aligned}\psi_{P,i} = (x_1)\,[\rightarrow + \leftarrow]^{0..\infty}\,(y_1)\,\psi_{P_1}\\ (z_1)\,[\rightarrow + \leftarrow]^{0..\infty}\,(y_2)\,\psi_{P_2}\\ (z_2)\,[\rightarrow + \leftarrow]^{0..\infty}\,\dots\\ [\rightarrow + \leftarrow]^{0..\infty}\,(y_k)\,\psi_{P_k}\\ (z_k)\,[\rightarrow + \leftarrow]^{0..\infty}\,(x_2)\,.\end{aligned}$$

Because all rule bodies for non-answer predicates are connected, the body of the analyzed rule will always be matched in a connected fragment of the graph. Hence, the auxiliary patterns $[\rightarrow + \leftarrow]^{0..\infty}$ used to move from variable to variable do not affect the semantics.

With the patterns ψ_P at hand, we can translate the whole regular query. The answer predicate need not be binary and the bodies of the rules defining it need not be connected, but we are at the very top of the Datalog program and we can simply use the join operator, followed by projection and union. Consider a rule defining the answer predicate,

$$Ans(x_1, x_2, \dots, x_l) :- P_1(y_1, z_1), P_2(y_2, z_2), \dots, P_k(y_k, z_k)$$

where $y_1, \dots, y_k$ and $z_1, \dots, z_k$ are variables from the set $\{z_1, \dots, z_m\}$ for some $m \geq l$. We replace this rule with

$$Ans(x_1, x_2) :- \text{shortest}\,(y_1)\psi_{P_1}(z_1), \dots, \text{shortest}\,(y_k)\psi_{P_k}(z_k)$$

where $\psi_{P_1}, \dots, \psi_{P_k}$ are the patterns (over fresh variables) obtained inductively for predicates $P_1, \dots, P_k$. Because we are only interested in the *existence* of paths connecting y_i to z_i, not the paths themselves, we can safely apply the restrictor shortest. The resulting collection of rules constitutes a GPC+ query equivalent to the regular query defined by the considered Datalog program. □

Complexity. When it comes to evaluating graph queries, one is used to dealing with high complexity. For instance, checking whether there is a query answer with a restrictor simple or trail on top is known to be NP-hard in data complexity (see Chapter 2 or [9, 12, 29, 84, 116]), and yet this feature is supported both by the GQL standard [96] and by concrete languages such as Cypher [15]. Accepting such high complexity bounds probably stems from the fact that query answers can be large in case of graph queries.

Indeed, there is no need to use convoluted 3-SAT reductions to have the engine run forever; one may just enumerate all simple paths in the graph with query: simple $\rightarrow^{0..\infty}$.

In this light, we provide some insights on computing answers of GPC queries, i.e., we study the *GPC enumeration problem*:

Problem:	GPC Enumeration
Input:	A property graph G, and a GPC query Q.
Output:	Enumerate all pairs $(p, \mu) \in [\![Q]\!]_G$ without repetitions.

A potential criticism we would like to address is the fact that the path p, witnessing the output binding μ, is also returned each time, which can make query answers larger than necessary. The reason that we study the problem like this, however, is that this is what the GQL standard

[9]: Barceló (2013), "Querying graph databases"
[12]: Wood (2012), "Query languages for graph databases"
[29]: Mendelzon et al. (1995), "Finding Regular Simple Paths in Graph Databases"
[84]: Martens et al. (2020), "A Trichotomy for Regular Trail Queries"
[116]: Bagan et al. (2020), "A trichotomy for regular simple path queries on graphs"
[96]: Deutsch et al. (2022), "Graph Pattern Matching in GQL and SQL/PGQ"
[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

asks for. In our analysis, we will use Turing machines with output tape (in order to enumerate the results), and will bound the size of work tape the machine uses. The main result of this section is the following:

Theorem 4.4.2

The problem GPC Enumeration can be solved by a Turing machine using exponential space (in G and Q). If we consider the query Q to be fixed, then the machine uses only polynomial space.

The basic idea is to enumerate all possible answers (p, μ) in increasing length of p, and check, one by one, whether they should be output. If we consider a single pattern with a restrictor on top, e.g., $Q = \rho\psi$, this approach works as described, and the size of the possible paths (and thus also bindings μ), can be bounded by a size that is exponential in the size of Q and G, and polynomial if we assume Q to be fixed. For each such answer, we can validate whether it should be output in polynomial space. Notice that once a result is output, we can discard it, and move to the next one. Enumeration stops once an appropriate path length has been reached, and the next binding μ is considered. Joins can then be evaluated by nesting this procedure.

By $|\psi|$ we denote the "structural" size of ψ, that is, the number of nodes in its parse tree plus the number of bits needed to represent numbers n and m in subexpressions of the form $\psi^{n..m}$. For a path p, we denote by $|p|$ the total number of node and edge ids in p. For a binding μ, we denote by $|\mu|$ the total length of paths occurring in μ plus the number of occurrences of variables in μ.

Lemma 4.4.3

Let $\rho\psi$ be a pattern with a restrictor on top. For a property graph G, if $(p, \mu) \in [\![\rho\psi]\!]_G$, then

(a) $\mathsf{len}(p) \leq |N|$, whenever ρ is simple;
(b) $\mathsf{len}(p) \leq |E_d| + |E_u|$, whenever ρ is trail;
(c) $\mathsf{len}(p) \leq (|N| + |E_d| + |E_u|) \times 2^{|\psi|}$, whenever ρ is shortest.

Proof. When ρ is simple, no node can be repeated, and thus $|p| \leq |N|$. Similarly, when ρ is trail, the path length is bounded by the total number of edges in the graph; namely, $|E_d| + |E_u|$.

The case when ρ is shortest can be obtained using standard automata-theoretic techniques. The challenge is how to deal with repetitions of the form $\psi^{n..m}$, where n and m are binary numbers. By unraveling the lower bounds n into concatenations, such expressions can be turned into expressions of size 2^n that only have repetitions of the form $\psi^{0..\infty}$. (Notice that doing so requires special care of group variables, but this does not influence the length of p.) The bound $(|N| + |E_d| + |E_u|) \times 2^{|\psi|}$ then follows from a compilation of the expression into a finite automaton and the fact that a shortest path in the product of this automaton with the graph is also a shortest path in the graph itself. Given that the argument here is quite standard in the research literature (see Chapter 2 or [9, 74, 117, 118]), we do not develop a fully formal automaton model for our queries. □

[9]: Barceló (2013), "Querying graph databases"
[74]: Mendelzon et al. (1989), "Finding Regular Simple Paths in Graph Databases"
[117]: Gelade et al. (2009), "Optimizing Schema Languages for XML: Numerical Constraints and Interleaving"
[118]: Losemann et al. (2013), "The complexity of regular expressions and property paths in SPARQL"

Lemma 4.4.4

Let G be a graph and ψ be a well-typed pattern. For every path p of G, the size $|\mu|$ of μ such that $(p, \mu) \in [\![\psi]\!]_G$ is at most $|p| \times (2^{|\psi|+1} - 2)$. Here, $|p|$ denotes the number of occurrences of node and edge ids in p.

Proof. By induction over the structure of ψ. The lemma is obviously true for all base cases, and all inductive cases are easy except for a repeated pattern of the form $\psi^{h..\infty}$, for some h, where we have that

$$[\![\psi^{h..\infty}]\!]_G = \bigcup_{i=h}^{\infty} [\![\psi]\!]_G^i .$$

By definition of **collect**, we have that every $(p, \mu) \in [\![\psi^{h..\infty}]\!]_G$ consists of a list $(p_1, \mu_1), \ldots, (p_n, \mu_n)$, where $n \leq |p|$ and $p_1 \cdots p_n = p$. Furthermore, we have that $\sum_{i=1}^{n} |p_i| \leq 2|p| - 1$. If every μ_i has size at most $(2^{|\psi|+1} - 2)$, then we have that the size of $(p_1, \mu_1), \ldots, (p_n, \mu_n)$ is bounded by

$$\begin{aligned}
\sum_{i=1}^{n} |p_i| + \sum_{i=1}^{n} |\mu_i| &\leq \sum_{i=1}^{n} |p_i| + \sum_{i=1}^{n} |p_i| \times (2^{|\psi|+1} - 2) \\
&\leq \left(\sum_{i=1}^{n} |p_i| \right) (1 + 2^{|\psi|+1} - 2) \\
&\leq 2|p| \times (2^{|\psi|+1} - 1) \\
&\leq |p| \times (2^{|\psi|+2} - 2) \\
&\leq |p| \times (2^{|\psi^{h..\infty}|+1} - 2)
\end{aligned}$$

□

Lemma 4.4.5

Let ψ be a pattern without variables and G a property graph. Given a path p, we can compute the set of $(p', \mu) \in [\![\psi]\!]_G$ such that p' is a subpath of p in time polynomial in the size of ψ, G, and p.

Proof. Let us denote by $[\![\psi]\!]_G^p$ the set of $(p', \mu) \in [\![\psi]\!]_G$ such that p' is a subpath of p. If ψ does not have variables, then notice that all results $(p', \mu) \in [\![\psi]\!]_G$ have $\mu = \mu_\varnothing$. Assume that $p = \mathsf{path}(u_0, e_0, u_1, \ldots, e_n, u_n)$. We prove by structural induction on ψ that we can compute in polynomial time the set $\mathsf{Pairs}(') := \{(i, j) \mid (\mathsf{path}(u_i, e_i, \ldots, u_j), \mu_\varnothing) \in [\![\psi']\!]_G^p\}$ for all subexpressions ψ' of ψ.

The base cases can clearly be computed in polynomial time. If ψ' is of the form $\psi_1\psi_2$, then $\mathsf{Pairs}(\psi')$ is the natural join of $\mathsf{Pairs}(\psi_1)$ with $\mathsf{Pairs}(\psi_2)$. If ψ' is of the form $\psi_1 + \psi_2$, then $\mathsf{Pairs}(\psi')$ is the union of $\mathsf{Pairs}(\psi_1)$ with $\mathsf{Pairs}(\psi_2)$. And if $\psi' = \psi_1^{n..m}$, then we can compute $\mathsf{Pairs}(\psi_1)^{n..m}$ in polynomial time using iterative squaring. □

Lemma 4.4.6

Let ψ be a fixed pattern and G a property graph. Given a path p, we can enumerate the set of $(p', \mu) \in [\![\psi]\!]_G$ such that p' is a subpath of p in space polynomial in the size of G and p.

Proof sketch. Let us denote by $[\![\psi]\!]_G^p$ the set of $(p', \mu) \in [\![\psi]\!]_G$ such that p' is a subpath of p. We prove the lemma by structural induction on ψ. If ψ does not have variables, then Lemma 4.4.5 tells us that we can compute $[\![\psi]\!]_G^p$ in polynomial time, even when ψ is not fixed. So we assume that ψ has at least one variable.

The base cases are clear. For the inductive cases, we enumerate the relevant results (p', μ) of the subexpressions in polynomial space inductively and compose them to form a result for the entire expression. For concatenation, union, and conditioned patterns, we can do this by just following the inductive definition of $[\![\psi_1\psi_2]\!]_G$, $[\![\psi_1 + \psi_2]\!]_G$, and $[\![\psi_{\langle\theta\rangle}]\!]_G$. It remains to discuss repeated patterns, i.e., $\psi = \psi_1^{n..m}$. Since ψ is constant, it suffices to consider the case $\psi = \psi_1^{n..n}$ and the case $\psi = \psi_1^{0..\infty}$. We first consider the case $\psi_1^{n..n}$. By the induction hypothesis, we can assume that we can enumerate $[\![\psi_1]\!]_G^p$ using polynomial space. Since ψ_1 has at least one variable (we already dealt with the case without variables), we need to enumerate the set (p, μ) such that there exist $(p_1, \mu_1), \dots, (p_n, \mu_n) \in [\![\psi_1]\!]_G^p$ where $p = p_1 \cdots p_n$ and $\mu = \mathbf{collect}[(p_1, \mu_1), \dots, (p_n, \mu_n)]$. Since n is a constant, we only need to keep constantly many (p_i, μ_i) in memory, so we can proceed in polynomial space by following the definition of **collect**. We now consider the case $\psi = \psi_1^{0..\infty}$. In this case, we again follow the definition of **collect**. Even though the number n in a sequence $(p_1, \mu_1), \dots, (p_n, \mu_n)$ with $(p_i, \mu_i) \in [\![\psi_i]\!]_G$ and $p = p_1 \cdots p_n$ can be arbitrarily large, **collect** always collapses those subpaths of length zero together. As such, only polynomially many different subpaths of p need to be considered. For a given (p, μ), we therefore systematically enumerate all sequences $(p_1, \mu_1), \dots, (p_n, \mu_n)$ with $p = p_1 \cdots p_n$ and $(p_i, \mu_i) \in [\![\psi_1]\!]_G^p$. Since by Lemma 4.4.4, the total size of the bindings μ_i is also polynomial in $|p|$ if ψ is fixed, we can enumerate everything using polynomial space. □

We are now ready to prove Theorem 4.4.2.

Proof. We first consider data complexity; namely, we assume Q to be fixed. First we consider the query $Q = \rho\psi$, which is a single pattern with a restrictor on top. The Turing machine in this case operates by trying to enumerate all possible paths p and bindings μ such that $(p, \mu) \in [\![Q]\!]_G$. W.l.o.g., we can assume an order on node and edge ids.

Assume first that the query does not use the restrictor shortest, but is of the form $\rho\psi$, with ρ either simple or trail. The machine starts enumerating all possible paths p one by one in radix order, that is, in increasing length, and then by the ordering we assume on node and edge ids. By Lemma 4.4.3, we know that we only need to consider paths of polynomial length. For each such path, we enumerate all possible μ that bind variables of ψ to elements in p one by one, which can be done in polynomial space in the size of G and p according to Lemma 4.4.6. For a given path p, we can thus enumerate all possible answers (p, μ) using polynomial space.

Assume now that the query uses the restrictor shortest. In this case, we have to modify our machine slightly. Notice that in this case we might have shortest either on top of trail or simple, or just on its own. In all of these cases we proceed in the same manner: we enumerate the results of the query below shortest, and the first time we encounter a result, we write it down to the output tape, and store it to an additional tape that this machine has access to. When another path is then considered, in case that it could be the answer, its length is compared to the stored path, and eliminated in case its length is longer. The path enumeration procedure can be halted upon reaching length $|G| \times 2^{|\psi|}$ (Lemma 4.4.3).

In a query that uses joins, the procedure can be thought as being nested for each part of the join. Given that the query Q is fixed, the amount of work space used by the machine remains polynomial.

Moving to combined complexity, we can observe that the space used thus far is exponential at worst. In fact, the main complication arises with iterations of the form $\psi^{n..m}$, where n and m are concrete numbers, given that their representation (e.g. in binary) is exponentially more succinct than their magnitude, which dictates the number of graph elements that might need to be bound in group variables. □

Theorem 4.4.7

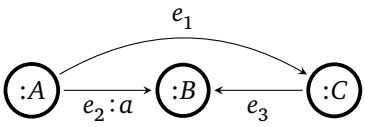
The problem ENUMERATE ANSWERS cannot be solved by a Turing machine using polynomial amount of space (in G and Q).

Proof. Queries of the form $x = \text{shortest}\ (\ )\ \rightarrow^{k..k}\ (\ )$ produce 2^k different paths on the graph G with nodes u, v, with a-edges from u to v and back; and b-edges from u to v and back. Since k is represented in binary, this number of paths is $\Theta(2^{2^n})$, where n is the input size. A polynomial space algorithm cannot represent the required 2^{2^n} configurations required to enumerate these paths without repetitions. □

4.5 Looking ahead

In this section, we discuss possible extensions of GPC that would reflect additional features envisioned in GQL and SQL/PGQ. In doing so, we also provide two examples of how theoretical research has directly influenced the drafts of the GQL and SQL/PGQ standards as they were being written.

Placement of restrictors We imposed strict requirements for placing restrictors: optional shortest followed by optional trail or simple, with at least one of the three present to ensure that the number of returned paths is finite. It is natural to wonder whether restrictors could be mixed arbitrarily, by allowing patterns $\rho\,\psi$ where ρ is one of shortest, trail, and simple. In fact, this was an earlier proposal in the GQL and PGQ drafts, which was then significantly modified. To see why, consider the following graph



(where e_1, e_2, and e_3 are edge ids) and the pattern

$$\text{trail}\Big[\big[\text{shortest } (:A) \xrightarrow{x}{}^{0..\infty} (:B)\big](:B) \xleftarrow{y:a}{}^{*} (:A)\Big] .$$

Matching the subpattern outside shortest produces the binding of y to the edge e_2. In the GQL rationale, shortest should restrict query answers in the sense that, out of all the answers to the query, it chooses the one with the shortest witness. If we follow this rationale, then, to keep the entire match a trail, the group variable x must be bound the list $[e_1, e_3]$. Therefore, counter-intuitively, a shortest match occurring under the scope of trail produces a path that is not shortest between two nodes.

As a result, GQL pattern matching now disallows arbitrary mixing of restrictors. At the same time, it is slightly more permissive than the version presented here: shortest must appear at the top of a pattern, but trail and simple can be mixed freely. Adding this feature is a possible extension of GPC.

Aggregation As one navigates along a path in a graph, aggregation is a natural feature for computing derived quantities, such as path length. For instance, with $(:A) \xrightarrow{x}{}^{0..\infty} (:B)$ looking for paths between A and B, one could return the total length $\sum x.length$ of each matched path. However, adding aggregation is problematic. To see why, consider the simple aggregate $\sharp(x)$ for a group variable x, which counts the number of bindings of that variable. Now assume we extend the language with *arithmetic conditions* of the form $t_1 = t_2$, where t_1 and t_2 are terms built from values $y.k$ and $\sharp(x)$ by means of addition "+" and multiplication "·". These already pack huge expressive power:

Proposition 4.5.1

The data complexity of GPC with arithmetic conditions is undecidable.

Proof. We first show the undecidability of combined complexity and then extend the construction to show data complexity as well. We do it by reduction from the Diophantine equation (or Hilbert's 10th) problem, namely whether a multivariate polynomial $f(x_1, \ldots, x_m)$ with integer coefficients has a solution, i.e., numbers $v_1, \ldots, v_m \in \mathbb{N}$ so that $f(v_1, \ldots, v_m) = 0$ (the problem is more commonly stated about finding solutions in $\mathbb{Z}$ but clearly this version is undecidable as well). Specifically we shall use a version where $m \leq 58$ and degrees of all the monomials in f are bounded by 4; this is already known to be undecidable [25].

Create a graph G_0 with nodes $n_1, \cdots, n_{58}$, an A_i-labeled loop on every node n_i for $i \leq 58$, and A-labeled edges from n_i to n_{i+1} for $i < 58$. We assume that n_1 is labeled S and has a value $\delta(n_1, k) = 0$. Consider a

pattern

$$\psi_0 = (u : S) \xrightarrow{x_1:A_1} 0..\infty \; ( ) \xrightarrow{:A} \cdots \xrightarrow{:A} ( ) \xrightarrow{x_{58}:A_{58}} 0..\infty \; ( )$$

This pattern navigates along the A-edges, potentially looping several times over each A_i-labeled edge. Now consider $\psi_{0\langle f(\sharp(x_1),\cdots,\sharp(x_{58}))=u.k\rangle}$. Then G_0 has a match for this pattern iff f has a solution $v_1, \ldots, v_m \in \mathbb{N}$. Indeed, the match will loop v_1 times over n_1, then move to n_2, loop v_2 times, and so on.

This shows undecidability of combined complexity as the polynomial f is part of the query. We now extend the construction to data complexity. Let α range over the set M of 5^{58} bindings from $\{1, \cdots, 58\}$ to $\{0, \cdots, 5\}$. By m_α we mean the monomial

$$x_1^{\alpha(1)} \cdot \ldots \cdot x_{58}^{\alpha(58)}$$

and we then assume without loss of generality that

$$f = \sum_{\alpha \in M} c_\alpha \cdot m_\alpha$$

where $c_\alpha \in \mathbb{Z}$. Note that the degree of some of the monomials may be higher than 4 but then the problem of the existence of integer solutions of f is still undecidable, and furthermore the number of monomials is fixed. Also notice that since each degree is bounded by 4, every monomial can be constructed as an arithmetic expression, e.g., we could write $u \cdot u \cdot w \cdot w \cdot w \cdot w$ instead of $u^2 w^4$.

We next extend G_0 to G as follows. We enumerate the set M as $\alpha_1, \cdots, \alpha_{|M|}$ and in G_0 continue the chain of A-labeled edges for another M nodes which will all have a property *coeff* holding values $c_{\alpha_1}, \cdots, c_{\alpha_{|M|}}$. In addition, these nodes will have loops, labeled $B_1, \cdots, B_{|M|}$.

Then, starting with ψ_0 we proceed to define ψ_i for $1 \leq i \leq M$ inductively as follows:

$$\psi_i = \left[\psi_{i-1} \xrightarrow{:A} ( ) \xrightarrow{y_i:B_i} 0..\infty \; ( )\right]_{\langle \sharp(y_i)=y_i.coeff \cdot m_1(\sharp(x_1),\cdots,\sharp(x_{58}))\rangle}$$

The effect of matching such a pattern is that the number of times $\sharp(y_i)$ the B_i-labeled loop is traversed equals the value of $c_{\alpha_i} \cdot m_{\alpha_i}$ over the values $\sharp(x_1), \cdots, \sharp(x_{58})$. Indeed, as mentioned earlier, monomials m_α are proper arithmetic terms.

Let $\psi = \psi_{|M|}$. Then

$$\psi_{\langle \sharp(y_1)+\cdots+\sharp(y_{|M|})=u.k\rangle}$$

matches iff $f(\sharp(x_1), \cdots, \sharp(x_{58})) = 0$. Since now the pattern is fixed (all the information about f is encoded in G), the data complexity is proved to be undecidable. □

In view of this, the current approach of GQL is to only allow aggregates in the outputs of queries (no operations on them are permitted). But it is a general open direction to understand how to tame and use the power of aggregates in a graph language.

Bag semantics Relational calculus is interpreted under set semantics, but SQL uses bag semantics, and so does GQL. In the basic version of GPC we opted for set semantics, following the relational calculus precedent, but it is necessary to study bag semantics as an extension of GPC.

Nulls and bound conditional variables We have left out the treatment of nulls, assuming that conditions involving non-applicable nulls — i.e., values $\delta(x,k)$ where the property k is not defined for x — evaluate to *false*. Following SQL (and the current GQL proposals) they would instead evaluate to *unknown*, leading to many known issues [119]. In addition, one could expand the language with a predicate that checks whether a conditional variable is bound, as done, in fact, in GQL.

[119]: Console et al. (2020), "Coping with Incomplete Data: Recent Advances"

Scoping of variables Consider the pattern $(x) \to \left[(y) \to (z)\right]_{\langle\theta\rangle}$, where θ is $x.k = y.k + z.k$; note, in particular, the non-local occurrence of $x.k$ in the condition. Should this be allowed? At first it seems innocent (we bind x before evaluating the rest of the pattern), but in a pattern like $(x) \to \left[(y) \to (z)\right]_{\langle x.k+u.k=y.k+z.k\rangle} \to (u)$ we need to evaluate the edge from y to z first. Moreover, if we have repetitions, the evaluation procedure becomes much less clear. Despite this, GQL plans to offer such kind of features.

Label expressions GQL will offer complex label expressions [96], and these too can be added to the calculus as an extension.

[96]: Deutsch et al. (2022), "Graph Pattern Matching in GQL and SQL/PGQ"

Enhancing conditions We assumed that data values come from one countable infinite set of constants, very much in line with the standard presentations of first-order logic. In reality, of course, data values are typed, and such typing must be taken into account (at the very least, for the study of aggregation, to distinguish numerical properties). Furthermore, one could permit explicit equalities between singleton variables in conditions (currently, such variables implicitly join when they are repeated in a pattern).

References

[6] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. "A Graphical Query Language Supporting Recursion". In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. Ed. by Umeshwar Dayal and Irving L. Traiger. ACM Press, 1987, pp. 323–330. DOI: 10.1145/38713.38749.

[8] Mariano P. Consens and Alberto O. Mendelzon. "GraphLog: a Visual Formalism for Real Life Recursion". In: *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. ACM Press, 1990, pp. 404–416. DOI: 10.1145/298514.298591.

[9] Pablo Barceló. "Querying graph databases". In: *Principles of Database Systems (PODS)*. 2013, pp. 175–188.

[11] Pablo Barceló et al. "Expressive languages for path queries over graph-structured data". In: *ACM Trans. Database Syst.* 37.4 (2012), 31:1–31:46.

[12] Peter T. Wood. "Query languages for graph databases". In: *SIGMOD Record* 41.1 (2012), pp. 50–60.

[14] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. "Relative Expressiveness of Nested Regular Expressions". In: *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27-30, 2012*. 2012, pp. 180–195.

[15] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

[18] Renzo Angles et al. "G-CORE: A Core for Future Graph Query Languages". In: *SIGMOD*. 2018, pp. 1421–1432.

[25] James Jones. "Undecidable diophantine equations". In: *Bulletin of the American Mathematical Society* 3.2 (1980), pp. 859–862.

[29] Alberto O. Mendelzon and Peter T. Wood. "Finding Regular Simple Paths in Graph Databases". In: *SIAM J. Comput.* 24.6 (1995), pp. 1235–1258.

[57] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. "Regular Queries on Graph Databases". In: *Theory Comput. Syst.* 61.1 (2017), pp. 31–83. DOI: 10.1007/s00224-016-9676-2.

[60] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. "Querying Graphs with Data". In: *Journal of the ACM* 63.2 (2016), 14:1–14:53.

[74] Alberto O. Mendelzon and Peter T. Wood. "Finding Regular Simple Paths in Graph Databases". In: *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*. 1989, pp. 185–193.

[84] Wim Martens, Matthias Niewerth, and Tina Trautner. "A Trichotomy for Regular Trail Queries". In: *International Symposium on Theoretical Aspects of Computer Science, (STACS)*. 2020, 7:1–7:16.

[96] Alin Deutsch et al. "Graph Pattern Matching in GQL and SQL/PGQ". In: *SIGMOD*. ACM, 2022, pp. 1–12.

[100] Diego Calvanese et al. "Containment of Conjunctive Regular Path Queries with Inverse". In: *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000*. 2000, pp. 176–185.

[114] Renzo Angles et al. "Foundations of Modern Query Languages for Graph Databases". In: *ACM Comput. Surv.* 50.5 (2017), 68:1–68:40.

[115] Mikolaj Bojanczyk et al. "Two-variable logic on data trees and XML reasoning". In: *PODS*. ACM, 2006, pp. 10–19. DOI: 10.1145/1142351.1142354.

[116] Guillaume Bagan, Angela Bonifati, and Benoît Groz. "A trichotomy for regular simple path queries on graphs". In: *J. Comput. Syst. Sci.* 108 (2020), pp. 29–48.

[117] Wouter Gelade, Wim Martens, and Frank Neven. "Optimizing Schema Languages for XML: Numerical Constraints and Interleaving". In: *SIAM J. Comput.* 38.5 (2009), pp. 2021–2043.

[118] Katja Losemann and Wim Martens. "The complexity of regular expressions and property paths in SPARQL". In: *ACM Trans. Database Syst.* 38.4 (2013), 24:1–24:39.

[119] Marco Console et al. "Coping with Incomplete Data: Recent Advances". In: *PODS*. ACM, 2020, pp. 33–47. DOI: 10.1145/3375395.3387970.

5 Putting together Pattern Matching and Relational Algebra

This chapter is joint work with A. Gheebrant, L. Libkin and L. Peterfreund and was submitted under the title "Limitations of Modern Graph Query Languages" at PODS 2025.

Database theory thrives when it comes up with mathematically concise abstractions of real life artifacts that can be analyzed, with the results then used in a feedback loop to influence those artifacts. A canonical example of this is the study of database query languages: we often take first-order logic (FO) and relational algebra (RA) as very rough abstractions of SQL. While real-life SQL goes way beyond these abstractions, its core features are indeed based on FO and RA. The latter have very concise mathematical definitions and are thus amenable to deep mathematical analysis. A typical example of such an analysis is inexpressibility of counting properties and recursion [19]; as a result, these were added to SQL and many other database languages.

[19]: Arenas et al. (2022), *Database Theory*

This productive approach has manifested itself in a multitude of scenarios. We know, for example, that no amount of counting power lets one express recursion, nor does recursion help with counting, hence both are necessary as separate additions to SQL [21]. This general philosophy helped us discover important results in database theory with direct practical applications. For example, in query evaluation, abstracting join evaluation as the existence of homomorphisms led to bounds on join sizes [120] and worst-case optimal join algorithms [121]. In other data models, abstracting path languages for XML as temporal logics [122] and their transformation and schema languages as tree transducers and automata [123] had a direct impact on various W3C standards, while an abstraction of SPARQL via RA operations producing a set of mappings [124] led to interesting discoveries about the design of SPARQL [76, 77].

[21]: Libkin (2004), *Elements of Finite Model Theory*

[120]: Atserias et al. (2013), "Size Bounds and Query Plans for Relational Joins"

[121]: Ngo et al. (2018), "Worst-case Optimal Join Algorithms"

[122]: Marx (2006), "Navigation in XML Trees"

[123]: Martens et al. (2006), "Expressiveness and complexity of XML Schema"

[124]: Pérez et al. (2006), "Semantics and Complexity of SPARQL"

[76]: Arenas et al. (2012), "Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard"

[77]: Losemann et al. (2013), "The complexity of regular expressions and property paths in SPARQL"

This brings us to the question: do we have a similarly clean mathematical model for graph databases? While some mathematical abstractions of these languages exist, namely the ones presented in Chapters 3 and 4, they are not yet at the same level as FO and RA, in terms of their simplicity and utility in formally proving results about GQL and SQL/PGQ. The main goal of this Chapter is therefore to provide very concise abstractions capturing main features of these newly standardized languages, and use them to analyze limitations of their expressive power. As is traditional in database theory, such an analysis is done to understand what is missing in their original design, in order to influence future enhancements of the languages.

Regarding the status of research and practice in graph languages, it appears that on the theory side we have a good abstraction. They are based on the the ubiquitous regular path queries (RPQs) [6] and their many derivatives such as CRPQs [8], their extensions with union and inverse [100], extended CRPQs [11], RPQs with data comparisons [60] and many others; see Chapter 2 and surveys [9, 114]. These however make an omission too big to be considered realistic models. The standard industry model is that of labeled property graphs while RPQs and related formalisms operate on labeled graphs, essentially omitting data stored in them. This is not a small omission (for databases, after all), and while some languages considered support for data and were even

[6]: Cruz et al. (1987), "A Graphical Query Language Supporting Recursion"

[8]: Consens et al. (1990), "GraphLog: a Visual Formalism for Real Life Recursion"

[100]: Calvanese et al. (2000), "Containment of Conjunctive Regular Path Queries with Inverse"

[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"

[60]: Libkin et al. (2016), "Querying Graphs with Data"

[9]: Barceló (2013), "Querying graph databases"

[114]: Angles et al. (2017), "Foundations of Modern Query Languages for Graph Databases"

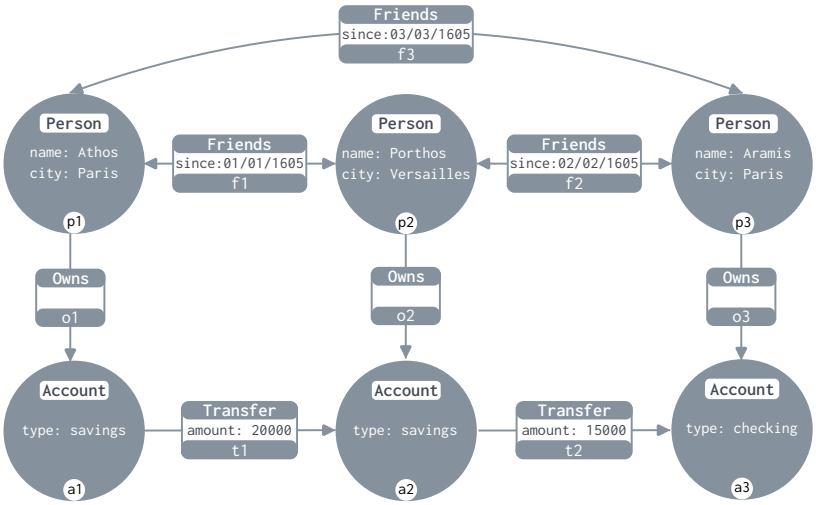


Figure 5.1: A labeled property graph

called academic influences on GQL development [97], they influenced the design, rather than reflected it.

[97]: (2023), *GQL Influence Graph*

In industry, the first modern declarative graph language for property graphs is Cypher [15]. Its workhorse is pattern matching which turns a graph into a table; the remaining operations manipulate that table. With several other languages designed along the same lines, a standardization was proposed, resulting in SQL/PGQ and GQL standards. These share pattern matching [96], but then there is a major diversion.

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

[96]: Deutsch et al. (2022), "Graph Pattern Matching in GQL and SQL/PGQ"

- In SQL/PGQ, a graph is a view defined on a relational database. The result of pattern matching is simply a table in the `FROM` clause of a SQL query that may use other relations in the database.
- GQL on the other hand is oblivious to how a graph is stored, and the table resulting from pattern matching is manipulated by a sequence of operators that modify it, in an imperative style that is referred to as linear composition.

We illustrate GQL and SQL/PGQ capabilities using the graph from Figure 5.1 and a money-laundering query asking to *Find a pair of friends in the same city who transfer money to each other via a common friend who lives elsewhere*.

In GQL, it will be expressed as the following query :

```
MATCH (x)-[:Friends]->(y)-[:Friends]->(z)-[:Friends]->(x),
      (x)-[:Owns]->(acc_x), (y)-[:Owns]->(acc_y),
      (z)-[:Owns]->(acc_z),
      (acc_x)-[t1:Transfer]->(acc_z)-[t2:Transfer]->(acc_y)
FILTER (y.city) <> (x.city) AND (x.city=z.city)
       AND (t2.amount < t1.amount)
RETURN x.name AS name1, y.name AS name2
```

In SQL/PGQ, graphs are a view of a tabular schema. The one from Figure 5.1 can be represented by the following set of tables:
`Acc(a_id,type)` for accounts, `Transfer(t_id,a_from,a_to,amount)` for transfers, `Person(p_id,name,city)` for people, `Owns(a_id,p_id)` for ownership, and `Friend(p_id1,p_id2,since)` for friendships. The property graph view is then defined by a `CREATE` statement, part of which is shown below:

```
CREATE PROPERTY GRAPH Interpol1625 (
  VERTEX TABLES
    Acc KEY (a_id) LABEL Account PROPERTIES (type)
    ....
  EDGE TABLES
    Transfer KEY (t_id)
        SOURCE KEY (a_from) REFERENCES Acc
        DESTINATION KEY (a_to) REFERENCES Acc
        LABEL Transfer PROPERTIES (amount)
    .... )
```

This view defines nodes (vertices) and edges of graphs, specifies endpoints of edges, and defines their labels and properties. We can then query it, using pattern matching to create a subquery:

```
SELECT T.name_x AS name1, T.name_y AS name2
FROM Interpol1625 GRAPH_TABLE (
  MATCH (x)-[:Friends]->(y)-[:Friends]->(z)-[:Friends]->(x),
    (x)-[:Owns]->(acc_x), (y)-[:Owns]->(acc_y),
    (z)-[:Owns]->(acc_z),
    (acc_x)-[t1:Transfer]->(acc_z)-[t2:Transfer]->(acc_y)
  COLUMNS x.city AS city_x, y.city AS city_y, z.city AS city_z,
    x.name AS name_x, y.name AS name_y,
    t1.amount AS amount1, t2.amount AS amount2 )  AS T
WHERE T.city_x=T.city_y AND T.city_x <> T.city_z
    AND T.amount1 > T.amount2
```

Note that in GQL, a sequence of operators can continue *after* the `RETURN` return clause. For example, if we want to find large transfers between the two potential offenders we could simply continue the query with extra clauses:

```
MATCH (u WHERE u.name=name1) -[t:Transfer]->(v WHERE v.name=v2)
FILTER t.amount > 100000
RETURN t.amount AS big_amount
```

This is what is referred to as linear composition: we can simply add clauses to the already existing query which apply new operations to the result of already processed clauses.

In SQL/PGQ, such an operation is also possible, though perhaps a bit more cumbersome as we would need to put the above PGQ query as a subquery in `FROM` and create another subquery for the second match, then join them on `name1` and `name2`.

Related work

While industry is dominated by property graphs (Neo4j, Oracle, Amazon, TigerGraph, etc), much of academic literature still works with the model of labeled graphs and query languages based on RPQs, with some exceptions [60, 125, 126]. These however appeared before the new standards became available, and their analyses of expressiveness and language features do not apply to GQL and SQL/PGQ. The first commercial language for property graphs was Cypher, and it was fully formalized in [15]. As GQL and SQL/PGQ were being developed, a few academic papers appeared: [96] gave an overview of their pattern matching facilities, which was then further analyzed in [102] (and presented in Chapter 4). In [127] (and Chapter 3), a digest of GQL suitable for the research community was presented. While a huge improvement compared to the actual standard from the point of view of clarity, the presentation of [127] replaced 500 pages of the text of the Standard

[60]: Libkin et al. (2016), "Querying Graphs with Data"
[125]: Abriola et al. (2018), "Bisimulations on Data Graphs"
[126]: Sharma et al. (2021), "Practical and comprehensive formalisms for modelling contemporary graph query languages"
[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"
[96]: Deutsch et al. (2022), "Graph Pattern Matching in GQL and SQL/PGQ"
[102]: Francis et al. (2023), "GPC: A Pattern Calculus for Property Graphs"
[127]: Francis et al. (2023), "A Researcher's Digest of GQL"

(notoriously hard to read) by a one-page long definition of the syntax, followed by a four-page long definition of the semantics. It achieved a two orders of magnitude reduction in the size of the definition of the language, but 5 pages is still way too long for "Definition 1".

Another element that is missing in the literature is a proper investigation of linear composition. Initially introduced in Cypher, it was then adopted in a purely relational language PRQL [128] (positioned as a "pipelined" alternative to SQL), and also embraced by GQL.

[128]: PRQL (2024), *Pipelined Relational Query Language*

Contributions

In this Chapter, we first formalize linear-composition relational algebra LCRA (which is a key ingredient of modern graph languages), show that it is equivalent to the usual relational algebra, and explain its origins in graph and relational querying.

We then define a very concise graph pattern calculus that has most of the essential features of GQL and SQL/PGQ pattern matching, and introduce two languages, Core GQL and Core PGQ, capturing the essence of the new standards. The former is captured by LCRA over relations resulting from pattern matching, and the latter by relational algebra over the same relations.

With these mathematical models, we analyze the expressiveness of the languages. We start with the pattern language and show that it cannot express, in a natural way, a very common query that tests if values of particular property in edges increase along a path. We also show that the pattern language is not powerful enough to capture "global" conditions (e.g., all property values on a path are different). We conclude this analysis by formalizing patterns from the original formulation of Cypher, and proving that they fall short of RPQs (a "folklore" result that has never been formally proved due to the lack of a formal model).

We then look at the entire GQL and SQL/PGQ, and show that they fail to capture Datalog queries of NLOGSPACE data complexity. Such a capture would be expected from a language that defines reachability and has full power of first-order logic on top of it, and yet the combination of patterns with querying their outputs renders it impossible.

In our inexpressibility results, we identify key shortcomings of the new query languages for graph databases that prevent us from expressing the desired properties. These concern joins of patterns and compositionality of queries, and could serve as a starting point for enhancing expressiveness in versions 2 of GQL and SQL/PGQ, expected to appear in about 5 years from now.

5.1 Linear Composition Relational Algebra

In this section we formalize the notion of linear composition that underlies Cypher and GQL, and is present independently of them in purely relational languages such as PRQL. We start with the definition of linear composition relational algebra, prove its equivalence with classical RA, and discuss the origins of this approach to query language design.

As in Chapter 2, we assume the named perspective for Relational Algebra, and take relation names from the infinite countable set Rel and

attribute names from the infinite countable set Attr.

Let μ be a tuple and $\boldsymbol{A} \subseteq \mathrm{dom}(\mu)$. We use $\mu \restriction \boldsymbol{A}$ to denote the *restriction* of μ to $\boldsymbol{A}$, that is, the mapping μ' with $\mathrm{dom}(\mu') = \boldsymbol{A}$ and $\mu'(A) ::= \mu(A)$ for every attribute $A \in \boldsymbol{A}$. Let μ_1, μ_2 be tuples. We say that μ_1 and μ_2 are *compatible*, denoted by $\mu_1 \sim \mu_2$, if $\mu_1(A) = \mu_2(A)$ for every $A \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$. For such compatible tuples we define $\mu_1 \bowtie \mu_2$ as the mapping μ with $\mathrm{dom}(\mu) ::= \mathrm{dom}(\mu_1) \cup \mathrm{dom}(\mu_2)$, and $\mu(A) ::= \mu_1(A)$ if $A \in \mathrm{dom}(\mu_1)$ and $\mu(A) ::= \mu_2(A)$ otherwise. If $A \in \mathrm{dom}(\mu)$ then we define $\rho_{A \to B}(\mu)$ as the mapping μ' with $\mathrm{dom}(\mu') = (\mathrm{dom}(\mu) \setminus \{A\}) \cup \{B\}$ where $\mu'(B) ::= \mu(A)$ and $\mu'(A') = \mu(A')$ for every other $A' \in \mathrm{dom}(\mu')$.

5.1.1 Linear Composition Relational Algebra (LCRA)

Linear Composition Relational Algebra (LCRA) underlies the sequential (linear) application of relational operators as seen in Cypher, GQL, and also PRQL. Its expressions over a schema $\mathbf{S}$, denoted by $\mathrm{LCRA}(\mathbf{S})$, are defined as follows:

$$\begin{aligned} \textit{Linear Clause:}\quad \mathsf{L} &::= S \mid \pi_{\boldsymbol{A}} \mid \sigma_\theta \mid \rho_{A \to A'} \mid \mathsf{LL} \mid \{\mathsf{Q}\} \\ \textit{Query:}\quad \mathsf{Q} &::= \mathsf{L} \mid \mathsf{Q} \cap \mathsf{Q} \mid \mathsf{Q} \cup \mathsf{Q} \mid \mathsf{Q} \setminus \mathsf{Q} \end{aligned}$$

where S ranges over $\mathbf{S}$, while $\boldsymbol{A} \subseteq \mathrm{Attr}$, and $A, A' \in \mathrm{Attr}$, and θ is defined as for RA. Unlike for RA, for LCRA the output schema can be determined only dynamically.

The *semantics* $[\![\,]\!]_{\mathscr{D}}$ of LCRA clauses L and queries Q is a mapping from relations into relations. Such relations are known in the context of graph languages Cypher and GQL as driving tables. It is defined as follows:

$$\begin{aligned} [\![S]\!]_{\mathscr{D}}(\mathbf{R}) &::= \mathbf{R} \ltimes \mathscr{D}(S) \\ [\![\pi_{\boldsymbol{A}}]\!]_{\mathscr{D}}(\mathbf{R}) &::= \{\mu \restriction_{\boldsymbol{A} \cap \mathrm{attr}(\mathbf{R})} \mid \mu \in \mathbf{R}\} \\ [\![\sigma_\theta]\!]_{\mathscr{D}}(\mathbf{R}) &::= \{\mu \mid \mu \in \mathbf{R}, \mu \vDash \theta\} \\ [\![\rho_{A \to A'}]\!]_{\mathscr{D}}(\mathbf{R}) &::= \{\rho_{A \to A'}(\mu) \mid \mu \in \mathbf{R}, A' \notin \mathrm{dom}(\mu)\} \\ [\![\mathsf{L}_1 \mathsf{L}_2]\!]_{\mathscr{D}}(\mathbf{R}) &::= [\![\mathsf{L}_2]\!]_{\mathscr{D}}([\![\mathsf{L}_1]\!]_{\mathscr{D}}(\mathbf{R})) \\ [\![\{\mathsf{Q}\}]\!]_{\mathscr{D}}(\mathbf{R}) &::= \mathbf{R} \ltimes [\![\mathsf{Q}]\!]_{\mathscr{D}}(\mathbf{R}) \\ [\![\mathsf{Q}_1 \ \mathrm{op}\ \mathsf{Q}_2]\!]_{\mathscr{D}}(\mathbf{R}) &::= [\![\mathsf{Q}_1]\!]_{\mathscr{D}}(\mathbf{R}) \ \mathrm{op}\ [\![\mathsf{Q}_2]\!]_{\mathscr{D}}(\mathbf{R}), \quad \text{for } \mathrm{op} \in \{\cup, \cap, -\} \end{aligned}$$

Attributes of results of linear clauses and queries are defined (also dynamically) then as follows:

$$\begin{aligned} \mathrm{attr}([\![S]\!]_{\mathscr{D}}(\mathbf{R})) &::= \mathrm{attr}(S) \cup \mathrm{attr}(R) \\ \mathrm{attr}([\![\pi_{\mathbf{A}}]\!]_{\mathscr{D}}(\mathbf{R})) &::= \mathbf{A} \cap \mathrm{attr}(\mathbf{R}) \\ \mathrm{attr}([\![\sigma_\theta]\!]_{\mathscr{D}}(\mathbf{R})) &::= \mathrm{attr}(\mathbf{R}) \\ \mathrm{attr}([\![\rho_{A \to A'}]\!]_{\mathscr{D}}(\mathbf{R})) &::= \mathrm{attr}(\mathbf{R}) \setminus \{A'\} \cup \{A\} \\ \mathrm{attr}([\![\{\mathsf{Q}\}]\!]_{\mathscr{D}}(\mathbf{R})) &::= \mathrm{attr}([\![\mathsf{Q}]\!]_{\mathscr{D}}(\mathbf{R})) \cup \mathrm{attr}(\mathbf{R}) \\ \mathrm{attr}([\![\mathsf{Q}_1 \circ \mathsf{Q}_2]\!]_{\mathscr{D}}(\mathbf{R})) &::= \mathrm{attr}([\![\mathsf{Q}_1]\!]_{\mathscr{D}}(\mathbf{R})) \end{aligned}$$

Query output on a database is defined as $[\![\mathsf{Q}]\!]_{\mathscr{D}}(I_\varnothing)$ where $I_\varnothing$ is a singleton relation containing the *empty tuple* $\mu_\varnothing$ where $\mathrm{dom}(\mu_\varnothing) ::= \varnothing$, cf. [15, 127]. Thus, when we simply write $[\![\mathsf{Q}]\!]_{\mathscr{D}}$, we mean that the se-

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"
[127]: Francis et al. (2023), "A Researcher's Digest of GQL"

mantic function is applied to the fixed database $I_\varnothing$.

For example, consider a schema with $\text{attr}(S) = \text{attr}(T) = \{A_2\}$ and $\text{attr}(P) = \{A_1, A_2\}$ and a query

$$\mathrm{Q} \ ::= \ P\,\pi_{A_1}\,S\,\sigma_{A_2=1}\,\{S - T\}\,.$$

An LCRA query is read left-to-right. We start with the driving table $\mathbf{R}_0 \ ::= \ I_\varnothing$. Every clause will modify it resulting in a new value of $\mathbf{R}$. After processing the clause P, it becomes $\mathbf{R}_1 \ ::= \ \mathbf{R}_0 \bowtie P = P$. The next clause is π_{A_1} so this is applied to $\mathbf{R}_1$ resulting in $\mathbf{R}_2 \ ::= \ \pi_{A_1}(P)$. The next clause is S so the new value of the driving table $\mathbf{R}_3$ becomes the old value $\mathbf{R}_2$ joined with S, i.e., $\mathbf{R}_3 \ ::= \ \pi_{A_1}(P) \times S$ (as their attributes are disjoint). After processing the clause $\sigma_{A_2=1}$, we arrive at $\mathbf{R}_4 \ ::= \ \sigma_{A_2=1}(\mathbf{R}_3) = \sigma_{A_2=1}(\pi_{A_1}(P) \times S)$ with attributes A_1, A_2. Next, starting with this value of the driving table, rather than $I_\varnothing$, we evaluate the query $S - T$. This results in $\mathbf{R}' \ ::= \ \mathbf{R}_4 \bowtie S - \mathbf{R}_4 \bowtie T$. This relation has attributes A_1, A_2 and hence the result of the entire query is $\mathbf{R}_4 \bowtie \mathbf{R}' = \mathbf{R}_4 \cap \mathbf{R}'$ (since the attributes of $\mathbf{R}'$ and $\mathbf{R}_4$ are the same).

5.1.2 Expressivity results

We say that two queries $\mathrm{Q}_1, \mathrm{Q}_2$ (possibly from different languages) are *equivalent* if for every database $\mathscr{D}$ it holds that $[\![\mathrm{Q}_1]\!]_{\mathscr{D}} = [\![\mathrm{Q}_2]\!]_{\mathscr{D}}$. A query language L_1 is *subsumed* by L_2 if for each query Q_1 in L_1 there is an equivalent query Q_2 in L_2. If there is also a query $\mathrm{Q}_2 \in L_2$ for which there is no equivalent query $\mathrm{Q}_1 \in L_1$ then L_1 is said to be *strictly less expressive* than L_2. Finally L_1 and L_2 are *equivalent* if L_1 is subsumed by L_2 and L_2 is subsumed by L_1.

Theorem 5.1.1

For every schema $\mathbf{S}$, languages RA($\mathbf{S}$) and LCRA($\mathbf{S}$) are equivalent.

Proof. We first prove that RA($\mathbf{S}$) is subsumed by LCRA($\mathbf{S}$). Let Q be a query in RA($\mathbf{S}$). We show that there exists an equivalent query Q^{LCRA} in LCRA($\mathbf{S}$), i.e. such that for every database $\mathscr{D}$ over $\mathbf{S}$ it holds that $[\![Q]\!]_{\mathscr{D}} = [\![Q_L]\!]_{\mathscr{D}}$ by induction on the structure of Q.

- (base case) If $Q = R$ then $Q_L = S$ where $S = \{R\}$
- If $Q = \psi_{\bar{A}}(Q')$ and Q'_L is the LCRA query equivalent to Q' then $Q_L = Q'_L\,\pi_{\bar{A}}$
- If $Q = \sigma_\theta(Q')$ and Q'_L is the LCRA query equivalent to Q' then $Q_L = Q'_L\,\sigma_\theta$
- If $Q = \rho_{\bar{A}\to\bar{B}}(Q')$ and and Q'_L is the LCRA query equivalent to Q' then $Q_L = Q'_L\,\rho_{\bar{A}\to\bar{B}}$
  For the three following cases, Q' (resp. Q'') is systematically translated as $\{Q'_L\}$ (resp. $\{Q''_L\}$)
- If $Q = Q' \cup Q''$ and Q'_L (resp. Q''_L) is the LCRA query equivalent to Q' (resp. Q'') then $Q_L = \{Q'_L\} \cup \{Q''_L\}$
- If $Q = Q' \setminus Q''$ and Q'_L (resp. Q''_L) is the LCRA query equivalent to Q' (resp. Q'') then $Q_L = \{Q'_L\} \setminus \{Q''_L\}$
- If $Q = Q' \bowtie Q''$ and Q'_L (resp. Q''_L) is the LCRA query equivalent to Q' (resp. Q'') then $Q_L = \{Q'_L\}\{Q''_L\}$

We now prove that LCRA($\mathbf{S}$) is subsumed by RA($\mathbf{S}$). Let Q be a query in LCRA($\mathbf{S}$). We show that there exists a query $Q^{RA} \in$ RA($\mathbf{S}$) such that for

every database $\mathscr{D}$ over $\mathbf{S}$ it holds that $[\![Q]\!]_{\mathscr{D}} = [\![Q^{RA}]\!]_{\mathscr{D}}$ by induction on the structure of Q.

We start with the linear clauses. Since their structure is not tree-shaped, but indeed *linear*, the query must be parsed from left to right, while the equivalent RA query will be built bottom-up. Let $C = c_0, \dots, c_n$ be a linear clause in LCRA($\mathbf{S}$). We build the induction on the number of clauses of C.

- (base case) If $n = 1$ and C is an instance of the S rule, i.e. $C = R$ for some R then the RA query equivalent to C is R
- (base case) If $n = 1$ and C is one of $\pi_{\bar{A}}, \sigma_\theta$ or $\rho_{\bar{A}\to\bar{B}}$ then tthe RA query equivalent to C is $I_\varnothing$ (the empty tuple relation)
- (base case) If $n = 1$ and is C is an instance of the $\{Q\}$ rule, then the RA query equivalent to C is the one equivalent to Q (see induction for queries below)
  For the inductive cases, let C_i^{RA} be the RA query such that $[\![c_0, \dots, c_i]\!] = [\![C_i^{RA}]\!]$
- If $n > 1$ and c_{i+1} is an instance of the S rule, i.e. $c_{i+1} = R$, then $C_{i+1}^{RA} = C_i^{RA} \bowtie R$
- If $n > 1$ and $c_{i+1} = \pi_{\bar{A}}$, then $C_{i+1}^{RA} = \pi_{\bar{A}}(C_i^{RA})$
- If $n > 1$ and $c_{i+1} = \sigma_\theta$, then $C_{i+1}^{RA} = \sigma_\theta(C_i^{RA})$
- If $n > 1$ and $c_{i+1} = \rho_{\bar{A}\to\bar{B}}$, then $C_{i+1}^{RA} = \rho_{\bar{A}\to\bar{B}}(C_i^{RA})$
- If $n > 1$, $c_{i+1} = \{Q\}$ and C_Q^{RA} is the RA query equivalent to Q, then $C_{i+1} = C_i^{RA} \bowtie C_Q^{RA}$

We now treat the queries. Let Q be an LCRA query. We build the induction on the structure of Q.

- (base case) If $Q = L$ then the RA query equivalent to Q is the one equivalent to L (see above for induction on linear clauses).
- If $Q = Q_1 \cap Q_2$ and Q_1^{RA} (resp. Q_2^{RA}) is the RA query equivalent to Q_1 (resp. Q_2) then the RA query equivalent to Q is $Q_1^{RA} \cap Q_2^{RA}$
- If $Q = Q_1 \cup Q_2$ and Q_1^{RA} (resp. Q_2^{RA}) is the RA query equivalent to Q_1 (resp. Q_2) then the RA query equivalent to Q is $Q_1^{RA} \cup Q_2^{RA}$
- If $Q = Q_1 \setminus Q_2$ and Q_1^{RA} (resp. Q_2^{RA}) is the RA query equivalent to Q_1 (resp. Q_2) then the RA query equivalent to Q is $Q_1^{RA} \setminus Q_2^{RA}$

□

Thus, LCRA proposed as the relational processing engine of graph languages like Cypher and GQL is the good old RA in a slight disguise.

Notice that the definition of linear clauses and queries of LCRA are mutually recursive as we can feed any query Q back into clauses via {Q} (this corresponds to the `CALL` feature of Cypher and GQL). If this option is removed, and linear clauses are not dependent on queries, we get a simplified language sLCRA (simple LCRA). Specifically, in this language *linear clauses* are given by the grammar $\mathsf{L} ::= S \mid \pi_{\mathbf{A}} \mid \sigma_\theta \mid \rho_{A\to A'} \mid \mathsf{LL}$. To see why the {Q} was necessary in the definition of LCRA, we prove

Proposition 5.1.2

sLCRA is strictly less expressive than LCRA (and thus RA).

Proof. At first, observe that simple linear clauses (without {Q}) can only express conjunctive queries, as their semantics applies operations

$\bowtie, \pi, \sigma$ and renaming to base relations. Hence, queries of sLCRA are *Boolean combinations of conjunctive queries*, known as BCCQs. While it appears to be folklore that BCCQs are strictly contained in first-order logic, we were unable to find the simple proof explicitly stated in the literature, hence we offer one here.

Consider a vocabulary of a single unary predicate U and databases $\mathscr{D}_1$ and $\mathscr{D}_2$ such that $\mathscr{D}_1(U) = \{a_1\}$ and $\mathscr{D}_2(U) = \{a_1, a_2\}$ for two different constants a_1 and a_2. Since $\mathscr{D}_1$ and $\mathscr{D}_2$ are homomorphically equivalent, they agree on all conjunctive queries, and therefore on all BCCQs, but they do not agree on first-order (and hence RA) query that checks if relation U has exactly one element. □

5.1.3 The origins of linear composition

Linear composition appeared prominently in the design of Cypher [15] as a way to bypass the lack of a compositional language for graphs. Specifically, pattern matching (as we shall see shortly) transforms graphs into relational tables, and other Cypher operations modify these tables. If we have two such read-only queries $G \to \mathbf{R}_1$ and $G \to \mathbf{R}_2$ from graphs to relations, it is not clear how to compose them. To achieve composition, Cypher read-only queries are of the form $\mathrm{Q} : G \times \mathbf{R} \to \mathbf{R}'$ (and its read/write queries are of the form $\mathrm{Q} : G \times \mathbf{R} \to G' \times \mathbf{R}'$). Thus, the composition of two queries $\mathrm{Q}_1, \mathrm{Q}_2 : G \times \mathbf{R} \to \mathbf{R}'$ is their *linear composition* $\mathrm{Q}_1\ \mathrm{Q}_2$ which on a graph G and table $\mathbf{R}$ returns $\mathrm{Q}_2\big(G, \mathrm{Q}_1(G, \mathbf{R})\big)$.

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

Independently, the same approach was adopted by a relational language PRQL [128], where P stands for "pipelined" but the design philosophy is identical. For example one could write

```
FROM R   FILTER A=1   JOIN:INNER S   FILTER B=2   SELECT C, D
```

[128]: PRQL (2024), *Pipelined Relational Query Language*

with each clause applied to the output of the previous clause. The above query is the same as relational algebra query $\pi_{C,D}\Big(\sigma_{B=2}\big(\sigma_{A=1}(R) \bowtie S\big)\Big)$.

While PRQL design is relations-to-relations, the motivation for pipelined or linear composition comes from creating a database analog of `dplyr` [129], a data manipulation library in R, that can be translated to SQL. While `dplyr`'s operations are very much relational in spirit, it is integrated into a procedural language, and hence the imperative style of programming was inherited by PRQL.

[129]: Wickham et al. (2023), *dplyr: A Grammar of Data Manipulation*

5.2 GQL and SQL/PGQ: theoretical abstractions

Equipped with the formal definition of linear composition relational algebra LCRA, we can now provide simple theoretical abstractions Core GQL of GQL and Core PGQ of SQL/PGQ. In a nutshell:

- They both share the same pattern matching language that turns graphs into relations;
- Core PGQ corresponds to using RA on top of these relations, while
- Core GQL is obtained by using LCRA instead.

Of course it should be kept in mind that these abstractions capture the essence of SQL/PGQ and GQL in the same way as relational algebra and first-order logic capture the essence of SQL: they define a theoretical

core that is easy to study, but real languages, be it 500 pages of the GQL standards or thousands of pages of the SQL standard, have many more features.

Throughout the rest of this Chapter, we use the standard definition of property graphs, as presented in Chapter 2, with one simplification: we consider only directed edges and denote the set of such edges by E.

5.2.1 Pattern Matching: Turning Graphs into Relations

Pattern matching is the key component of graph query languages. We define the *pattern matching language* of Core GQL and Core PGQ as follows:

$$\psi \quad ::= \quad (x) \mid \overset{x}{\rightarrow} \mid \overset{x}{\leftarrow} \mid \psi_1\,\psi_2 \mid \psi^{n..m} \mid \psi\langle\theta\rangle \mid \psi_1 + \psi_2$$

where

- $x \in \mathrm{Vars}$
- $0 \le n \le m \le \infty$
- variables in node and edge patterns (x), $\overset{x}{\rightarrow}$, and $\overset{x}{\leftarrow}$ are optional,
- *conditions* are given by $\theta \;::=\; x.k = x'.k' \mid x.k < x'.k' \mid \ell(x) \mid \theta \vee \theta \mid \theta \wedge \theta \mid \neg\theta$ where $x, x' \in \mathrm{Vars}$ and $k, k' \in \mathscr{K}$;
- $\psi_1 + \psi_2$ is only defined when the schemas $\mathrm{sch}(\psi_1)$ and $\mathrm{sch}(\psi_2)$ are equal.

The *schemas of patterns* are defined by:

- $\mathrm{sch}\left((x)\right) = \mathrm{sch}\left(\overset{x}{\rightarrow}\right) = \mathrm{sch}\left(\overset{x}{\leftarrow}\right) \;::=\; \{x\}$;
- $\mathrm{sch}\left(\psi_1 + \psi_2\right) \;::=\; \mathrm{sch}\left(\psi_1\right)$
- $\mathrm{sch}\left(\psi_1\,\psi_2\right) \;::=\; \mathrm{sch}\left(\psi_1\right) \cup \mathrm{sch}\left(\psi_2\right)$
- $\mathrm{sch}\left(\psi^{n..m}\right) \;::=\; \varnothing$
- $\mathrm{sch}\left(\psi\langle\theta\rangle\right) \;::=\; \mathrm{sch}\left(\psi\right)$

Finally, we specify *outputs of patterns*, i.e., variables and properties of graph elements that form the table returned by a pattern. Let Ω be a (possibly empty) tuple whose elements are either variables x or *terms* $x.k$. Then a *pattern with output* is an expression π_Ω such that every variable present in Ω is in $\mathrm{sch}(\psi)$.

Semantics To define the semantics, we set Values as the union of $\mathrm{Const} \cup \mathscr{N} \cup E$. The *semantics of path patterns* ψ, with respect to a graph G, is a set of pairs (p, μ) where p is a path and μ is a mapping $\mathrm{sch}(\psi) \to \mathrm{Values}$. Recall that we write $\mu_\varnothing$ for the unique empty mapping with $\mathrm{dom}(\mu) = \varnothing$.

For the *semantics of path patterns with output* ψ_Ω we define the projection $: \Omega \to \mathrm{Values}$ of μ on Ω as :

$$\mu_\Omega(\omega) \;::=\; \begin{cases} \mu(x) & \text{if } \omega = x \in \mathrm{Vars} \\ \delta(\mu(x), k) & \text{if } \omega = x.k\,. \end{cases}$$

Full definitions are presented in Figure 5.2. For node and edge patterns with no variables, the mapping part of the semantics changes to $\mu_\varnothing$. The definition of $\models$ is standard: $\mu \models x.k = x'.k'$ if both $\delta(\mu(x), k)$ and $\delta(\mu(x'), k')$ are defined and equal (likewise for $<$), and $\mu \models \ell(x)$ if $\ell \in \mathrm{lab}(\mu(x))$, then extended to Boolean connectives $\wedge, \vee, \neg$.

For the reader familiar with Cypher and/or GQL, we explain how our formalization compares with these languages' patterns.

- (x) is a *node pattern* that binds the variable x to a node;
- $\xrightarrow{x}$ and $\xleftarrow{x}$ are *forward edge* and *backward edge patterns*, also bind x to the matched edge;
- $\psi_1\ \psi_2$ is the *concatenation* of patterns,
- $\psi^{n..m}$ is the *repetition* of ψ between n and m times (with a possibility of $m = \infty$)
- $\psi\langle\theta\rangle$ corresponds to `WHERE` in patterns, conditions involve (in)equalities between property values and their Boolean combinations;
- $\psi_1 + \psi_2$ is the *union* of patterns;
- ψ_Ω corresponds to the *output* forming clauses `RETURN` of Cypher and GQL and `COLUMNS` of SQL/PGQ, with Ω listing the attributes of returned relations.

Compared to GQL patterns as described in [96, 102, 127] and Chapters 3 and 4, we make some simplifications. First, the mapping part of any repeated pattern $\psi^{n..m}$ is empty. This is because in GQL repeated patterns return lists of bindings of variables, resulting in relations that violate the first normal form (1NF), which states that all relation elements must be atomic values. We believe that for the simple formalization in the spirit of relational algebra we ought to stay with 1NF relations, adding further complications (nesting, bags, etc) further down the road.

[96]: Deutsch et al. (2022), "Graph Pattern Matching in GQL and SQL/PGQ"
[102]: Francis et al. (2023), "GPC: A Pattern Calculus for Property Graphs"
[127]: Francis et al. (2023), "A Researcher's Digest of GQL"

Second, we do not impose any conditions on paths that can be matched. In GQL and PGQ they can be simple paths (no repeated nodes), or trails (no repeated edges), or shortest paths. In GQL as well as in Cypher, paths themselves may be returned, and such restrictions ensure finiteness of output. Since we can only return graph nodes or edges, or their properties, we never have the problem of infinite outputs, and thus we chose not to overcomplicate the definition of core languages by deviating from flat tables as outputs.

$$
\begin{aligned}
[\![(x)]\!]_G &::= \{(\mathrm{path}(n), \{x \mapsto n\}) \mid n \in \mathcal{N}\} \\
\left[\!\!\left[\xrightarrow{x}\right]\!\!\right]_G &::= \{(\mathrm{path}(n_1, e, n_2), \{x \mapsto e\}) \mid e \in \mathrm{E},\ \mathrm{src}(e) = n_1,\ \mathrm{tgt}(e) = n_2\} \\
\left[\!\!\left[\xleftarrow{x}\right]\!\!\right]_G &::= \{(\mathrm{path}(n_2, e, n_1), \{x \mapsto e\}) \mid e \in \mathrm{E},\ \mathrm{src}(e) = n_1,\ \mathrm{tgt}(e) = n_2\} \\
[\![\psi_1 + \psi_2]\!]_G &::= [\![\psi_1]\!]_G \cup [\![\psi_2]\!]_G \\
[\![\psi_1\ \psi_2]\!]_G &::= \left\{(p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \;\middle|\; (p_1, \mu_1) \in [\![\psi_1]\!]_G,\ (p_2, \mu_2) \in [\![\psi_2]\!]_G,\ \mu_1 \sim \mu_2,\ p_1 \| p_2\right\} \\
[\![\psi\langle\theta\rangle]\!]_G &::= \{(p, \mu) \in [\![\pi]\!]_G \mid \mu \vDash \theta\} \\
[\![\psi^{n..m}]\!]_G &::= \bigcup_{i=n}^{m} [\![\psi]\!]^i_G \text{ where} \\
[\![\psi]\!]^0_G &::= \left\{(\mathrm{path}(n), \mu_\varnothing) \;\middle|\; n \in \mathcal{N}\right\} \\
[\![\psi]\!]^n_G &::= \left\{(p_1 \cdots p_n, \mu_\varnothing) \;\middle|\; \exists \mu_1, \ldots, \mu_n : (p_i, \mu_i) \in [\![\psi]\!]_G \text{ and } p_i \| p_{i+1} \text{ for all } i < n\right\},\ n > 0 \\
[\![\psi_\Omega]\!]_G &::= \{\mu_\Omega \mid \exists p : (p, \mu) \in [\![\psi]\!]_G\}
\end{aligned}
$$

Figure 5.2: Semantics of patterns and patterns with output

Third, in disjunctions $\psi_1 + \psi_2$ we require that the schemas of ψ_1 and ψ_2 be the same. Otherwise, the output of the disjunction would be the outer union of the outputs of ψ_1 and ψ_2, and thus populated with nulls for variables that do not belong to both schemas. Again, we believe that nulls is not the feature that needs to be present in the first formalization in the spirit of RA.

In what might look like a simplification compared to GQL and SQL/PGQ, we do not have explicit joins of patterns, i.e., ψ_1, ψ_2 which result in the semantics being tuples of paths and an associated binding, i.e., $[\![\psi_1, \psi_2]\!]_G = \left\{((p_1, p_2), \mu_1 \bowtie \mu_2) \mid (p_i, \mu_i) \in [\![\psi_i]\!]_G,\ i = 1, 2\right\}$. Joins will however be definable with RA operations since we allow joins over tables $[\![\psi_\Omega]\!]_G$, in particular when Ω contains all the variables in $\mathrm{sch}(\psi)$.

5.2.2 GQL Vs. PGQ

Syntax

Assume that for each variable $x \in \mathrm{Vars}$ and each key $k \in \mathscr{K}$, both x and $x.k$ belong to the set of attributes $\mathscr{A}$. For each pattern ψ and each output specification Ω, we introduce a relation symbol $R_{\psi,\Omega}$ whose set of attributes are the elements of Ω. Let Pat contain all such relation symbols. Then:

- *Core PGQ* is defined as RA(Pat)
- *Core GQL* is defined as LCRA(Pat)

Semantics

We can assume without loss of generality that $\mathrm{Values} \subseteq \mathrm{Const}$. This ensures that results of pattern matching are relations of Pat:

Proposition 5.2.1

For every path pattern with output ψ_Ω and every graph database G, the set $[\![\psi_\Omega]\!]_G$ is an instance of relation $R_{\psi,\Omega}$ from Pat.

With this, the *semantics of Core GQL* and *PGQ* is a straightforward extension of the semantics of RA and LCRA as we only need to define the semantics of base relations by

$$\left[\!\left[R_{\psi,\Omega}\right]\!\right]_G \ ::= \ [\![\psi_\Omega]\!]_G$$

and then use the semantic rules for patterns from Fig. 5.2 and for RA, from Chapter 2, and LCRA from Section 5.1.

As an immediate consequence of Theorem 5.1.1 we have:

Corollary 5.2.2

The languages Core PGQ and Core GQL have the same expressive power.

5.2.3 Example

We use a simplified query based on the medieval money laundering query on page 97. It looks for someone who has two friends in a city different from theirs, and outputs the person's name and account (`Porthos` and `a2` in our example):

```
MATCH (x)-[:Friends]->(y)-[:Friends]->(z), (y)-[:Owns]->(acc_y)
FILTER (y.city) <> (x.city) AND (x.city=z.city)
RETURN y.name AS name, acc_y AS acc
```

The equivalent Core GQL formula is

$$R_{\psi_1,\Omega_1}\ R_{\psi_2,\Omega_2}\ \sigma_{y.\text{city}\neq x.\text{city}\wedge x.\text{city}=z.\text{city}}\ \pi_{y.\text{name},\text{acc\_}y}\ \rho_{y.\text{name}\to\text{name}}\ \rho_{\text{acc\_}y\to\text{acc}}$$

where

$$\begin{array}{lcl}
\psi_1 & ::= & \big((x) \xrightarrow{e_1} (y)\ (y) \xrightarrow{e_2} (z)\big)\langle \text{Friends}(e_1) \wedge \text{Friends}(e_2)\rangle \\
\psi_2 & ::= & \big((y) \xrightarrow{e_3} (\text{acc_}y)\big)\langle \text{Owns}(e_3)\rangle \\
\Omega_1 & ::= & (x,\ y,\ z,\ x.\text{city},\ y.\text{city},\ z.\text{city}) \\
\Omega_2 & ::= & (y,\ \text{acc_}y)\,.
\end{array}$$

5.3 Case study 1: Expressiveness of Pattern Matching

In this section we concentrate on the expressiveness of pattern matching, specifically negative results, that certain patterns cannot be expressed in a natural way in the pattern language of SQL/PGQ and GQL. To analyze the expressiveness of *patterns*, there is a natural way of defining queries given by them, in the spirit of RPQs which define pairs of nodes connected by a path. Take any pattern ψ and turn it into a pattern with output

$$\big((x_s)\ \psi\ (x_t)\big)_{x_s,x_t} \tag{5.1}$$

that will output pairs of source and target nodes x_s and x_t connected by a path satisfying ψ. If we are given a query that maps a property graph G (possibly from a class of graphs $\mathscr{C}$) to a pair of nodes, we say it is *expressible* by a pattern ψ if its output is given by pattern with output on every graph (from $\mathscr{C}$).

To see what types of patterns are beyond GQL and PGQ capabilities, notice the idea of their design: we first test *local* properties (e.g., the existence of edges between certain nodes) and then, if necessary, repeat those patterns to test for reachability and similar properties. It thus appears intuitive that testing a condition on a path *as a whole* will be out of reach. Our sample query for this will be checking if all nodes on a path have different values of some given property (e.g., "find a path of transfers between two accounts where all transfer amounts are different").

What is more interesting however, as it reveals deficiencies of the design of GQL and PGQ patterns, is that testing for local conditions under repetition is highly sensitive to whether we specify conditions on nodes or edges. For example:

- we *can* check, by a very simple pattern, if there is a path of transfers between two accounts where balances in intermediary accounts (values held in nodes) increase, but

- we *cannot* check, by natural patterns, if there is a path of transfers between two accounts where amounts of transfers (values held in edges) increase.

Of course what it means to be natural in this context will be precisely defined. This is a type of inexpressibility result that points to a deficiency in language design that ought to be fixed.

We now start with these results, before looking at conditions on paths as a whole. We conclude this section by adjusting our definition of patterns to model Cypher patterns as they originally appeared [15] and confirm a previously unproven folklore result that Cypher patterns cannot express all RPQs.

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

5.3.1 Repeated local conditions

To capture the queries we just outlined, we define two queries on any property graph G:

- $Q^{\mathrm{N}}_{<}$ returns endpoints of a path along which the value of property k of nodes increases. Specifically, it returns pairs (u_0, u_n) of nodes such that there is path $\mathrm{path}(u_0, e_1, u_1, \ldots, e_n, u_n)$ so that $u_0.k < u_1.k < \cdots < u_n.k$
- $Q^{\mathrm{E}}_{<}$ returns endpoints of a path along which the value of property k of edges increases. Specifically, it returns pairs (n_1, n_m) of nodes such that there is path $\mathrm{path}(u_0, e_1, u_1, \ldots, e_n, u_m)$ so that $e_1.k < e_2.k < \cdots < e_m.k$

It is a simple observation that $Q^{\mathrm{N}}_{<}$ is expressible by the Core PGQ and GQL pattern

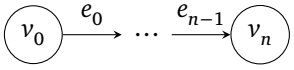
$$\big((x) \to (y)\langle x.k < y.k\rangle\big)^{0..\infty} .$$

However, the same approach does not work for edges. If we (erroneously) try to express $Q^{\mathrm{E}}_{<}$ by

$$\Big(() \xrightarrow{x} () \xrightarrow{y} ()\langle x.k < y.k\rangle\Big)^{0..\infty}$$

it fails, as on the input $() \xrightarrow{3} () \xrightarrow{4} () \xrightarrow{1} () \xrightarrow{2} ()$ (where the numbers on the edges are values of property k) it returns the start and the end node of the path, even though values in edges do not increase. This is explained by the semantics of path concatenation, as two paths concatenate if the last node of the first path is the same as the first node of the second path, and therefore conditions on edges in two concatenated or repeated paths are completely "local" to those paths.

In some cases, where graphs are of special shape, we can obtain the desired patterns by using "tricks" a normal query language user would be unhappy to resort to. Specifically, consider graphs whose underlying graph structures are just paths. That is, we look at *annotated paths*, which are of the form



where $v_0, \ldots, v_n$ are distinct nodes, $e_0, \ldots, e_{n-1}$ are distinct edges (for $n > 0$), and each edge e_i has the property $e_i.k$ defined. Then define the pattern

$$\psi^{\mathrm{E}}_{<} \;::=\; (x_s)\Big(\big((u) \xrightarrow{x} (z) \xrightarrow{y} (v) \xleftarrow{w} (z)\big)\langle x.k < y.k\rangle\Big)^{0..\infty}(x_t)$$

Proposition 5.3.1

The pattern with output $(\psi^{\mathsf{E}}_{<})_{\{x_s,x_t\}}$ expresses $Q^{\mathsf{E}}_{<}$ on annotated paths.

Intuitively, the forward edge $\xrightarrow{y}$ serves as a look-ahead that enables checking whether the condition holds, and the backward edge $\xleftarrow{w}$ enables us to continue constructing the path in the next iteration from the correct position (z).

This way of expressing queries is very unnatural however. It assumes a restriction on a graph that is a directed path with forward edges only, but then the pattern uses backward edges, as if we traversed an oriented path (in which edges can go in either direction [130]). This of course is used to traverse the same forward edge back and forth.

[130]: Hell et al. (2004), *Graphs and homomorphisms*

The question is: can $Q^{\mathsf{E}}_{<}$ be expressed in a *natural* way? We formalize being natural by means of *one-way path patterns* ρ defined by a restriction of the grammar:

$$\psi \;::=\; (x) \;\mid\; \xrightarrow{x} \;\mid\; \psi + \psi \;\mid\; \psi^{n..m} \;\mid\; \psi_{\langle\theta\rangle} \;\mid\; \psi_1\,\psi_2$$

where $\psi_1\,\psi_2$ is defined only when $\mathrm{sch}(\psi_1) \cap \mathrm{sch}(\psi_2) = \varnothing$ and x is optional. The omission of backward edges $\xleftarrow{x}$ in one-way patterns is quite intuitive. The reason for the restriction on variable sharing in concatenated patterns is that backward edges can be simulated by simply repeating variables, as we have done above in $\psi^{\mathsf{E}}_{<}$.

If there is a natural way of writing the $Q^{\mathsf{E}}_{<}$ query in PGQ and GQL, one would expect it to be done without backward edges. However, this is not the case.

Theorem 5.3.2

No one-way path pattern query expresses $Q^{\mathsf{E}}_{<}$.

To give the idea behind the proof, assume by contradiction that one can express $Q^{\mathsf{E}}_{<}$ by a pattern π. Since, by definition, there are infinitely many annotated paths that conform to $Q^{\mathsf{E}}_{<}$, the pattern π must contain an unbounded repetition. However, since the semantics disregards variables that occur under unbounded repetition, transferring information from one iteration to the other is restricted. This enables us to repeat parts of the annotated path while maintaining the same semantics, and in particular, conforming to π. Nevertheless, due to the definition of $Q^{\mathsf{E}}_{<}$, repeating non-empty parts of the annotated path breaks the condition which leads to the desired contradiction.

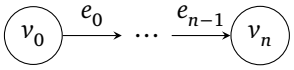
To prove the above formally, we start by defining the language of patterns on annotated paths and showing that the words that belong to such a language must conform to specific shapes, in particular when the pattern contains an unbounded repetition.

For an annotated path p, we define w_p as the sequence $e_0.k \cdots e_{n-1}.k$.

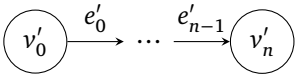
Lemma 5.3.3

For every one-way path pattern ψ and annotated paths p, p' with $w_{p'} = w_p$, if $(p, \mu) \in [\![\psi]\!]_p$ then there is μ' such that $(p', \mu') \in [\![\psi]\!]_{p'}$.

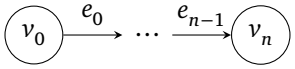
Proof. Assume that $(p, \mu) \in [\![\psi]\!]_p$. Let us denote p by

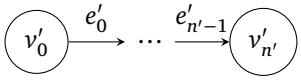


and p' by



(Note that they have the same length since $w_p = w_{p'}$.) We denote by f the mapping defined by $f(v_i) ::= v'_i$ for $0 \leq i \leq n$, and $f(e_i) = e'_i$ for $0 \leq i \leq n-1$. We then define μ' by setting $\mathrm{dom}(\mu') = \mathrm{dom}(\mu)$ and $\mu'(x) ::= f(\mu(x))$. It can be shown by induction on ψ that if $(p, \mu) \in [\![\psi]\!]_p$ then $(p', \mu') \in [\![\psi]\!]_{p'}$. □

For two annotated paths $p ::=$ 

and $p' ::=$ 

we use $p' \sqsubseteq p$ to denote that p' is contained within p. Formally, $p' \sqsubseteq p$ if there is j such that $v'_0 = v_j, \ldots, v'_{n'} = v_{j+n'}$, and $e'_0 = e_j, \ldots, e'_{n'-1} = e_{j+n'-1}$.

Lemma 5.3.4

For every one-way path pattern ψ and annotated paths p, p' where $p' \sqsubseteq p$,

(1) if there is μ such that $(p', \mu) \in [\![\psi]\!]_p$ then $(p', \mu) \in [\![\psi]\!]_{p'}$

(2) $[\![\psi]\!]_{p'} \subseteq [\![\psi]\!]_p$

Proof. We use the same notation as above and show the claim by a mutual induction on ψ. We skip the induction basis since it is trivial, and refer to the following interesting cases in the induction step:

(1) Let $\psi = \psi_1\psi_2$ and assume $(p', \mu) \in [\![\psi_1\psi_2]\!]_p$. By definition, there are p_1, p_2, μ_1, μ_2 such that $(p_1, \mu_1) \in [\![\psi_1]\!]_p$, $(p_2, \mu_2) \in [\![\psi_2]\!]_p$, p_2 concatenates to p_1, $\mu_1 \sim \mu_2$ and $p' = p_1p_2$. Notice that $p_1 \sqsubseteq p'$ and $p_2 \sqsubseteq p'$, and hence also $p_1p_2 \sqsubseteq p'$. By applying induction hypothesis (1), we get $(p_1, \mu_1) \in [\![\psi_1]\!]_{p_1}$, $(p_2, \mu_2) \in [\![\psi_2]\!]_{p_2}$. By applying induction hypothesis (2) we get $(p_1, \mu_1) \in [\![\psi_1]\!]_{p_1p_2}$, $(p_2, \mu_2) \in [\![\psi_2]\!]_{p_1p_2}$. This allows us to conclude that, by definition, $(p_1p_2, \mu_1 \bowtie \mu_2) \in [\![\psi_1\psi_2]\!]_{p_1p_2}$.

(2) If $\psi = \psi_1\psi_2$ then

$$[\![\psi_1\psi_2]\!]_{p'} = \left\{ (p_1\,p_2, \mu_1 \bowtie \mu_2) \;\middle|\; \begin{array}{l} (p_1,\mu_1) \in [\![\psi_1]\!]_{p'}, \\ (p_2,\mu_2) \in [\![\psi_2]\!]_{p'}, \\ \mu_1 \sim \mu_2, \\ p_2 \text{ concatenates to } p_1 \end{array} \right\}.$$

By induction hypothesis (2), $[\![\psi_1]\!]_{p'} \subseteq [\![\psi_1]\!]_p$ and $[\![\psi_2]\!]_{p'} \subseteq [\![\psi_2]\!]_p$. Thus,

$$[\![\psi_1\psi_2]\!]_{p'} \subseteq \left\{ (p_1\,p_2, \mu_1 \bowtie \mu_2) \;\middle|\; \begin{array}{l} (p_1,\mu_1) \in [\![\psi_1]\!]_{p}, \\ (p_2,\mu_2) \in [\![\psi_2]\!]_{p}, \\ \mu_1 \sim \mu_2, \\ p_2 \text{ concatenates to } p_1 \end{array} \right\}.$$

By definition, $[\![\psi_1\psi_2]\!]_{p'} \subseteq [\![\psi_1\psi_2]\!]_p$. □

Let ψ be a path pattern. We define the *language* of ψ as:

$$\mathcal{L}(\psi) = \{w \mid \forall\, p : \Big(w = w_p \rightarrow \exists\, \mu : (p,\mu) \in [\![\psi]\!]_p\Big)\}$$

For concatenation, we have:

Lemma 5.3.5

For every one-way path patterns ψ_1, ψ_2, the following equivalence holds: $\mathcal{L}(\psi_1\psi_2) = \mathcal{L}(\psi_1)\mathcal{L}(\psi_2)$.

Proof. $\subseteq$ **direction:** Assume $w \in \mathcal{L}(\psi_1\psi_2)$. By definition, for every p such that $w = w_p$ there is μ such that $(p,\mu) \in [\![\psi_1\psi_2]\!]_p$. In turn, by definition of $[\![\psi_1\psi_2]\!]_p$ we can conclude that there are paths p_1, p_2 such that $p = p_1p_2$ and there are μ_1, μ_2 for which $(p_1,\mu_1) \in [\![\psi_1]\!]_p$, $(p_2,\mu_2) \in [\![\psi_2]\!]_p$, and $\mu_1 \sim \mu_2$. We set $w_1 = w_{p_1}$ and $w_2 = w_{p_2}$. For p_1 we already saw that there is a μ_1 such that $(p_1,\mu_1) \in [\![\psi_1]\!]_p$. Let p' be such that $w_1 = w_{p_1} = w_{p'}$. By Lemma 5.3.3, we can conclude that there is μ' such that $(p',\mu') \in [\![\psi_1]\!]_{p'}$. Hence, by definition $w_1 \in \mathcal{L}(\psi_1)$. We can show similarly that $w_2 \in \mathcal{L}(\psi_2)$, which completes this direction.

$\supseteq$ **direction:** Assume that $w \in \mathcal{L}(\psi_1)\mathcal{L}(\psi_2)$. That is, there are w_1, w_2 such that $w = w_1w_2$ and $w_1 \in \mathcal{L}(\psi_1), w_2 \in \mathcal{L}(\psi_2)$. Let p be a path such that $w_p = w = w_1w_2$. Let p_1, p_2 be its subpaths such that $p = p_1p_2$ and $w_{p_1} = w_1, w_{p_2} = w_2$. Since $w_{p_1} = w_1$ there are μ_1, μ_2 such that $(p_1,\mu_1) \in [\![\psi_1]\!]_{p_1}, (p_2,\mu_2) \in [\![\psi_2]\!]_{p_2}$. Since ψ is a one-way path pattern, it holds that $\mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2) = \emptyset$, and thus $\mu_1 \sim \mu_2$. Since $p_1, p_2 \sqsubseteq p$, applying Lemma 5.3.4 (2) enables us to conclude that $(p_1,\mu_1) \in [\![\psi_1]\!]_p, (p_2,\mu_2) \in [\![\psi_2]\!]_p$. By definition of $[\![\psi_1\psi_2]\!]_p$, we can conclude that $(p_1p_2, \mu_1 \bowtie \mu_2) \in [\![\psi_1\psi_2]\!]_p$. This suffices to conclude that $w \in \mathcal{L}(\psi_1\psi_2)$, which completes this direction. □

For repetition, we have:

Lemma 5.3.6

For every one-way path pattern ψ, $\mathscr{L}(\psi^2) = \mathscr{L}(\psi)\mathscr{L}(\psi)$.

Proof. $\subseteq$ **direction:** Assume that $w \in \mathscr{L}(\psi^2)$ and let p be a path with $w = w_p$. By definition

$$[\![\psi^2]\!]_p = \left\{ (p_1p_2, \varnothing) \;\middle|\; \begin{array}{l} \exists \mu_1, \mu_2 : \\ (p_1, \mu_1) \in [\![\psi]\!]_p, (p_2, \mu_2) \in [\![\psi]\!]_p, \\ p_2 \text{ concatenates to } p_1 \end{array} \right\}$$

Therefore, $p = p_1p_2$ for some p_1, p_2 such that there are μ_1, μ_2 where $(p_1, \mu_1), (p_2, \mu_2) \in [\![\psi]\!]_p$.
Let us denote $w_1 ::= w_{p_1}$ and $w_2 ::= w_{p_2}$. It holds that $w = w_1w_2$ and it suffices to show that $w_1, w_2 \in \mathscr{L}(\psi)$. For p_1 there is μ_1 such that $(p_1, \mu_1) \in [\![\psi]\!]_p$. Due to Lemma 5.3.4 (1), since $p_1 \sqsubseteq p$ we can conclude that $(p_1, \mu_1) \in [\![\psi]\!]_{p_1}$. Due to Lemma 5.3.3, for every p with $w_p = w_1$ it holds that $(p, \mu_1) \in [\![\psi]\!]_p$. Hence $w_1 \in \mathscr{L}(\psi)$. We show similarly that $w_2 \in \mathscr{L}(\psi)$, which completes this direction.

$\supseteq$ **direction:** Assume that $w_1, w_2 \in \mathscr{L}(\psi)$. Let p_1, p_2 be paths such that $w_{p_1} = w_1, w_{p_2} = w_2$. By definition of $\mathscr{L}(\psi)$, there are μ_1, μ_2 such that $(p_1, \mu_1) \in [\![\psi]\!]_{p_1}$ and $(p_2, \mu_2) \in [\![\psi]\!]_{p_2}$. If p_2 does not concatenate to p_1, then by Lemma 5.3.3 we can change its first node id to match the last node id of p_1. Thus, we can assume that p_2 concatenates to p_1. Due to Lemma 5.3.4 (2), it holds that $(p_1, \mu_1) \in [\![\psi]\!]_p$ and $(p_2, \mu_2) \in [\![\psi]\!]_p$ where $p = p_1p_2$. Therefore, $(p_1p_2, \varnothing) \in [\![\psi^2]\!]_p$ by definition which completes this direction. □

We can further generalize this lemma by replacing 2 with any $k \geq 1$.

Lemma 5.3.7

For every one-way path patterns ψ, $\mathscr{L}(\psi^k) = (\mathscr{L}(\psi))^k$, $k \geq 1$

Proof. The claim can be shown by induction on k by combining Lemmas 5.3.5 and 5.3.6. □

To apply the above lemmas to our settings we first present a Normal Form for path patterns. We say that a one-way path pattern is in *+NF* if + occurs only under unbounded repetition.

Lemma 5.3.8

For every one-way path pattern ψ there is a one-way path pattern ψ' in +NF such that for every annotated path p, $[\![\psi]\!]_p = [\![\psi']\!]_p$.

Proof. We show that there are translation rules that preserve semantics on annotated paths. This translation tr is described inductively.

- $\mathrm{tr}((x)) ::= (x)$
- $\mathrm{tr}\left(\overset{x}{\rightarrow}\right) ::= \overset{x}{\rightarrow}$

- $\mathrm{tr}\left(\overset{x}{\leftarrow}\right) ::= \overset{x}{\leftarrow}$
- $\mathrm{tr}\left(\psi_1 + \psi_2\right) ::= \psi_1 + \psi_2$
- $\mathrm{tr}\left(\psi_1\psi_2\right) ::= +_{1\le i\le k, 1\le j\le m} \rho_i\rho'_j$ where $\mathrm{tr}\left(\psi_1\right) ::= \rho_1 + \cdots + \rho_k$ and $\mathrm{tr}\left(\psi_2\right) ::= \rho'_1 + \cdots + \rho'_m$
- When $m < \infty$ we define $\mathrm{tr}\left(\psi^{n..m}\right) ::= \mathrm{tr}(\underbrace{\psi\cdots\psi}_{n \text{ times}}) + \cdots + \mathrm{tr}(\underbrace{\psi\cdots\psi}_{m \text{ times}})$
- When $m = \infty$ we define $\mathrm{tr}\left(\psi^{n..m}\right) ::= \psi^{n..m}$
- $\mathrm{tr}\left(\psi_{\langle\theta\rangle}\right) ::= \mathrm{tr}\left(\psi\right)_{\langle\theta\rangle}$

It can be shown that the output of tr is indeed in +NF and the equivalence of the semantics of ψ and $\mathrm{tr}(\psi)$ can be shown by induction based on Lemmas 5.3.7 and 5.3.5. □

We are now ready to move to the main proof.

Proof of Theorem 5.3.2. Let ψ be a one-way path pattern in +NF of the form $\psi_1 + \cdots + \psi_m$. By pigeonhole principle, if $\mathscr{L}(\psi)$ is infinite then there is at least one ψ_i with infinite $\mathscr{L}(\psi_i)$. Since ψ is in +NF we can denote ψ_i as $\rho_1\rho^{n..\infty}\rho_2$. By Lemmas 5.3.5 and 5.3.7, we know that $\mathscr{L}(\psi_i) = \mathscr{L}(\rho_1)\mathscr{L}(\rho)^{n..\infty}\mathscr{L}(\rho_2)$. Let w be a word in $\mathscr{L}(\rho)^{n..\infty}$. As $w \in \mathscr{L}(\rho)^{n..\infty}$, then $ww \in \mathscr{L}(\rho)^{n..\infty}$ as well, i.e. there exists a pair $(p, \mu) \in [\![\psi]\!]_p$ such that ww is a subword of w_p. However, as ww repeats each value of w twice, it cannot satisfy the condition. □

5.3.2 Global conditions

Previously, we discussed repeated local conditions: intuitively, to see if values in nodes or edges increase, we need to look at two consecutive nodes or edges in a path, and repeat that pattern. There are, however, conditions that cannot be checked without examining a path as a whole. A canonical example of such a condition is whether all values of a given property of nodes are distinct. Formally, we define this query as

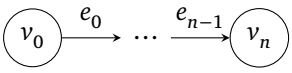
- $Q^{\mathrm{N}}_{\neq}$ returns pairs (u_0, u_n) of nodes such that there is path $\mathrm{path}(u_0, e_1, u_1, \ldots, e_n, u_n)$ so that $u_i.k \neq u_j.k$ for all $0 \le i < j \le n$.

Similarly to what we saw in the previous section, one-way path patterns do not have sufficient power to express such queries.

Theorem 5.3.9

No one-way path pattern expresses $Q^{\mathrm{N}}_{\neq}$.

First, notice that if a path contains a cycle, it cannot be an answer to $Q^{\mathrm{N}}_{\neq}$ as the value in the node the path comes back to would be repeated (at least) twice, so we can safely consider only "path shaped" graphs with values on the nodes. Formally, we look at *node-annotated paths* of the form



where $v_0, \ldots, v_n$ are distinct nodes, $e_0, \ldots, e_{n-1}$ are distinct edges (for $n > 0$) and each v_i has the property $v_i.k$ defined. The *node-value word* of a node-annotated path is defined as $w^N_p = v_1.k \cdot v_2.k \cdots v_n.k$.

As for the proof of Theorem 5.3.2, we define the *node-value language* of a one-way path pattern ψ as $\mathscr{L}^N(\psi) = \{w \mid \exists (p,\mu) \in [\![\psi]\!]_p, w_p^N = w\}$.

Lemma 5.3.10

For every one-way infinitely-repeated path pattern $\psi^{n..\infty}$, if a word $w \in \mathscr{L}^N(\psi^{n..\infty})$ then the word ww also belongs to $\mathscr{L}^N(\psi^{n..\infty})$.

Proof. Let w be a word in $\mathscr{L}^N(\psi^{n..\infty})$. By definition of $\mathscr{L}^N$, there exists a path $p = \text{path}(v_1, e_1, v_2, \dots e_{k-1}, v_k)$ and a mapping μ such that $(p,\mu) \in [\![\psi^{n..\infty}]\!]_p$ and $w_p^N = w$. Let $p' = \text{path}(v'_1, e'_1, \dots e'_{k-1}, v'_k)$ be a copy of p with fresh ids for all elements (nodes and edges) except for v'_1 for which $id(v'_1) = id(v_k)$. By definition of path concatenation, p and p' concatenate and, since $(p,\mu) \in [\![\psi^{n..\infty}]\!]_p$, we can construct a mapping μ' such that $(p',\mu') \in [\![\psi^{n..\infty}]\!]_{p'}$ as follows: $\mu'(x) = v'_i$ whenever $\mu(x) = v_i$ for all $0 \le i \le k$ and $\mu'(y) = e'_j$ whenever $\mu(y) = e_j$ for all $0 \le j < k$. As $(p,\mu) \in [\![\psi^{n..\infty}]\!]_p$ (resp. $(p',\mu') \in [\![\psi^{n..\infty}]\!]_{p'}$), it can easily be shown by induction that $(p,\mu) \in [\![\psi^{n..\infty}]\!]_{p\|p'}$ (resp. $(p',\mu') \in [\![\psi^{n..\infty}]\!]_{p\|p'}$) and, by definition of the semantics of repetition, we can conclude that $(p\|p', \mu_\varnothing) \in [\![\psi^{n..\infty}]\!]_{p\|p'}$ and so we get that $w_p^N \cdot w_{p'}^N = w \cdot w$ belongs to $\mathscr{L}^N(\psi^{n..\infty})$. □

Lemma 5.3.11

For any one-way path patterns ψ, ψ', ψ'' such that $\mathscr{L}^N(\psi'\psi\psi'') \neq \varnothing$ and any node-value word w, if $w \in \mathscr{L}^N(\psi)$ then there exists a node-annotated path p such that $w_p^N \in \mathscr{L}^N(\psi'\psi\psi'')$ and w is a subword of w_p^N.

Proof. Let w be a word in $\mathscr{L}^N(\psi)$. By definition of $\mathscr{L}^N$, there exists a path $p = \text{path}(v_1, e_1, \dots, v_k)$ and a mapping μ such that $(p,\mu) \in [\![\psi]\!]_G$ and $w = w_p^N$ for some graph G. By the semantics of concatenation and since $\mathscr{L}^N(\psi_1\psi\psi_2) \neq \varnothing$, there are paths $p' = \text{path}(v'_1, e'_1, \dots, v'_{k'})$, $p'' = \text{path}(v''_1, e''_1, \dots, v''_{k''})$, and mappings μ', μ'' such that $(p',\mu') \in [\![\psi']\!]_G$, $(p'',\mu'') \in [\![\psi'']\!]_G$, $p'\|p$, $p\|p''$ and μ, μ' and μ'' are compatible. By definition of path concatenation, $p'\|p\|p''$ forms a path whose node-value word is $w^N_{p'\|p\|p''} = v'_1.k \cdots v'_{k'}.k \cdot v_2.k \cdots v_k.k \cdot v''_2.k \cdot v''_{k''}.k$. Since the paths concatenate, we know that $v'_{k'} = v_1$ and $v''_1 = v_k$ and so w is a subword of $w^N{}_{p'\|p\|p''}$. Once again by definition of concatenation, we get that $(p'\|p\|p'', \mu' \bowtie \mu \bowtie \mu'') \in [\![\psi'\psi\psi'']\!]_G$ and so we can conclude that $w^N{}_{p'\|p\|p''} \in \mathscr{L}^N(\psi'\psi\psi'')$. □

The above lemma can be easily extended to concatenations of arbitrary size.

We can now prove Theorem 5.3.9.

Proof. Assume, by contradiction, that there is a one-way path pattern ψ equivalent to $Q^N_{\neq}$. By lemma 5.3.8, we can assume that ψ is in +NF, and so of the shape $\psi_1 + \cdots + \psi_m$, and since the language of $Q^N_{\neq}$ is infinite, we can further assume that at least one of the ψ_i is of the shape $\rho_1 \dots \rho_j^{n..\infty} \dots \rho_k$. Let w be a word in $\mathscr{L}^N(\rho_j^{n..\infty})$. By lemma 5.3.10, we

have that ww also belongs to $\mathscr{L}^N(\rho_j^{n..\infty})$ and, by lemma 5.3.11 we can conclude that there exists a path p and a mapping μ such that ww is a subword of w_p^N and $(p,\mu) \in \mathscr{L}^N(\rho_1 \dots \rho_j^{n..\infty} \dots \rho_k)$. Since all values in ww are repeated twice, ψ cannot be equivalent to $Q_{\neq}^{\mathsf{N}}$. □

How do SQL/PGQ and GQL handle such patterns? To start with, our results do not preclude a possibility of using backward edges to express such patterns on unrestricted graphs, though we conjecture that this is impossible for $Q_{\neq}^{\mathsf{N}}$ as well as $Q_{<}^{\mathsf{N}}$ and $Q_{<}^{\mathsf{E}}$. In GQL and PGQ paths can be named, and included in the output. That is, we may have an additional constructon $p ::= \psi$ that names paths matched by ψ as p, and this p could be returned: $(p ::= \psi)_p$ returns all paths satisfying ψ. This is however an advanced feature that we omitted in our core language as it entails having non-1NF outputs as the resulting paths are represented as lists, making the output a non-flat relation.

However, this gives a backdoor to highly inefficient expressivity of the queries studied in this section. The key observation is that their *complements* are expressible. For example, the complement of $Q_{\neq}^{\mathsf{N}}$ is the query that finds endpoints of paths on which two values of property k are equal. This is very easy to express by a pattern $\psi_=$, and the entire query can be formulated as the difference of $(p ::= ((x)\to(z))^{1..\infty})_p - (p ::= \psi_=)_p$ which results in all the paths in a graph satisfying $Q_{\neq}^{\mathsf{N}}$. Note that the first subquery simply returns *all* paths, making it highly inefficient.

5.3.3 Cypher patterns

We can use our formalization approach to prove a result about limitations of Cypher patterns; in fact those limitations specifically led to new features present in GQL and PGQ. Note that Cypher as a query language is constantly evolving, and in particular with the development of the GQL standard, every new release of Cypher adopts some new GQL features. When we speak of Cypher we mean the original language as described in [15]. Its *patterns* are given by the grammar:

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

$$\begin{aligned}\psi \quad ::= \quad & (x) \mid (x) \overset{y:\ell}{\to} (z) \mid (x) \overset{y:\ell}{\leftarrow} (z) \mid (x) \overset{:\ell*}{\to} (z) \mid (x) \overset{:\ell*}{\leftarrow} (z) \\ & \mid \psi_1\, \psi_2 \mid \psi\langle\theta\rangle \mid \psi_1 + \psi_2\end{aligned}$$

with $x, y, z \in \mathsf{Vars}$ (all optional). That is, the only repetition allowed is the Kleene star over edges with the same label. The *semantics* is modified as follows:

$$\begin{aligned}\left[\!\left[(x) \overset{y:\ell}{\to} (z)\right]\!\right]_G &= \{(\mathsf{path}(n_1, e, n_2), \{x \mapsto n_1, y \mapsto e, z \mapsto n_2\}) \mid \\ &\qquad e \in \mathsf{E}, \mathsf{src}(e) = n_1, \mathsf{tgt}(e) = n_2, \mathsf{lab}(e) = \ell\} \\ \left[\!\left[(x) \overset{:\ell*}{\to} (z)\right]\!\right]_G &= \{(\mathsf{path}(n_1, e_1, n_2, \dots, e_{k-1}, n_k), \{x \mapsto n_1, z \mapsto n_k\}) \mid \\ &\qquad e_1, \dots, e_{k-1} \in \mathsf{E}, \mathsf{src}(e_i) = n_i, \\ &\qquad \mathsf{tgt}(e_i) = n_{i+1}, \text{ and } \mathsf{lab}(e_i) = \ell \text{ for all } i < k\}\end{aligned}$$

and likewise for backwards arrows. The *schema* for a pattern will now have two distinguished variables, its *source* and *target variables*. For (x), both source and target are x, for forward arrows in the above grammar sources are x and targets are z, and for backwards arrows it is the

other way around. We then require that $\psi_1\,\psi_2$ be defined only when the source variable of ψ_2 is the target variable of ψ_1; the source of $\psi_1\,\psi_2$ is then the source of ψ_1 and its target is the target of ψ_2. For $\psi_1 + \psi_2$ we require that the schemas of ψ_1 and ψ_2 be the same, including their source and target.

The notion of patterns with output is defined exactly as in Section 5.2. We then define *Core Cypher* as $\mathrm{LCRA}(\mathrm{Pat}^{\mathrm{Cypher}})$ where $\mathrm{Pat}^{\mathrm{Cypher}}$ consists of all symbols $R_{\psi,\Omega}$ with ψ now ranging over Cypher patterns.

Theorem 5.3.12

Core Cypher patterns are strictly weaker than Core GQL patterns. In particular they cannot express the pattern testing for an even-length path of edges labeled ℓ.

In other words, the regular path query $(\ell\ell)^*$ cannot be expressed in Cypher. This was "folklore" knowledge, hitherto unproved, in a way similar to the folklore result that basic SQL cannot express the transitive closure query (which required many years of development of finite model theory to prove formally). It was largely due to this type of limitations that GQL chose to add repetitions of arbitrary patterns rather than just edges. With our formal treatment, we can now prove this result formally.

Proof. We first assume without loss of generality that no variable is optional, i.e., every node pattern uses one variable, every edge pattern uses three variables, and every Kleene star edge pattern uses two variables [1].

1: A modification to account for missing variables is straightforward but simply increases the number of easy base cases.

We now restrict our attention to simple graphs G_n with $N = \{v_1, \dots, v_n\}$ and $E = \{e_1, \dots, e_{n-1}\}$ so that $\mathrm{src}(e_i) = v_i$ and $\mathrm{tgt}(e_i) = v_{i+1}$ for each $i < n$ (i.e., directed paths), with each edge labelled ℓ, and no properties existing on edges or nodes. We can therefore assume that all edge labels used in patterns are ℓ (if not, such a pattern is not matched, and thus the entire subpattern in which it occurred cannot be matched, up to $+$ in the parse tree, and thus can be removed). We can further assume that all conditions are of the form $x = y$ where x, y are variables (any conditions involving properties can be replaced by false). We can also further assume that no variable is used as both a node variable and an edge variable (as this would falsify the pattern), nor any explicit equality between such variables is used in conditional patterns.

We represent such a graph G_n as a first-order structure S_n in the vocabulary R, R^* with the universe N, and relations interpreted as follows:

- $R = \{(v_i, v_{i+1}) \mid 1 \le i < n\}$ is the edge relation;
- $R^* = \{(v_i, v_j) \mid 1 \le i \le j \le n\}$ is the reflexive transitive closure of R.

We next show how patterns are translated into first-order formulae over this vocabulary. Notice that we use R for convenience only as it is definable from R^* which is isomorphic to a linear order on $\{1, \dots, n\}$. We will then easily obtain the inexpressibility results since FO cannot define even cardinality of linear orders, cf. [21].

[21]: Libkin (2004), *Elements of Finite Model Theory*

For the translation, with each pattern ψ we associate two new variables x^s_ψ, x^t_ψ (intuitively, to be witnessed by the endpoints of patterns),

and with each edge variable z used in a pattern we associte two first-order variables z^s, z^t to be used in FO formulas (for source and target of edges). Then a pattern ψ with node variables $y_1, \dots, y_m$ and edge variables $z_1, \dots, z_k$ (recall that they are all distinct) is translated into an FO formula

$$\alpha_\psi(x^s_\psi, x^t_\psi, y_1, \dots, y_m, z^s_1, z^t_1, \dots, z^s_k, z^t_k)\,.$$

The condition on the translation is that for a path $p = \mathtt{path}(u_0, f_0, \dots, u_r)$ we have

$$\begin{aligned}(p,\mu) \in [\![\psi]\!]_{G_n} \Leftrightarrow S_n \vDash \alpha_\psi\big(u_0, u_r, \mu(y_1), \dots, \mu(y_m),\\ \mathrm{src}(\mu(z_1)), \mathrm{tgt}(\mu(z_1)),\\ \dots,\\ \mathrm{src}(\mu(z_k)), \mathrm{tgt}(\mu(z_k))\big)\end{aligned} \tag{5.2}$$

Now suppose the pattern $(\ell\ell)^*$ is definable in Cypher over graphs G_n by a pattern ψ as above. Then $\beta(x^s_\psi, x^t_\psi) ::= \exists y_1, \dots, y_m, z^s_1, \dots, z^t_k\ \alpha_\psi$ is true for v_i, v_j iff the path between them is of even length and therefore the sentence $\gamma ::= \exists s, t\ \big(\neg\exists s'\ R(s', s) \wedge \neg\exists t'\ R(t, t') \wedge \beta(s, t)\big)$ states that the path from v_1 to v_n is of even length, which is impossible.

Next, to conclude the proof, we present the translation.

- If $\psi = (y)$ then $\alpha_\psi(x^s_\psi, x^t_\psi, y) ::= x^s_\psi = x^t_\psi \wedge x^t_\psi = y$.
- If $\psi = (y_1) \overset{z:\ell}{\rightarrow} (y_2)$ then

$$\begin{aligned}\alpha_\psi(x^s_\psi, x^t_\psi, y_1, y_2, z^s, z^t) ::= {} & x^s_\psi = y_1 \wedge x^t_\psi = y_2 \wedge z^s = y_1\\ & \wedge z^t = y_2 \wedge R(y_1, y_2).\end{aligned}$$

- If $\psi = (y_1) \overset{z:\ell}{\leftarrow} (y_2)$ then

$$\begin{aligned}\alpha_\psi(x^s_\psi, x^t_\psi, y_1, y_2, z^s, z^t) ::= {} & x^s_\psi = y_2 \wedge x^t_\psi = y_1 \wedge z^s = y_2\\ & \wedge z^t = y_1 \wedge R(y_2, y_1).\end{aligned}$$

- If $\psi = (y_1) \overset{:\ell^*}{\rightarrow} (y_2)$ then

$$\alpha_\psi(x^s_\psi, x^t_\psi, y_1, y_2) ::= x^s_\psi = y_1 \wedge x^t_\psi = y_2 \wedge R^*(y_1, y_2).$$

- if $\psi = (y_1) \overset{:\ell^*}{\leftarrow} (y_2)$ then

$$\alpha_\psi(x^s_\psi, x^t_\psi, y_1, y_2) ::= x^s_\psi = y_2 \wedge x^t_\psi = y_1 \wedge R^*(y_2, y_1).$$

- If $\psi = \psi_1\psi_2$ with ψ_1, ψ_2 translated as $\alpha_{\psi_1}(x^s_{\psi_1}, x^t_{\psi_1}, \overline{vars}_1)$ and $\alpha_{\psi_2}(x^s_{\psi_2}, x^t_{\psi_2}, \overline{vars}_2)$ respectively (where $\overline{vars}_i$ list variables in those formulae corresponding to node and edge variables in patterns), then $\alpha_\psi(x^s_\psi, x^t_\psi, \overline{vars}_1, \overline{vars}_2)$ is defined as

$$\begin{aligned}\exists x^s_{\psi_1}, x^t_{\psi_1}, x^s_{\psi_2}, x^t_{\psi_2}\ \Big(&\alpha_{\psi_1}(x^s_{\psi_1}, x^t_{\psi_1}, \overline{vars}_1) \wedge \alpha_{\psi_2}(x^s_{\psi_2}, x^t_{\psi_2}, \overline{vars}_2)\\ &\wedge x^s_\psi = x^s_{\psi_1} \wedge x^t_\psi = x^t_{\psi_2} \wedge x^t_{\psi_1} = x^s_{\psi_2}\Big)\end{aligned}$$

  where in $\overline{vars}_1, \overline{vars}_2$ repeated variables are mentioned only once.
- If $\psi = \psi_1 + \psi_2$ with ψ_1, ψ_2 translated as $\alpha_{\psi_1}(x^s_{\psi_1}, x^t_{\psi_1}, \overline{vars})$ and $\alpha_{\psi_2}(x^s_{\psi_2}, x^t_{\psi_2}, \overline{vars})$ (note that variables must be the same as the

schemas of ψ_1 and ψ_2 coincide), then

$$\alpha_\psi(x^s_\psi, x^t_\psi, \overline{vars}) \ ::= \ \alpha_{\psi_1}(x^s_\psi, x^t_\psi, \overline{vars}) \vee \alpha_{\psi_2}(x^s_\psi, x^t_\psi, \overline{vars})$$

- If $\psi = \psi_1\langle\theta\rangle$ then $\alpha_\psi(x^s_\psi, x^t_\psi, \overline{vars}) \ ::= \ \alpha_{\psi_1}(x^s_\psi, x^t_\psi, \overline{vars}) \wedge \theta'$ where θ' is obtained from θ by the following transformations:
  - each condition $y_i = y_j$ stays;
  - each condition $z_i = z_j$ is replaced by $z^s_i = z^s_j \wedge z^t_i = z^t_j$;
  - these are propagated through the Boolean connectives.

It is routine and straightforward to check that with these translations (5.2) holds, thus completing the proof. □

5.4 Case Study 2: Expressiveness of GQL and SQL/PGQ

Having examined the expressiveness of patterns, we now look at the entire query languages Core GQL and Core PGQ. Since the early days of graph query languages, the gold standard of language expressiveness was the class of queries with NLOGSPACE data complexity. Indeed, this is the complexity of the reachability problem, and already the very early language GraphLog [8], that gave rise to the ubiquitous notion of CRPQs, was shown to capture the transitive closure logic and thus NLOGSPACE (over ordered graphs). The idea behind GraphLog is that one can use CRPQs to define new relations, that can then be used in other CRPQs, just as idb relations can be used to define new idb relations in Datalog. Other Datalog-based languages for graph databases were developed, notably generalized regular queries [131], that extended Datalog with transitive closure restricted to binary predicates. These guaranteed decidability of static analyses, and importantly served as a precursor to GQL reachability patterns [97].

[8]: Consens et al. (1990), "GraphLog: a Visual Formalism for Real Life Recursion"

[131]: Vardi (2016), "A Theory of Regular Queries"

[97]: (2023), *GQL Influence Graph*

We thus ask ourselves whether Core GQL and Core PGQ capture all NLOGSPACE queries on graphs. The reason to think they might is that reachability is complete for NLOGSPACE via first-order reductions [132], and on top of patterns in these languages we have equivalents of first-order logic: RA and LCRA. The answer however is negative:

[132]: Immerman (1999), *Descriptive complexity*

Theorem 5.4.1

There are Datalog-expressible queries over property graphs that have DLOGSPACE data complexity, but are not expressible in Core GQL nor Core PGQ.

When we speak of queries over property graphs expressible in Datalog, we mean queries over property graphs $G = \langle N, E, \mathrm{lab}, \mathrm{src}, \mathrm{tgt}, \delta\rangle$ represented as relational structures with unary relations N and E, as well as unary relations N_ℓ and E_ℓ having ids of nodes and edges labelled ℓ, for each label ℓ present in G, as well as binary relations src and tgt which are subsets of $E \times N$, and for each $k \in \mathscr{K}$ present in G, a relation $P_k \subset (N \cup E) \times \mathrm{Const}$.

Our result shows that despite having patterns that possess at least the power of reachability and more generally RPQs, and having the power of Relational algebra to manipulate their results, we fall short of capturing even DLOGSPACE queries. It thus appears that the relational

querying of Core GQL and PGQ is somehow lacking in power. Looking at languages such as GraphLog, we can actually point out precisely the missing ingredient, which is *compositionality*. In GraphLog, an output of a query can be treated as a new edge and used in subsequent querying. Read-only GQL and PGQ lack this ability. Specifically, if we have a query Q, we cannot designate its projection on two columns containing node ids as a new edge that can be used in subsequent patterns in an LCRA expression.

To sketch the idea of the separating example, we define *data-less paths* as graphs G_n, $n > 0$, with $N = \{v_1, \dots, v_n\}$, $E = \{e_1, \dots, e_{n-1}\}$ so that $n \geq 0$, and $v_i \neq v_j \in \mathcal{N}, e_i \neq e_j \in \mathsf{E}$ for every $i \neq j$, $\mathrm{src}(e_i) = v_i$ and $\mathrm{tgt}(e_i) = v_{i+1}$ for each $i \in [1, n]$, and in addition $\mathrm{lab}(v_1) = \mathit{first}$ and $\mathrm{lab}(v_n) = \mathit{last}$. We then show that there is no Boolean Core GQL query that returns true on G_n iff n is a power of 2.

The proof of this is based on showing that GQL queries can only define Presburger properties of lengths of data-less paths, by translating Core GQL queries on such paths into formulae of Presburger Arithmetic (PA). As it is known that formulas of PA define semilinear sets (see e.g. [133]), the only definable properties of lengths are semilinear sets, while the set $\{2^k \mid k \in \mathbb{N}\}$ is not semilinear. On the other hand, checking whether the length of a path is a power of 2 can easily be done in DLOGSPACE and expressed in Datalog.

[133]: Ginsburg et al. (1966), "Semigroups, Presburger formulas, and languages."

Theorem 5.4.2

There is no Core GQL query Q such that for every data-less path $\mathbf{p}$ the following holds: $[\![\mathrm{Q}]\!]_{\mathbf{p}} = \mathbf{true}$ if and only if $\mathrm{len}(\mathbf{p}) = 2^n, n \in \mathbb{N}$

For a data-less path $\mathbf{p}_\epsilon$, we define the function pos that maps each node in $\mathbf{p}_\epsilon$ to its position (where the position of the first node in $\mathbf{p}_\epsilon$ is 1) and each edge in $\mathbf{p}_\epsilon$ to the tagged position of its source node. We do so to distinguish whether a position correspond to a node or an edge. For example, if $\mathbf{p}_\epsilon$ is $(n_1) \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} (n_k)$ then $\mathrm{pos}(n_j) = j$ and $\mathrm{pos}(e_j) = j'$. We treat tagged integers as integers wrt to arithmetics and equality.

Before proving the translation from path patterns to Presburger Arithmetics (PA), we first show that path patterns without variables can be translated to automata.

Lemma 5.4.3

For every path pattern ψ with no variables, there exists a finite automaton A_ψ such that

$$o_1 \cdots o_{m-1} \in \mathscr{L}(A_\psi) \text{ iff } ((n_1, e_1, \dots, n_{m-1}, e_{m-1}, n_m), \varnothing) \in [\![\psi]\!]_{\mathbf{p}_\epsilon}$$

where $o_j ::= a$ if e_j is a forward edge (from n_{j-1} to n_j) and $o_j ::= b$ if e_j is a backward edge (from n_j to n_{j-1}).

Proof. We use the standard definition of finite automata as presented in Chapter 2. We fix the input alphabet to be $\{a, b\}$, and prove the claim by induction on ψ's structure.

The intuition behind the construction is that the a's represent the forward edges and the b's the backward edges in a path that matches ψ. As a convention, q_0 and q_f denote the initial and final states, respectively.

Base cases.

- $\psi = ()$ then $\delta(q_0, \epsilon) = q_f$.
- $\psi = () \rightarrow ()$ then $\delta(q_0, a) = q_f$.
- $\psi = () \leftarrow ()$ then $\delta(q_0, b) = q_f$.

Induction step For $\psi = \psi_1\psi_2$, $\psi = \psi'^*$ and $\psi = \psi_1 + \psi_2$ we use similar construction to Thompson construction by adding epsilon transitions and redefining the initial and accepting states. It is straightforward to show that this construction satisfies the connection stated in the lemma. □

Corollary 5.4.4

For every path pattern ψ without variables, and every data-less path $\mathbf{p}_\epsilon$ the set

$$\{\text{pos}(n_m) - \text{pos}(n_1) \mid (p, \varnothing) \in [\![\psi]\!]_{\mathbf{p}_\epsilon}, p ::= (n_1, e_1, \ldots, e_{m-1}, n_m)\}$$

is semi-linear.

Proof. The proof follows from Lemma 5.4.3, the fact that the Parikh image of a finite automaton is semilinear, and the closure of semilinear sets under subtraction. □

We obtain the following connection:

Lemma 5.4.5

For every data-less path $\mathbf{p}_\epsilon$, and path pattern ψ there is a PA formula $\varphi_\psi(x_s, x_1, \ldots, x_m, x_t)$ where $\text{sch}(\psi) ::= \{x_1, \ldots, x_m\}$ s.t.

$$\varphi_\psi(i_s, i_1, \ldots, i_m, i_t) \text{ if and only if } \exists p : (p, \mu) \in [\![\psi]\!]_{\mathbf{p}_\epsilon}$$

where $i_j = \text{pos}(\mu(x_j))$ for every j, and i_s, i_t are the positions of the first and last nodes of p in $\mathbf{p}_\epsilon$, respectively.

Proof. We simplify ψ: Since we deal with data-less paths, it is straightforward that any $\psi_{\langle\theta\rangle}$ can be rewritten to ψ while preserving the semantics (on data-less paths). Formally, for every data-less path P and for every condition θ, it holds that $[\![\psi]\!]_P = \left[\!\left[\psi_{\langle\theta\rangle}\right]\!\right]_P$. In addition, we can assume that every repetition $\psi^{n..m}$ where $m < \infty$ can be rewritten to $\cup_{i=n}^{m} \psi_i$. This is true for any input (no need to restrict to data-less paths) due to the semantics definition. For repetition $\psi^{n..m}$ with $m = \infty$, we can rewrite the pattern and obtain $\psi^n\psi^{0..\infty}$. Also here, equivalence holds for any input (no need to restrict to data-less paths) due to the semantics definition. Recall that now we focus only on path patterns in which we disallow having variables on the edges.

Under the above assumptions on the form of ψ, we define φ_ψ inductively as follows:

- If $\psi ::= (x)$ then $\varphi_\psi(x_s, x, x_t) ::= x_s = x \wedge x = x_t$;
- If $\psi ::= ()$ then $\varphi_\psi(x_s, x_t) ::= x_s = x_t$;
- If $\psi ::= (x) \rightarrow (y)$ then
$$\varphi_\psi(x_s, x, y, x_t) ::= y = x + 1 \wedge x = x_s \wedge y = y_s$$
- If $\psi ::= (x) \leftarrow (y)$ then
$$\varphi_\psi(x_s, x, y, x_t) ::= x = y + 1 \wedge x = x_s \wedge y = y_s$$
- If $\psi ::= \xrightarrow{x}$ then $\varphi_\psi(x_s, x, x_t) ::= x_t = x_s + 1 \wedge x = x_s$
- If $\psi ::= \rightarrow$ then $\varphi_\psi(x_s, x_t) ::= x_t = x_s + 1$
- If $\psi ::= \psi_1\psi_2$ then
$$\varphi_\psi(x_s, \bar{z}, y_t) ::= \varphi_{\psi_1}(x_s, \bar{x}, x_t) \wedge \varphi_{\psi_2}(y_s, \bar{y}, y_t) \wedge x_t = y_s$$
where $\bar{z}$ is the union $\bar{x} \cup \bar{y}$.
- If $\psi ::= \psi_1 + \psi_2$ then
$$\varphi_\psi(x_s, \bar{x}, x_t) ::= \varphi_{\psi_1}(x_s, \bar{x}, x_t) \vee \varphi_{\psi_2}(x_s, \bar{x}, x_t)$$
- If $\psi ::= \psi_1^n$ then
$$\begin{aligned}\varphi_\psi(x_s, x_t) =& \exists x_1, y_1, \bar{z}_1 \ldots, x_n, y_n, \bar{z}_n, \\ & \bigwedge_{i=1}^{n} \varphi_{\psi_1}(x_i, \bar{z}_i, y_i) \bigwedge_{i=1}^{n-1} y_i = x_{i+1} \\ & \wedge x_1 = x_s \wedge y_n = x_t\end{aligned}$$
- If $\psi ::= \psi_1^*$ then, due to Corollary 5.4.4, there is a PA formula $\psi_{\psi_1^*}(m)$ for $\{|p| \mid (p, \varnothing) \in [\![\psi_1^*]\!]_\pm\}$. Therefore,
$$\varphi_\psi(x_s, x_t) ::= \exists m : x_t = x_s + \psi_{\psi_1^*}(m)$$

Showing that the condition in the Lemma holds for φ_ψ is straightforward from the definition. □

Lemma 5.4.6

For every data-less path $\mathbf{p}_\epsilon$, and path pattern with output ψ_Ω with variables only on nodes, there is a PA formula $\varphi_\psi(x_1, \ldots, x_m)$ where $\Omega ::= \{x_1, \ldots, x_m\}$ such that

$$\varphi_{\psi_\Omega}(i_1, \ldots, i_m) \text{ if and only if } \mu \in [\![\psi]\!]_{\mathbf{p}_\epsilon}$$

where $i_j = \text{pos}(\mu(x_j))$ for every j.

Proof. Notice that since we are dealing with data-less paths as input, all elements in Ω are variables. Hence, the proof is implied directly from Lemma 5.4.5. In particular, by existentially quantifying over variables that are omitted in Ω. For example,

if $\varphi_\psi(x_s, \bar{x}, \bar{y}, x_t)$ then $\varphi_{\psi_{\bar{y}}}(\bar{y}) = \exists x_s, \bar{x}, x_t : \varphi_\psi(x_s, \bar{x}, \bar{y}, x_t)$

if $\varphi_\psi(x_s, \bar{x}, \bar{z}, \bar{y}, x_t)$ then $\varphi_{\psi_{\bar{y}\bar{x}}}(\bar{y}, \bar{x}) = \exists x_s, \bar{z}, x_t : \varphi_\psi(x_s, \bar{x}, \bar{z}, \bar{y}, x_t)$ □

Lemma 5.4.7

For every data-less path $\mathbf{p}_\epsilon$, and Core GQL query Q, there is a PA formula $\varphi_Q(x_1, \dots, x_m)$ such that $\text{attr}(Q) = \{x_1, \dots, x_m\}$ and

$$\varphi_Q(i_1, \dots, i_m) \text{ if and only if } \mu \in [\![Q]\!]_{\mathbf{p}_\epsilon}$$

where $i_j = \text{pos}(\mu(x_j))$ for every j.

Proof. The proof is by induction on the structure of Q. The base case of path pattern with output is covered by the previous lemma. For the induction step, we distinguish between the form of Q.

- If $Q ::= \psi_{\mathbf{A}}(Q')$ then $\varphi_Q(\bar{z}) ::= \exists \bar{y}\, \varphi_{Q'}(x_1, \dots, x_m)$ where $\bar{y} = \mathbf{A} \setminus \{x_1, \dots, x_m\}$ and $\bar{z} = \mathbf{A} \cap \{x_1, \dots, x_m\}$.
- If $Q ::= \sigma_\theta(Q')$ then $\varphi_Q(\bar{x}) ::= \varphi_{Q'}(\bar{x})$. Notice that the correctness here follows from the fact the input is a data-less path.
- If $Q ::= \rho_{A \to A'}(Q')$ then $\varphi_Q(\bar{x}, A, \bar{y}) ::= \varphi_{Q'}(\bar{x}, A', \bar{y})$.
- If $Q ::= Q_1 \times Q_2$ then $\varphi_Q(\bar{x}) ::= \varphi_{Q_1}(\bar{z}) \wedge \varphi_{Q_2}(\bar{y})$ with $\bar{x}$ the union of $\bar{z}$ and $\bar{y}$.
- If $Q ::= Q_1 \cup Q_2$ then $\varphi_Q(\bar{x}) ::= \varphi_{Q_1}(\bar{x}) \vee \varphi_{Q_2}(\bar{x})$.
- If $Q ::= Q_1 \cap Q_2$ then $\varphi_Q(\bar{x}) ::= \varphi_{Q_1}(\bar{x}) \wedge \varphi_{Q_2}(\bar{x})$.
- If $Q ::= Q_1 \setminus Q_2$ then $\varphi_Q(\bar{x}) ::= \varphi_{Q_1}(\bar{x}) \wedge \neg\varphi_{Q_2}(\bar{x})$.

Showing that the condition holds follows from the definition. □

We can now prove Theorem 5.4.2.

Proof. Let us assume by contradiction that there is a Boolean GQL query Q for which $[\![Q]\!]_{\mathbf{p}_\epsilon} \neq \emptyset$ if and only if $\text{len}(\mathbf{p}_\epsilon) = 2^n$, $n \in \mathbb{N}$.

Let us define a new query Q′ that binds x to the last node of a data-less path. We set $Q' ::= \psi_1 \setminus \psi_2$ where $\psi_1 ::= () \to^* (x)$ and $\psi_2 ::= () \to^* (x) \to ()$. It holds that $[\![Q']\!]_{\mathbf{p}_\epsilon} ::= \{\mu \mid \text{dom}(\mu) = \{x\}, \mu(x) = n_k\}$ where n_k is the last node in $\mathbf{p}_\epsilon$. Now let us consider $Q \times Q'$. It can be easily shown that $[\![\psi_x(Q \times Q')]\!]_{\mathbf{p}_\epsilon} \neq \emptyset$ iff $[\![Q]\!]_{\mathbf{p}_\epsilon} \neq \emptyset$. By previous lemma, $[\![\psi_x(Q \times Q')]\!]_{\mathbf{p}_\epsilon}$ is expressible by a PA formula. Nevertheless, by our assumption, $[\![\psi_x(Q \times Q')]\!]_{\mathbf{p}_\epsilon}$ is exactly the set $\{2^n \mid n \in \mathbb{N}\}$ which leads to the desired contradiction. □

5.4.1 Datalog on Graphs

We use the standard definition of Datalog, as introduced in Chapter 2.

As before, we view property graphs $G ::= \langle N, E, \text{lab}, \text{src}, \text{tgt}, \delta \rangle$ as relational structures $\mathscr{D}_G$ over the schema consisting of relation symbols $N, E, \text{lab}, \text{src}, \text{tgt}, \delta$ with the straightforward interpretation.

Example 5.4.8

The Datalog program defined by the rules:

$$\text{eqLen}(x, y, z, w) \leftarrow E(x, y), E(z, w)$$
$$\text{eqLen}(x, y, z, w) \leftarrow \text{eqLen}(x, y', z, w'), E(y', y), E(w'w)$$

extracts from a bounded data-less path a mapping μ of x, y, z, w such that the number of edges between $\mu(x)$ and $\mu(y)$ equals to that between $\mu(z)$ and $\mu(w)$.

Example 5.4.9

The Datalog program given by the rules

$$\begin{aligned} \mathsf{len}_{2^n}(x,y) &\leftarrow E(x,z), E(z,y) \\ \mathsf{len}_{2^n}(x,y) &\leftarrow \mathsf{len}_{2^n}(x,z), \mathsf{len}_{2^n}(w,y), \mathsf{eqLen}(x,z,w,y) \\ \mathsf{Ans}() &\leftarrow \mathsf{len}_{2^n}(x,y), \mathsf{lab}(x,\mathsf{first}), \mathsf{lab}(y,\mathsf{last}) \end{aligned}$$

outputs **true** if and only if the length of the input bounded data-less path is 2^n for some $n \in \mathbb{N}$.

Using a straightforward induction, we can show that:

Lemma 5.4.10

For every Datalog program P and data-less paths p, p', if $p \sim p'$ then $[\![P]\!]_{\mathscr{D}_{p'}} = [\![P]\!]_{\mathscr{D}_p}$.

We denote $\mathscr{D}_p$ by the $\mathscr{D}_n$ where n is the length of p. We can view Boolean Datalog programs as queries on the quotient space $P/\sim$ and define

$$\mathsf{len}(P) \ ::= \ \{n \mid [\![P]\!]_{\mathscr{D}_n} = \textbf{true}\}.$$

We can conclude:

Corollary 5.4.11

Datalog over property graphs is strictly more expressive than GQL. There is a Datalog over property graphs program such that for every data-less path $\mathbf{p}$ the following holds:

$$[\![Q]\!]_{\mathbf{p}} = \textbf{true} \text{ if and only if } \mathsf{len}(\mathbf{p}) = 2^n, n \in \mathbb{N}$$

The proof of Theorem 5.4.1 then follows directly from Theorem 5.4.2 and Corollary 5.4.11.

5.5 Conclusions and future work

We had a dual goal: to formalize recently standardized graph query languages in a simple way that makes them amenable to a theoretical investigation, and to analyze their expressive power, trying to identify notable holes in the language design. Towards the first goal, we also formalized a pipelined, or linear, presentation of Relational Algebra, having a distinctly different flavor, preferred in languages such as Cypher and GQL. With respect to the expressive power, we classify our results into three groups, and discuss their implications on the design of graph languages.

- First, we have shown that some very natural patterns cannot be expressed in a reasonable way in the pattern language of GQL and PGQ. Specifically, analyzing data in nodes is easy while analyzing data in edges is not. The current way in which GQL and PGQ deal with this involves highly inefficient queries to express simple patterns. The problem is recognized by language designers who seek new solutions for the second version of the standard. Our formal proof pinpoints exactly the main deficiency: limited capabilities of joining patterns under repetitions.
- Second, we looked at the language as a whole, and noticed that even though it expresses an NLOGSPACE-complete problem (under first-order reductions) and possesses the full power of FO, it fails to express all NLOGSPACE (and even DLOGSPACE) properties. The reason behind this discrepancy is poor compositional capabilities of GQL and PGQ that stem directly from their syntactic design.
- Third, we proved a folklore result that Cypher patterns fall short of RPQs. This was widely believed, to the point of GQL and PGQ expanding Cypher patterns to capture all RPQs. A formal proof of this result is akin to a formal proof of inexpressibility of transitive closure in SQL (cf. [21, 132]) motivating the addition of recursive queries to the standard.

[21]: Libkin (2004), *Elements of Finite Model Theory*
[132]: Immerman (1999), *Descriptive complexity*

Our immediate challenge in the future is to plug these gaps in expressive power, by addressing the underlying issues in the language design. This will be much easier to do initially on our simple formalizations of the languages, rather than on their full descriptions that run hundreds of pages.

Our next challenge is to expand our formalization to include other aspects of the language, and study their properties. These aspects include bag semantics, aggregation, different path modes, and forming outputs more complex than 1NF relations.

References

[6] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. "A Graphical Query Language Supporting Recursion". In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. Ed. by Umeshwar Dayal and Irving L. Traiger. ACM Press, 1987, pp. 323–330. DOI: 10.1145/38713.38749.

[8] Mariano P. Consens and Alberto O. Mendelzon. "GraphLog: a Visual Formalism for Real Life Recursion". In: *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. ACM Press, 1990, pp. 404–416. DOI: 10.1145/298514.298591.

[9] Pablo Barceló. "Querying graph databases". In: *Principles of Database Systems (PODS)*. 2013, pp. 175–188.

[11] Pablo Barceló et al. "Expressive languages for path queries over graph-structured data". In: *ACM Trans. Database Syst.* 37.4 (2012), 31:1–31:46.

[15] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

[19] Marcelo Arenas et al. *Database Theory*. Open source at https://github.com/pdm-book/community, 2022.

[21] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[60] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. "Querying Graphs with Data". In: *Journal of the ACM* 63.2 (2016), 14:1–14:53.

[76] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. "Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard". In: *World Wide Web (WWW)*. 2012, pp. 629–638.

[77] Katja Losemann and Wim Martens. "The complexity of regular expressions and property paths in SPARQL". In: *ACM Trans. Database Syst.* 38.4 (2013), p. 24.

[96] Alin Deutsch et al. "Graph Pattern Matching in GQL and SQL/PGQ". In: *SIGMOD*. ACM, 2022, pp. 1–12.

[97] *GQL Influence Graph*. https://www.gqlstandards.org/existing-languages. Accessed: 2023-01-17. 2023.

[100] Diego Calvanese et al. "Containment of Conjunctive Regular Path Queries with Inverse". In: *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000*. 2000, pp. 176–185.

[102] Nadime Francis et al. "GPC: A Pattern Calculus for Property Graphs". In: *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*. Ed. by Floris Geerts, Hung Q. Ngo, and Stavros Sintos. ACM, 2023, pp. 241–250. DOI: 10.1145/3584372.3588662.

[114] Renzo Angles et al. "Foundations of Modern Query Languages for Graph Databases". In: *ACM Comput. Surv.* 50.5 (2017), 68:1–68:40.

[120] Albert Atserias, Martin Grohe, and Dániel Marx. "Size Bounds and Query Plans for Relational Joins". In: *SIAM J. Comput.* 42.4 (2013), pp. 1737–1767. DOI: 10.1137/110859440.

[121] Hung Q. Ngo et al. "Worst-case Optimal Join Algorithms". In: *J. ACM* 65.3 (2018), 16:1–16:40. DOI: 10.1145/3180143.

[122] Maarten Marx. "Navigation in XML Trees". In: *Bull. EATCS* 88 (2006), pp. 126–140.

[123] Wim Martens et al. "Expressiveness and complexity of XML Schema". In: *ACM Trans. Database Syst.* 31.3 (2006), pp. 770–813. DOI: 10.1145/1166074.1166076.

[124] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and Complexity of SPARQL". In: *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*. Ed. by Isabel F. Cruz et al. Vol. 4273. Lecture Notes in Computer Science. Springer, 2006, pp. 30–43. DOI: 10.1007/11926078_3.

[125] Sergio Abriola et al. "Bisimulations on Data Graphs". In: *J. Artif. Intell. Res.* 61 (2018), pp. 171–213. DOI: 10.1613/JAIR.5637.

[126] Chandan Sharma, Roopak Sinha, and Kenneth Johnson. "Practical and comprehensive formalisms for modelling contemporary graph query languages". In: *Inf. Syst.* 102 (2021), p. 101816. DOI: 10.1016/J.IS.2021.101816.

[127] Nadime Francis et al. “A Researcher’s Digest of GQL”. In: *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*. Ed. by Floris Geerts and Brecht Vandevoort. Vol. 255. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 1:1–1:22. DOI: 10.4230/LIPICS.ICDT.2023.1.

[128] PRQL. *Pipelined Relational Query Language*. 2024. URL: https://prql-lang.org.

[129] Hadley Wickham et al. *dplyr: A Grammar of Data Manipulation*. 2023. URL: https://dplyr.tidyverse.org.

[130] Pavol Hell and Jaroslav Nešetřil. *Graphs and homomorphisms*. Oxford University Press, 2004.

[131] Moshe Y. Vardi. “A Theory of Regular Queries”. In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 2016, pp. 1–9. DOI: 10.1145/2902251.2902305.

[132] Neil Immerman. *Descriptive complexity*. Springer, 1999.

[133] Seymour Ginsburg and Edwin H. Spanier. “Semigroups, Presburger formulas, and languages.” In: *Pacific Journal of Mathematics* 16 (1966), pp. 285–296.

6 Lists and Graphs languages

This chapter is joint work with Amélie Gheerbrant and Leonid Libkin.

In the last three chapters, we focused on the *core* of GQL and SQL/PGQ, or more generally of pattern matching. But practical languages are, of course, much richer. One functionality included in virtually all (graph) query languages is aggregation. In fact, even GQL includes a minimal set of list functions in its first version.

Lists were first added in Cypher to circumvent its inability to express all RPQs: as shown in Theorem 5.3.12, applying repetition onto labels instead of general patterns means the resulting language is not closed under repetition and cannot express queries as simple as $(aa)^*$[1]. For the same reason, Cypher pattern matching is not powerful enough to express extensions of RPQs such as UC2RPQs and ECRPQs.

1: This feature was made available in Cypher pattern matching in Neo4j version 5.9

Another Cypher restriction is that variables are not allowed in a repeated pattern and so cannot be referred to in conditions. Because of this, queries like finding a path along which all node values are equal are not expressible.

There is a common theme among these queries. Imagine that for a given path $\mathtt{p} = x_1\ e_1\ x_2\ \ldots e_{n-1}\ x_n$, we have collected two lists:

- `nodes(p)` $= [x_1, \ldots, x_n]$ of all nodes of `p`, and
- `relationships(p)` $= [e_1, \ldots, e_{n-1}]$ of all edges of `p`,

both in the order in which they appear in `p`. Then the queries such as the ones above are easy: to check that `p` conforms to $(aa)^*$ we need to check that every label is a (by `-[:a*]->`) and that `-[:a*]->` is odd (or `length(relationships(p))` is even). To check that the node values are the same, we need to verify that the values of all nodes in `nodes(p)` are equal to the value of its first element.

Combining list functions and pattern matching brings the expressive power of Cypher to the level of RPQs and can also be used to emulate UC2RPQs, RDPQs and ECRPQs.

But lists and their functions are powerful tools. Already in the relational context, adding polynomial time predicates, functions and aggregates, brings the combined complexity of query evaluation from PSPACE to EXPSPACE and its data complexity from AC^0 to PTime [19]. This phenomenon also occurs in the graph context. For example, one can check for the existence of a Hamiltonian path in the graph, a known NP problem [134], by simply verifying that there exists a path `p` for which `relationships(p)` is equal to the list of all edges of the graph.

[19]: Arenas et al. (2022), *Database Theory*

[134]: Garey et al. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*

One might however object that the complexity of query evaluation for Cypher pattern matching is already NP-hard because of its use of trail semantics for paths. Despite this theoretical intractability, Neo4j tends to work quite well in practice, so the natural question is: does the theoretical intractability caused by lists have a significant negative impact on actual performance?

We show experimentally that the addition of lists does cause an important decrease in performance. Pure pattern matching Cypher queries do reasonably well even if they are theoretically NP-hard. For example,

matching a trail path whose labels form a word in A^*BA^* is known to be NP-hard [84] and yet the engine performs well on this query. On the other hand, queries such as those listed above, perform very poorly. We tested them on random graphs, and for edge probability > 0.2, they time out already on graphs with fewer than 10 nodes! While this may sound dramatic, recall that the number of simple paths in a graph grows exponentially with the number of nodes, reaching in the worst case $(N-2)!$. Query plans that use lists, especially for filtering out matched paths, are bound therefore to generate a huge number of possible paths.

[84]: Martens et al. (2020), "A Trichotomy for Regular Trail Queries"

There are still questions remaining. In NP-hard queries such as Hamiltonian path we rely heavily on the default trail semantics of Cypher. Could it still be that the trail semantics is somehow the main culprit, when used together with list processing? We eliminate this hypothesis by forcing the shortest path semantics, which is an option in Cypher. We do this with the subset sum query, and see that even without trails, the results are similar: queries time out on very small graphs. In fact the size of graphs on which we see timeouts is so small that we could only reasonably do experiments on synthetic data, as real data with such super small graphs is hardly in existence.

Our last technical question is whether SQL, especially with recursive CTEs, would perform better than a native graph database such as Neo4j on these problematic queries. Recursive queries give SQL enough power to express them. Our first observation is that SQL handles the theoretically NP-hard query of matching A^*BA^* paths about as well as Cypher does, just marginally worse. On queries such as Hamiltonian path and subset sum, SQL performs slightly better than Cypher, but still cannot handle even modest sized graphs.

These performance results are the consequences of putting list operations in a graph query language rather than a particular implementation of it. Indeed, in the absence of useful optimizations that can be applied, they force the engine to build a very large number of paths, rendering queries completely impractical. In the context of very active work on GQL 2.0 as well as the new version of SQL/PGQ, this leads to the question of how list operations should be managed. With the next version of the standards expected to be released in about five years, what lessons can be learned from the point of view of language design? It turns out that we can summarize rather simply (before dedicating the last section of the chapter to it) what is wrong with using lists in a graph query language:

- It is ok to use lists to post-process results of pattern matching;
- It is not, however, advisable, to use expressions with list operations to *filter* the set of selected paths.

These recommendations have the advantage of being easily adopted by query languages, by imposing syntactic restrictions on where and how list operations can occur.

Related work

Relational algebra with aggregation was studied by L. Libkin and L. Wong in [135] where it was shown that this extension is not powerful enough to encode arbitrary transitive closure and recursion. As transitive closure is a basic component of graph languages, this result cannot be applied to our context.

[135]: Libkin et al. (1997), "On the Power of Aggregation in Relational Query Languages"

The generic list operator *fold* was studied in the programming languages community and found to be able to encode all primitive recursive functions in the general case, and Ackermann's function in higher-order languages [136]. While also not directly applicable to Cypher pattern matching, as its does not include functions (in the sense of a named piece of code that can be referenced later), these results give an idea of how dangerous the fold operator can be.

[136]: Hutton (1999), "A tutorial on the universality and expressiveness of fold"

Contributions

We give a brief overview of the main operations of Cypher in Section 6.1. Then in Section 6.2 we show some shortcomings of Cypher, which motivated the introduction of lists. In Section 6.3 we show how lists help express many useful queries, particularly RPQs and their extensions. Section 6.4 demonstrates that the same list facilities lead to problems, namely expressing computationally intractable queries. In Section 6.5 we provide an experimental evaluation of such highly intractable queries, and show that it is list operations, rather than the trail semantics, that are responsible for the poor performance. In Section 6.6 we compare a native graph implementation with SQL. Finally, in Section 6.7 we discuss what it means for language design, especially for future enhancements to GQL. Supplementary material (full code and results) can be found here.

6.1 Cypher Pattern Matching

The main selling points of Neo4j are its native graph storage, which allows for optimized traversal of nodes and edges, and its associated declarative query language Cypher [15], which shares a lot of features with SQL and was the main inspiration behind GQL. This familiar look and feel makes it easy to learn for most users. As a matter of fact, since 2015 and the launch of the openCypher project, Cypher has been implemented by a number of different vendors such as Amazon, Memgraph and Tigergraph.

[15]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

Just like GQL and SQL/PGQ, Neo4j uses the property graph model. Figure 6.1 represents a property graph with five nodes, four of them labeled `Person` and having attribute `name` and one labeled `Club` with an attribute `type` equal to `Cycling` and an attribute `name` equal to `Springfield Cycle Gang`. The graph also contains six edges, all oriented, four with label `Friend` and two with label `Member`. Each node and edge also carries a unique identifier (`n1, e1,` and so on).

Like in GQL, queries in Cypher are composed in a linear way. They take as input a graph and usually end with a `RETURN` statement, outputting a table.

An example is the following query, which returns the names of friends of Milhouse who have at least one but no more than three other friends who belong to a cycling club :

```
MATCH (p1:Person)-[:Friend]-(p2:Person),
      (p2)-[:Friend]-(p3:Person),
      (p3)-[:Member]->(c:Club)
WHERE p1.name="Milhouse" AND c.type="Cycling"
WITH p2, COUNT(DISTINCT p3) AS fof
WHERE fof <= 3 AND fof >= 1
RETURN p2.name
```

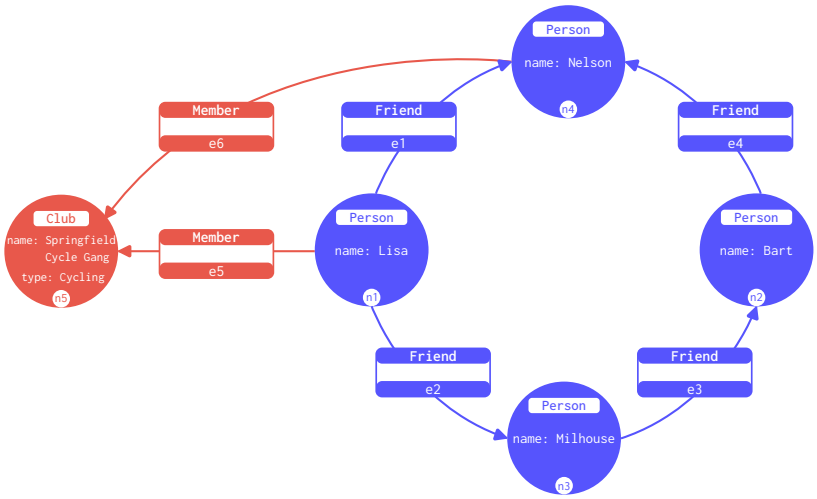


Figure 6.1: A cycling property graph

The core of the Cypher language is its pattern matching mechanism. Basic building blocks are patterns, which have a user friendly "ASCII art" flavor, as witnessed by the three patterns in the `MATCH` clause above. The first and second connect three nodes labeled `Person` using two `Friend` edges. They are joined together via the variable `p2`. The third pattern, starting on line 3, is joined to the second one via the third `Person` node using the variable `p3` and connects it to a node labeled `Club` via a `Member` edge. Variables `p1`, `p2`, `p3` and `c` are used to bind matches in the working table. Values are first filtered by the `WHERE` clause (checking `p1`'s name and c's type), resulting in the following working table.

p1	p2	p3	c
n3	n2	n4	n5
n3	n1	n4	n5

The subsequent `WITH` clause modifies the table, by retaining only matches for `p2`, and computing, for each of the values `p2`, the number of distinct values of `p3` that occur with it in the tuple (and binding it to attribute `fof`). This results in the modified working table shown below.

p2	fof
n2	1
n1	1

Finally, the `RETURN` clause acts similarly to SQL's `SELECT`, forming the output; in our example the value of the attribute `name` of the node matched to `p2` will be output, which corresponds to Bart and Lisa for the graph in Figure 6.1.

Notice that the orientation of both `Friend` edge patterns is not specified in the query and so the path can traverse them in *any direction*, thus creating a first solution traversing the nodes corresponding to Milhouse then Bart then Nelson, and a second solution going from Milhouse to Lisa to Nelson. As `p3` is Nelson in both cases, the value of `fof` stays `1`.

An important aspect of Cypher pattern matching is the enforced trail semantics, meaning each edge can be traversed at most once per path. Thanks to this rule, finiteness of matches is always guaranteed. For example if the condition on `p1` were to be changed to `p1.name='Lisa'`, there would be no valid answer as the only `Person` at distance 2 from Lisa who is also a Member of a cycling club is Lisa herself and such a path would necessarily use the same edge (between either Lisa and

Milhouse or Lisa and Nelson) twice, even if in opposite directions.

It is a crucial feature of any reasonable graph query language to allow the retrieving of paths of arbitrary length (which falls short of the expressive power of usual conjunctive queries). Cypher enables this via a restricted form of Kleene star. This is implemented by the use of *variable length edge patterns* inside patterns, indicating that some possibly unbounded number of edges should be traversed. The most basic form of such patterns is `()-[*]-()`, which can be matched to any path (once again with no repeated edges). Minimal and maximal length of paths to be matched can be set, as well as admissible labels for edges, as in `()-[:l1|l2*2..3]-()`, where matches will be restricted to paths of length 2 to 3 where edges are only labelled with `l1` or `l2`. However, it is not possible to directly label the path with a regular expression.

This sets Cypher apart from regular path queries, but also from GQL (which can express full RPQs, as shown in Chapter 4). As a downside, there is no commercial implementation of GQL yet, and thus we focus our analysis on Cypher.

6.2 Adding Lists

6.2.1 Cypher limitations

As seen in Chapter 5, Cypher has a number of limitations that lead to the need for a more expressive language. Some of these limitations have been fixed in GQL and SQL/PGQ but others remain.

To start with, Cypher as originally designed could not express all RPQs. Recall that an RPQ is given by a regular expression e over the alphabet of edge labels. Such a query returns pairs of nodes connected by a path of edges whose labels, when read along the path, form a word in the language of e. Cypher can express RPQs such as a^* directly by `(x)-[:a*]->(y)`; it can also express some more complex expressions such as a^*b^* by combining patterns such as `(x)-[:a*]->()-[:b*]->(y)`. However the main limitation of Cypher patterns is that the Kleene star $*$ can only be applied to (disjunctions of) edge labels, and not to more complex regular expressions. This renders even simple regular expressions such as $(aa)^*$ inexpressible with Cypher's basic pattern matching mechanism. We note that this limitation led to more expressive GQL pattern matching capabilities, that have been adopted by Cypher pattern matching in Neo4j starting with version 5.9.

Similarly, more complex path queries are not definable with Cypher's basic pattern matching. These include (a) CRPQs [137], or joins of RPQs, (b) ECRPQ [11], which allow path comparisons with regular predicates (such as: lengths of paths p_1 and p_2 are the same, or the label of p_1 is a prefix of the label of p_2, etc), (c) various extensions of CRPQs with handling data, such as checking for equality of property values in nodes or edges of matched paths [60].

[137]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"

[60]: Libkin et al. (2016), "Querying Graphs with Data"

There are other rather natural conditions on paths that cannot be expressed, neither in the original Cypher nor in GQL and SQL/PGQ. Recall that a path in a property graph is an alternating sequence of nodes and edges that starts and ends in a node [137]. That is, such a path is a sequence $p = n_1\ e_1\ n_2\ e_2 \ldots e_{m-1}\ n_m$ where each n_i is a node and e_j is an edge connecting n_j and n_{j+1} (meaning it either goes from n_j to n_{j+1} or from n_{j+1} to n_j or is an undirected edge between them). Assume

[137]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

now that each node has a property k and each edge has a property s. Consider the following properties of path p:

1. Values in nodes increase: $n_1.k < n_2.k < \dots n_m.k$;
2. Values in edges increase: $e_1.s < e_2.s < \dots e_{m-1}.s$;
3. Values in all nodes are different: $n_i.k \neq n_j.k$ for $1 \leq i < j \leq m$;
4. Values in all edges are different: $e_i.s \neq e_j.s$ for $1 \leq i < j < m$;
5. Values in all nodes/edges are similar: $|n_i.k - n_j.k| < t$ for $1 \leq i < j \leq m$ and some threshold t, and likewise for edges.

Of these, only the first one can be expressed by a simple pattern (available in GQL and the latest version of Cypher):

```
MATCH (x) ((n1)->(n2) WHERE n1.k<n2.k)+ (y)
RETURN x, y
```

Others cannot be expressed in pattern matching only (as shown in Chapter 5).

While these fairly simple properties are not expressible by patterns alone, there is a seemingly natural way to add them to the language. This was followed by Cypher that introduced the following capabilities. Given a path $\mathtt{p} = n_1\ e_1\ n_2\ \dots e_{m-1}\ n_m$, one can define two lists:

- `nodes(p)` $= [n_1, \dots, n_m]$ of all nodes of `p`, and
- `relationships(p)` $= [e_1, \dots, e_{n-1}]$ of all edges of `p`,

both in the order in which they appear in `p`.

Then the above queries are easy with some standard list functions. Checking that `p` conforms to $(aa)^*$ we need to check that every label is a (using for example `-[:a*]->`) and that `size(nodes(p))` is odd (or `size(relationships(p))` is even). For other conditions, we apply the standard `reduce` (or `fold`) function on lists that accumulates a value as it iterates over list elements:

$$\mathtt{reduce}[\iota, f]([a_1, \dots, a_n] = f(\dots f(f(\iota, a_1), a_2), \dots, a_n)$$

For example, to check whether a non-negative list $[a_1, \dots, a_n]$ is in the increasing order, we use $\iota = (0, \text{true})$ and $f\big((a, \text{truth\_value}), b\big) = \big(b, \text{truth\_value} \wedge (a.k < b.k)\big)$

6.2.2 Cypher support for lists

We now briefly outline the operators provided in Cypher for list manipulation.

Creating lists There are several ways to generate a list. We already saw two, namely `nodes(p)` and `relationships(p)` for creating lists of nodes and edges of a path `p`. Entries of such lists are node and edge ids, and thus their labels and properties can be retrieved too. The list of property names of an element `x` (node or edge) can be obtained with `keys(x)`, the list of labels of a node `n` with `labels(n)`.

There are ways to create lists independently of graph elements: the function `range(i, j [, step])` returns the list containing all elements between `i` and `j`, where the difference between two consecutive elements is given by the value of the expression `step`. Also an arbitrary set `s` can be turned into a list by `collect(s)`.

6.2.3 Operations on lists

Alongside list creation functions, Cypher offers many ways of manipulating lists. Among the basic list operations are:

- `L1 + L2` is the concatenation of `L1` and `L2`;
- `e IN L` checks if element `e` belongs to the list `L`;
- `L[n]` returns the element at position `n` in `L`.

The workhorse of list processing is the *reduce()* function (sometimes called `fold` in the context of functional programming). In its most general form it is given as

```
reduce(acc = init, x::L | f(x, acc))
```

where `L` is a list, `init` is the initial configuration for the accumulator variable `acc` and `f(x,acc)` is a function that is applied to a list element `x` and the accumulator value `acc` to produce a new accumulator value. That is

```
reduce(acc=init, [] | f) = init
reduce(acc=init, x::L | f) = f(x, reduce(acc=init, L | f))
```

There are important special cases of `reduce` that have their own syntactic construct due to their frequent use. These are

- **`all`**`(L, p)` returns true if all elements of `L` satisfy the predicate `p` (in this case `init=true` and `f` is conjunction);
- **`none`**`(L, p)` checks if no element of `L` satisfies `p`; this is `all` applied to the negation of `p`;
- **`any`**`(L, p)` returns true if some element of `L` satisfies `p` (here `init=false` and `f` is disjunction);
- `size(L)` outputs the size of `L`; here `init=0` and `f` increments `acc` by 1.

Finally, a list can be turned into a table by **`UNWIND`** `L`; it creates a row for each element of `L`.

Of course Cypher provides many other functions such as `head`, `tail`, `reverse`, `isEmpty`, and others, but the above will suffice for our examples.

In what follows we shall consider two variants of using lists in Cypher:

1. Using the full power of `reduce`;
2. Without the general `reduce` but with derived functions **`all`**, **`none`**, **`any`**, and `size`. Despite the simplicity of this fragment, we shall see that many queries can be written in it, and it comes loaded with issues.

6.3 Expressing RPQs and beyond

We now show how adding lists to the language lets us express what was previously inexpressible in Cypher, namely RPQs and several of their extensions, as well as checking conditions on paths mentioned in the previous section.

Aggregation

The most obvious use of lists is not a limitation of pattern matching per se, but a very common and natural extension: aggregation. Just as in the relational context, aggregation operations can be separated into two categories, vertical and horizontal.

In vertical aggregation, the resulting value is computed over the different matches of the pattern (i.e. a column of the driving table). For example, the query below computes the average length of all road edges in the graph by calculating the average value of the `length` attribute of all the edges that match `r`.

```
MATCH ()-[r :Road]->()
RETURN avg(r.length) AS avg_length
```

In horizontal aggregation, the resulting value is computed over the different values of each match of the pattern (i.e. the rows of the driving table), such as a property along the path. For example, the query below computes the cost of the different routes from Springfield to Shelbyville, taking into account the cost of gas and the tolls. The total length of the route is calculated as the sum of the elements of the `lengths` list, obtained from `(r in relationships(p) | r.length)` which contains the `length` values of all the traversed edges. The cost of gas is then the total length multiplied by the cost of gas per km. The total sum of the tolls is obtained in the same way as the total length except for the attribute `toll`.

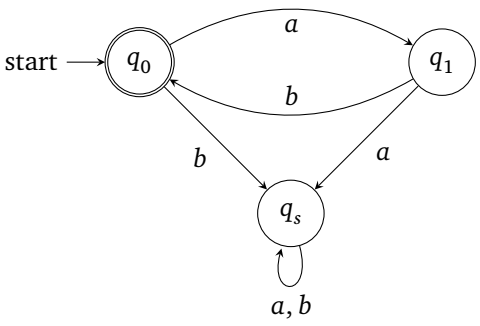
```
MATCH p=({name: Springfield}}-[*]->({name: Shelbyville})
WITH (r IN relationships(p) | r.length) AS lengths,
     (r IN relationships(p) | r.toll) AS tolls
RETURN reduce(sum=0, l IN lengths | sum+l)*1.8 +
       reduce(sum=0, t IN tolls | sum+t) AS cost
```

RPQs and CRPQs

Recall that an RPQ is given by a regular expression e over edge labels, and returns pairs of nodes connected by a path whose label forms a word in the language denoted by e.

Using the well known equivalence between regular languages and DFAs (as explained in Chapter 2), it is clearly enough to write a query that will simulate the run of the underlying automaton of any RPQ over the word composed of the edge-labels of the path.

We illustrate this by an example of a query inexpressible in earlier versions of Cypher, given by the regular expressions $(ab)^*$; the construction will make it clear that it can be extended to an arbitrary DFA.



The automaton capturing $(ab)^*$ (represented above) is given by $\mathscr{A}_Q = (\{q_0, q_1, q_s\}, \{a, b\}, \delta, \{q_0\}, \{q_0\})$ where δ contains the following transitions: (q_0, a, q_1), (q_1, b, q_0), (q_0, b, q_s), (q_1, a, q_s), (q_s, a, q_s), (q_s, b, q_s). Notice that the second (resp. third) transition moves from q_0 (resp. q_1) to the sink state q_s. This is necessary, as our translation requires that the automaton be complete, i.e. there must exist a transition for every input symbol from every state. As any automaton can be completed, by adding missing transitions to a sink state, this is not a restriction.

To emulate $\mathscr{A}$ as a Cypher query, we start by matching an arbitrary path p and then creating its list of edge labels (which we call `types_p`):

```
MATCH p = (x)-[*]-(y)
WITH [r in relationships(p) | type(r)] as types_p, p
```

We then emulate the run of $\mathscr{A}$ over `types_p` using `reduce()`. The current state is stored in the accumulator variable `state` and the transition function is written out explicitly as a list of `CASE` statements (one for each state) each containing a sub-list of `CASE` statements (one for each transition from that state) that returns the next state.

```
WITH reduce (state = 'q0', label in types_p |
CASE state
    WHEN 'q0' THEN
        CASE label
            WHEN 'a' THEN 'q1'
            ELSE 'qs'
        END
    WHEN 'q1' THEN
        CASE label
            WHEN 'b' THEN 'q0'
            ELSE 'qs'
        END
    WHEN 'qs' THEN 'qs'
END
) AS final_state, p
```

Finally, we check that the value `final_state` returned by the `reduce` function, belongs to our set of final states $F = \{q_0\}$ and return `p`.

```
WHERE final_state in ['q0']
RETURN p
```

This approach can clearly be generalized to any finite automaton, by writing out explicitly the whole transition function δ as a series of `CASE` statements in the `reduce` function.

The same approach works for CRPQs (introduced in Chapter 2). Again, we convert each e_i into a DFA $\mathscr{A}_i$ and follow the approach above, instead matching k paths:

```
MATCH p0 = (x0)-[*]-(y0), ..., (pk) = (xk)-[*]->(yk)
WITH [r in relationships(p0) | type(r)] AS types_p0, p0
....
WITH [r in relationships(pk) | type(r)] AS types_pk, pk,
```

followed by k `reduce` statements simulating the k automata, and then checking that all of them are in their respective final states, just as we did above for a single RPQ.

Notice that the implicit join on variables of the same name is preserved

in the translation. Finally this query concludes with a `RETURN` statement projecting out variables corresponding to $\bar{z}$.

6.3.1 Extended CRPQs

[11]: Barceló et al. (2012), "Expressive languages for path queries over graph-structured data"

Using the same ideas, this translation can be adapted to Extended CRPQs (ECRPQs, as defined in [11] and Chapter 2), which, as the name implies, extend CRPQs in two ways: they add the ability to talk about whole paths instead of just their endpoints, and they can express conditions on multiple paths in relation to one another.

The translation to Cypher of CRPQs can be adapted to ECRPQs by creating an additional structure that contains the list of edge-labels of all paths at any given point. This can be achieved by first matching m paths mentioned in the query, extracting their labels into m lists as before and then building a fresh list `path_labels`, of length equal to the longest path, such that `path_label`(i) contains the labels of all edges at position i in the matched paths (or a special symbol if such an edge does not exist). The transition function translation then follows the same structure as for CRPQs, that is, we write a `CASE` statement for each combination of automaton state and permutation of letters from the edge label alphabet.

As an example, assume we have an array of path labels as defined above. Then the condition checking that a path is the prefix of another can be encoded as follows.

```
reduce(state="q0", labels IN path_labels |
CASE state
    WHEN "q0" THEN
        CASE WHEN labels[0]=labels[1] OR labels[0]="-"
            THEN "q0"
            ELSE "qs"
        END
    WHEN "qs" THEN "qs"
END
) AS final_state
WHERE final_state = "q0"
```

The automaton this `reduce()` simulates has two states: q_0, the initial and only final state, and q_s, the sink state. As long as the label of the second path, stored in `labels[1]`, is equal to that of the first, stored in `labels[0]`, or the first path has ended, the automaton stays in q_0. If at any point this is not true, the automaton switches to q_s and remains stuck there until the end. If the state reached at the end of the computation is q_0 then the path whose labels are stored first in `path_labels` is a prefix of the one whose labels are stored second.

Queries comparing values in nodes and edges To illustrate how value-based queries inexpressible in pattern matching can be expressed using lists, we given as an example the query "values in all edges from Start to End are different".

```
MATCH p=(:Start)-[*]->(:End)
WITH [r IN relationships(p) | r.val] AS values, p
WITH reduce(res=[true,[]], val IN values |
        CASE res[0]
```

```
            WHEN true THEN
                CASE WHEN val IN res[1] THEN [false,[]]
                     ELSE [true, res[1]+val]
                END
            ELSE res
        END
    ) AS result,p
WHERE result[0]=true
RETURN p
```

As before, the query iterates over the edges of the potential path, this time storing the values (obtained via the list comprehension expression `[r in relationships(p) | r.val]`) instead of the label. The `reduce` function checks the condition by storing all values encountered thus far in the second element of the accumulator. If the next value is already present in the list, the first element of the accumulator is set to `false` and will stay `false` until the end of the computation, otherwise the second element remains `true` and the new value is added to the list. Notice that the accumulator of this `reduce` function is a complex object: it is a list of size 2, whose first element is a boolean and whose second element is a list of values of arbitrary (albeit all the same) type.

6.4 The Pitfalls of Lists

We have shown in section 6.3, that the `reduce()` function is a powerful tool as it gives a way to express conditions that go beyond regular expressions on any structure. In this section, we show how this expressive power can lead to intractable queries.

To start with, the high expressive power of `reduce` in the context of a query language is not surprising as such. Even for bags, that drop the order from lists, adding `reduce` and nesting allows queries whose complexity is a fixed-height tower of exponentials (e.g., $k-$EXPTIME complexity for any fixed k) and the class of encoded numerical functions is Kalmar-elementary [138, 139]. The latter optimistic name dates back to the early days of recursion theory where it meant "less than primitive recursive"; in reality it captures definitions given by second-, third, …, k-order logic, and is thus completely impractical.

[138]: Grumbach et al. (1996), "Towards Tractable Algebras for Bags"
[139]: Libkin et al. (1997), "Query Languages for Bags and Aggregate Functions"

To see where this extremely high complexity comes from, and crucially how to exclude an easy way of writing effectively non-computable queries, we note that Cypher imposes no restrictions on either the accumulator value, or the combining function. Hence a query computing the powerset of a set, such as the one below, which uses a list of lists as the accumulator and a second `reduce` as the combining function, is allowed.

```
WITH reduce(res=[[]], i in range(n,m) |
    reduce(subres=[], j in res |
        subres+[j+[i]]+[j]
    )
) AS powerset
RETURN powerset
```

The input in this example is a range of numbers (but could be any list). It uses two nested `reduce` functions. The outer loop iterates over a given set S and returns a set of subsets containing elements of S. The

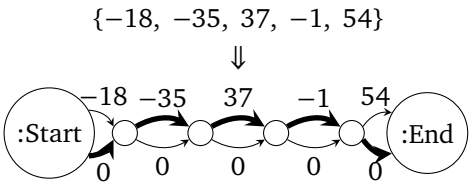


Figure 6.2: Subset sum as a graph problem, in bold a solution for $T = 1$

inner loop iterates over the current subsets and executes two operations: (1) it adds the current element to each subset, and (2) it creates a new subset that contains only the current element.

As the main property of powersets is their exponential size, the above query generates exponentially many results (in the size of the list) and is therefore unreasonably slow for any list containing more than a dozen elements.

To try to curb this behaviour, we consider two restrictions on `reduce()`.

Restriction 1: no composite type accumulators Since the power set query relies on accumulating lists in lists, the first restriction that we consider is to disallow composite type accumulators (such as lists and maps). However, even under such a restriction it is easy to write intractable queries, in fact even avoiding the default trail semantics of Cypher and using a common shortest path semantics.

The problem we consider is *subset sum*: given a (multi)set S and a target-sum T, is there a subset S' of S such that $\sum_{s \in S'} = T$, in other words such that the sum of all elements of S' is equal to the target-sum T? This problem is known to be NP-hard [134] [2].

[134]: Garey et al. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*

2: assuming a binary encoding for the integers. If a unary encoding is used instead, the problem can be solved in logarithmic space [140]

The query computing the SUBSET SUM problem shows that this restriction is not sufficient.

Given a set S, we model it as a graph for the SUBSET SUM problem in the following way:

Assume some enumeration $s_1, \ldots, s_k$ of elements of S. The graph has nodes $n_0, n_1, \ldots, n_k$, with n_0 having label *Start* and n_k having label *End*. All edges have the same label *Edge* and one property *value*. We have two edges from n_{i-1} to n_i, for $0 < i \leq k$, one with $value = 0$ and the other with $value = s_i$. See figure 6.2 for an illustration.

It is clear that all shortest paths from n_0 to n_k have the same length, there are 2^k of them, and along each path one chooses an edge from n_{i-1} to n_i that either has value 0, thereby skipping s_i from the sum, or value s_i, thereby adding it. Using flat lists that only have edges from the path following by *summing up* their elements we encode the SUBSET SUM problem by the following query.

```
MATCH p = allShortestPaths((:Start)-[:Edge*]->(:End))
WITH [r IN relationships(p) | r.value] AS values, p
WHERE reduce(sum = 0, v IN values | sum+v) = $T
RETURN p
```

For the above shaped graphs, this query solves the SUBSET SUM problem by finding a path along this graph such that the sum of the edge values is equal to $\$T$.

Therefore, restricting `reduce()` accumulators to primitive types only is not sufficient. We thus look at much more drastic restriction of `reduce` but even with that, we can still encode computationally intractable problems.

Restriction 2: the only permitted instances of `reduce` are `all` and `size` In other words, only the four simplest incarnations of `reduce` from Section 6.4 are allowed. Notice that `none` and `any` are expressible with `all` using negation. This may seem to be a draconian restriction, eliminating much of the power of `reduce`, and yet the resulting language retains enough power to express intractable problems. Indeed, we now show how to express the HAMILTONIAN PATH PROBLEM, using boolean-only `reduce`.

The *Hamiltonian path* problem is defined as follows: Given a graph G, is there a path p in G such that p visits each node of G exactly once? This problem is also known to be NP-complete [141]. This problem is solved by the following query.

[141]: Karp (1972), "Reducibility Among Combinatorial Problems"

```
MATCH (n)
WITH collect(n.name) AS allNodes
MATCH path=(:Start)-[*]-()
WITH path, allNodes,
    [y IN nodes(path) | y.name] AS nodesInPath
WHERE all(node IN allNodes WHERE node IN nodesInPath)
AND size(allNodes)=size(nodesInPath)
RETURN path LIMIT 1
```

The first two lines of the query collect all node ids into the list `allNodes`. Then the query matches any (trail) path and collects its node ids into the list `nodesInPath`. To check that each node is traversed at least once, we use the `all()` function, a specialization of `reduce()` which checks that the argument predicate holds for all elements of the argument list (notice that this is a boolean check). We apply `all()` on `allNodes` to check that each of its elements appears in `nodesInPath`. To check that each node is traversed at most once we compare the size of `allNodes` with `nodesInPath`, and require them to be equal.

Clearly, restricting `reduce()` to only its simplest special cases is not a good solution either.

Thus, even with severe restrictions on `reduce`, one can encode intractable problems in Cypher. There is a worrying element here from the language design point of view: it is *how easy* it was to write these queries. It is true that full SQL is Turing complete and one can write a Turing machine simulator in it, but this is hardly a natural query. The queries we have shown in this section are simple and natural and one can thus assume that a moderately advanced programmer will be able to handle them.

The question this leads to is the following: will this theoretical complexity show up in practice? After all, there are many NP-hard problems that are routinely solved even in the database context, not least the problem of finding trail paths. This is an NP-hard problem and it has not stopped Cypher from being the (so far) dominant graph query language, despite trails being its default path semantics. There are multiple other examples of this kind, like exponential-time-hard typechecking problems in some widely used programming languages [142] that nonetheless only

[142]: Mairson (1989), "Deciding ML typability is complete for deterministic exponential time"

manifests itself in artificial examples and does not affect everyday programming practice.

Perhaps then theoretically intractable queries will be handled by the query engine with ease? We next do an experimental study to show that to the contrary, such queries are completely impossible, even on tiny graphs.

6.5 Experimental results

As explained at the end of the last section, NP-hardness is usually bad news but there are exceptions when the cases showing hardness do not realistically occur. Thus, to see whether the results of the previous section indeed spell bad news for list processing in graph queries, we evaluate such queries experimentally, with the result fully expected by the reader who has glanced at the title of the section.

To tests the actual performance of hard queries, we ran a set of tests using Neo4j, the most widely used graph database engine. We started our experiments with randomized data, and discovered that the performance was so poor (only very small graphs with fewer than 30 nodes could be handled before timeout) that there was simply no real data of such tiny size we could realistically extend the experiments to. The testing program is written in Go and communicates with Neo4j (v5.18.1) via the Neo4j Go driver. All tests were executed on a machine with the following configuration: 16 Intel i7-10700 @ 2.90GHz CPUs, 16GB RAM, Ubuntu 22.04.3 LTS.

The tests proceed as follows :

1. For each p between 0.1 and 1.0 [3] and each n between `minNodes` and `maxNodes`, do the following 5 times:
2. Generate a random graph with n nodes, with the property `name` ranging from 1 to n, in which any two nodes are connected with probability p by an edge labelled `Edge`.
3. Do the following 5 times and log the execution time of the last 4 iterations [4]:
4. Generate the specified query with random start and end points
5. Ask the DBMS to run the query on the generated graph. If the query has not returned after 5 minutes, declare it timed-out.

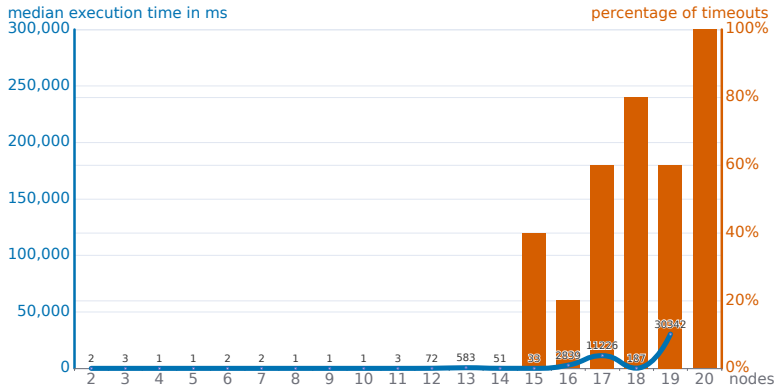
3: To avoid memory pollution between rounds, the Neo4j server was manually restarted between each increase of p.

4: The first iteration gives Neo4j a chance to generate the appropriate indices.

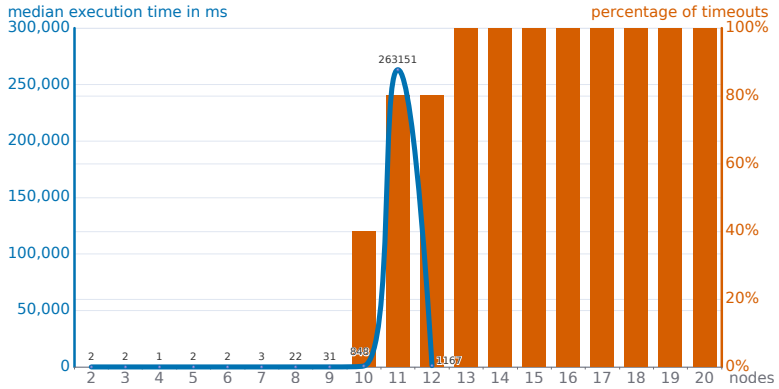
To start, Figure 6.3 presents a sample of runtimes of the HAMILTONIAN PATH query written in Cypher. We present it here for small edge probabilities $p = 0.1,\ 0.2,\ 0.3$ and one high probability $p = 0.8$. As p increases, the graph becomes closer to a complete graph with the number of paths growing as the factorial function, resulting in a quickly degrading performance.

The figure shows both the median execution time (blue line, scale on the left) and percentage of timeouts (bars; scale on the right). If some runs time out, we take the median only over those that do not (explaining nonmonotone behavior in some cases where we see both timeouts and successful executions). The timeout was set at $3 \cdot 10^5$ms.

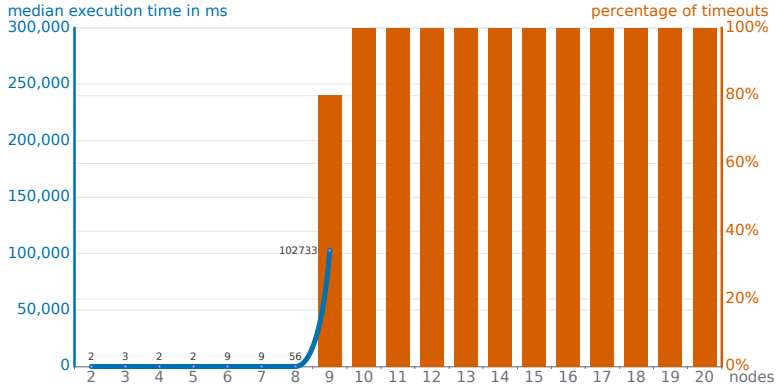
According to these tests, for 65.26% of the configurations all iterations time out. When nodes have a probability of being connected of more than 0.7, the engine cannot handle more than 4 nodes! For sparse graphs the situation is not much better with the engine failing on 10



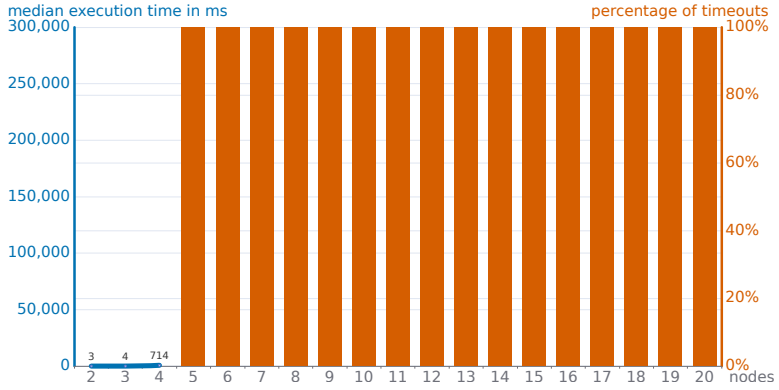
(a) Median execution time and number of timeouts for $p = 0.1$



(b) Median execution time and number of timeouts for $p = 0.2$



(c) Median execution time and number of timeouts for $p = 0.3$



(d) Median execution time and number of timeouts for $p = 0.8$

Figure 6.3: Results of the performance tests on Neo4j for the Hamiltonian path problem

nodes for $p = 0.3$ and 8 nodes for $p = 0.4$. The biggest graph for which Neo4j can find a solution for the Hamiltonian path problem contains just 19 nodes. For the configurations for which Neo4j does manage to find a solution, the running time is surprisingly slow: for a graph of 6 nodes and probability 0.6, the median time before a solution is found is a bit more than 3min.

Of course there could be another potential culprit, namely trail semantics, which is the default semantics of Cypher: only paths with no repeated edges are returned. In fact this makes Cypher pattern matching NP-complete in general [137] though it is not the case for all queries. Indeed [84] identified the class T_{tract} of tractable queries for the Regular Trail Query problem. Any query that belongs to T_{tract} is either NLOGSPACE-complete or in AC^0, and any query that does not belong to T_{tract} is NP-complete. In fact relatively few real-life regular patterns fall outside the class T_{tract} which explains the good behavior of the trail semantics in practice [143]. A rare pattern that does occur in practice and that has theoretical NP-complete bound is of the form A^*BA^*, i.e., a path of edges labeled by A with the exception of a single edge labeled B that occurs anywhere on the path. The existence of such a path is of course easily checked in Cypher:

[137]: Francis et al. (2018), "Cypher: An Evolving Query Language for Property Graphs"

[84]: Martens et al. (2020), "A Trichotomy for Regular Trail Queries"

[143]: Bonifati et al. (2017), "An Analytical Study of Large SPARQL Query Logs"

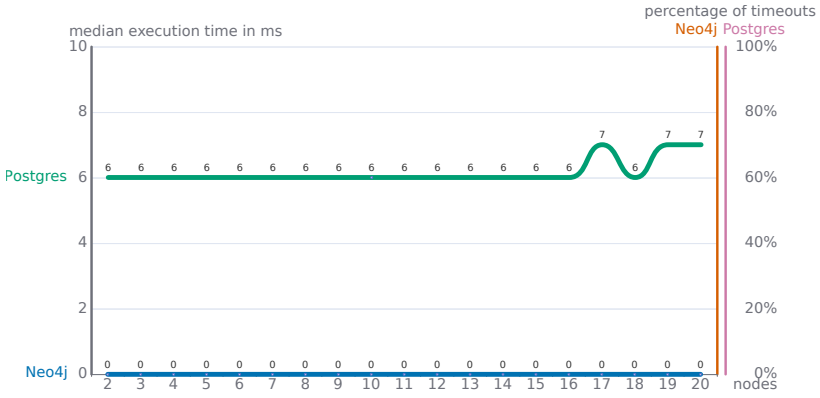
```
MATCH p = ()-[:A*]->()-[:B]->()-[:A*]->()
RETURN p LIMIT 1
```

Thus, as the first test to see whether the culprit of the bad behavior of the Hamiltonian path query is lists of trails, we test the performance of the above query. These performance results are visualized in figure 6.4, with the blue line. The highest median execution time is only 2ms (as opposed to $3 \cdot 10^5$ms for timeouts for Hamiltonian path). Even with three observed timeouts for the high probability $p = 0.8$ (among a total of 100 tests), the enormous gap between the results for two theoretically NP-complete queries, one using lists and trails and the other using trails alone, clearly point to lists as the reason.

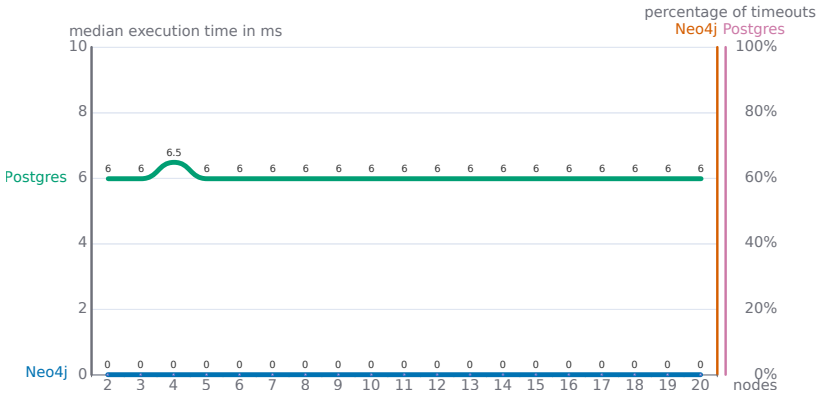
To further confirm that lists rather than trails are the real cause of extremely poor behavior we test the subset sum query from Section 6.4. Recall that this query also encodes an NP-complete problem, but does so with shortest paths rather than trails, and of course finding shortest paths is tractable.

The results of the performance tests of the subset sum query are shown in figure 6.5. Recall that in our encoding of this problem as a line graph with parallel edges, the graph is fixed (there is no random generation), so we only report the number of nodes on the x axis. On the y axis, as before, we show the median running time with timeout set at $3 \cdot 10^5$ms, and on the right the bars indicate the percentage of timeouts. This time we do 20 iterations for each length. Although the performance is very good for very small graphs, staying under 2000ms for up to 20 nodes, the exponential nature of the problem becomes very quickly apparent, eventually reaching the 100% five-minute time out on 27 nodes.

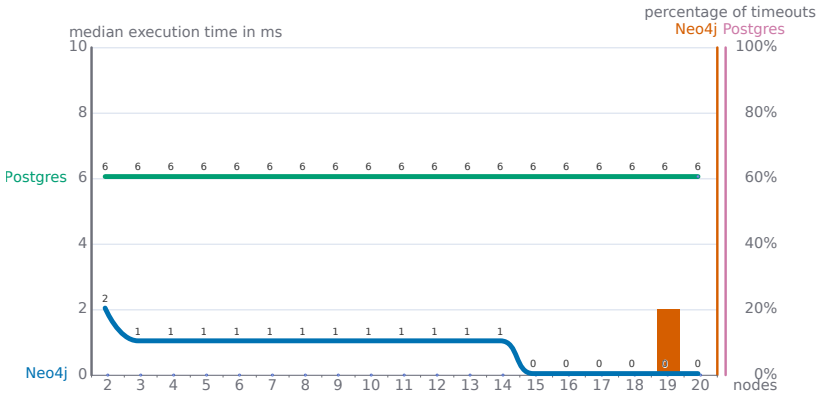
Since for this query the only source of complexity is the use of lists to encode subset sum, together with other results of this section it clearly points to complete inability of the state-of-the-art graph database engine to handle anything other than the tiniest of inputs.



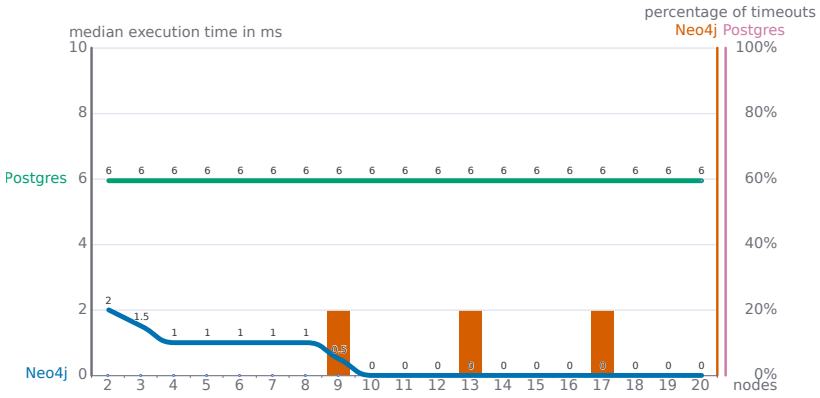
(a) Median execution time and number of timeouts for $p = 0.1$



(b) Median execution time and number of timeouts for $p = 0.2$

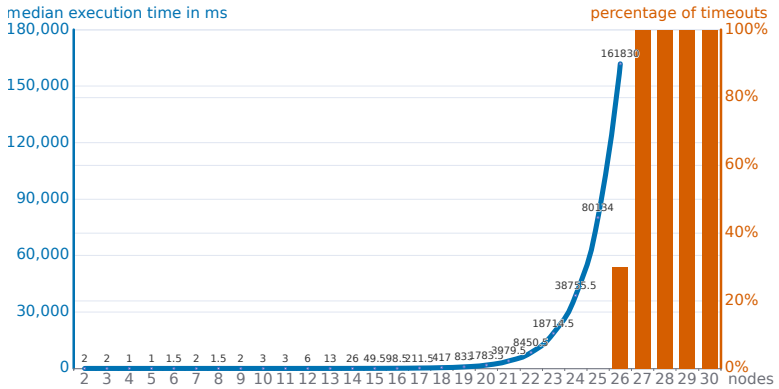


(c) Median execution time and number of timeouts for $p = 0.3$

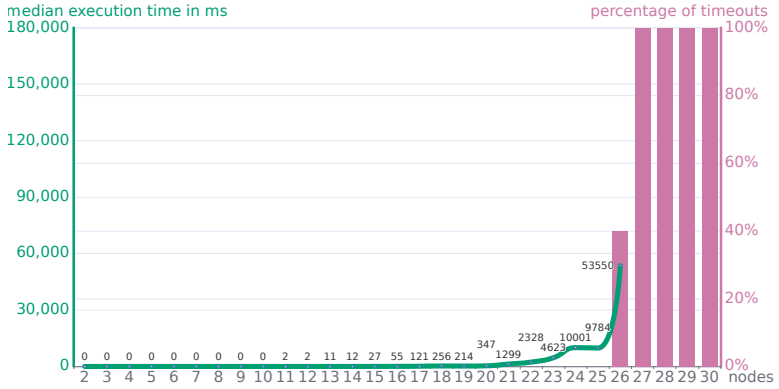


(d) Median execution time and number of timeouts for $p = 0.8$

Figure 6.4: Results of the performance tests on Neo4j and Postgres for A^*BA^*



(a) Neo4j



(b) Postgres

Figure 6.5: Results of the performance tests on Neo4j and Postgres for the SUBSET SUM problem

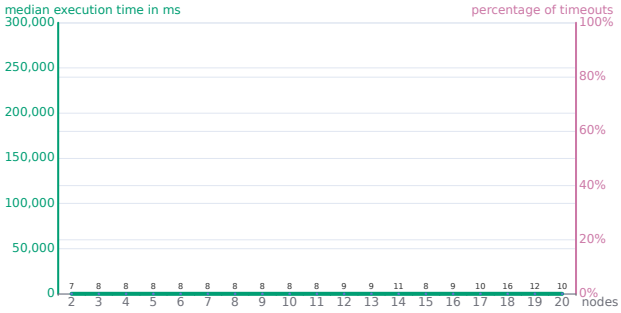
6.6 Can SQL help?

A case for the use of relational database engines over specialized graph engines has been made in the context of analytic and concurrent transactional workloads [144–146]. On the other hand, the idea that graph databases outperform relational databases for navigational queries is widespread, even though reservations have been expressed in the case of complex queries [147, 148]. Following pointers, as described in [149], can indeed be done in constant time, thereby avoiding costly joins, which suggests an advantage for native graph structures.

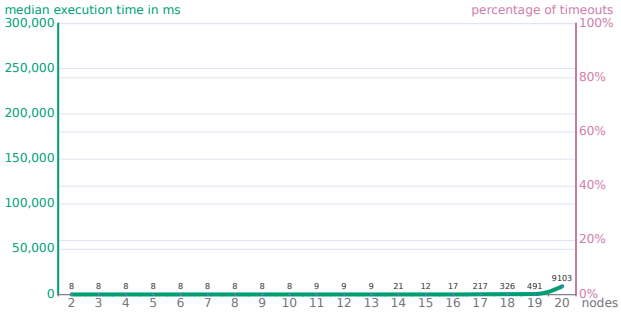
Since no crystal clear picture emerges from existing studies as to the advantages of a relational representation or a native graph engine, we look at the problematic queries from the previous sections and see how an SQL DBMS would handle them. Notice that all queries from the previous sections can be expressed in SQL with recursive common table expressions[5].

There are multiple ways to encode property graphs in relational structures. We settle here for simple encodings which minimize the number of joins and facilitate writing queries. In the case of the A^*BA^* query, which is intended to show how well the trail semantics is enforced, we encode edge labeling as ternary relations with attributes for source, target and label, thus also implicitly encoding edges. We use PostgreSQL and its built-in array type in order to store paths. The recursive part of the query constructs every A-labeled trail of length k as an array of the

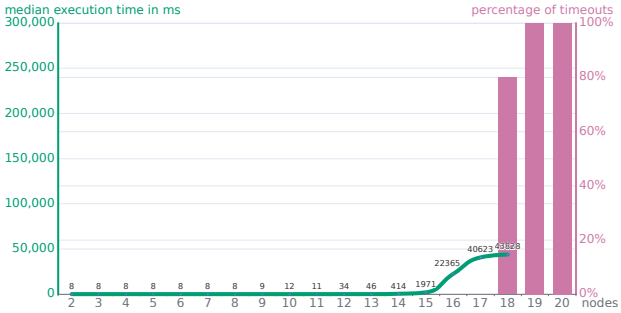
[144]: Najork et al. (2012), "Of hammers and nails: an empirical comparison of three paradigms for processing large graphs"
[145]: Fan et al. (2015), "The Case Against Specialized Graph Analytics Engines"
[146]: Pacaci et al. (2017), "Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications"
[147]: Angles et al. (2013), "Benchmarking database systems for social network applications"
[148]: Kotiranta et al. (2022), "Performance of Graph and Relational Databases in Complex Queries"
[149]: Robinson et al. (2013), *Graph databases*
5: Indeed, this is folklore knowledge that SQL with recursive CTEs is Turing complete, as shown for example here using a Turing machine simulation.



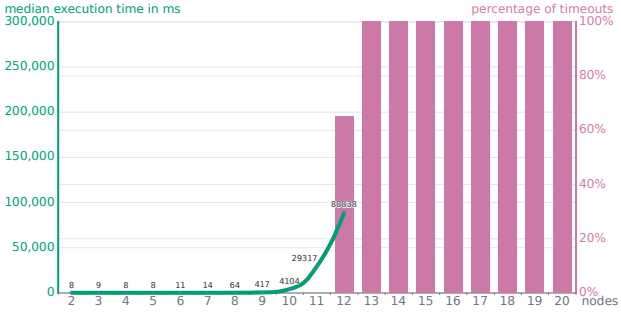
(a) Median execution time and number of timeouts for $p = 0.1$



(b) Median execution time and number of timeouts for $p = 0.2$



(c) Median execution time and number of timeouts for $p = 0.3$



(d) Median execution time and number of timeouts for $p = 0.8$

Figure 6.6: Results of the performance tests on Postgres for the Hamiltonian path problem

form $[s_0.t_0, \ldots, s_k.t_k]$, where identifiers s_i and t_i of the source and the target from each edge in the path are just concatenated using the dot as a separator. The trail condition is then enforced in the `WHERE` clause by filtering out already visited edges. The query is shown below:

```
WITH RECURSIVE a_kleene_star AS (
    SELECT s, t, 0 AS depth, array[s,t] AS path,
            array[s||'.'|| t] AS edges FROM A
    UNION
    SELECT A.s, A.t, a_kleene_star.depth+1,
            a_kleene_star.path||A.t,
            a_kleene_star.edges ||
            concat(A.s||'.',A.t)
    FROM A , a_kleene_star
    WHERE A.s=a_kleene_star.t AND
    NOT concat(A.s||'.',A.t)=ANY(a_kleene_star.edges)
)
SELECT A1.s, A2.t
FROM a_kleene_star A1, a_kleene_star A2, B
WHERE A1.t=B.s AND B.t=A2.s LIMIT 1;
```

PostgreSQL performance on this query, on data generated as for the Cypher query (modulo the representation of data as relations) is shown in Figure 6.4 as the green line. We see that the performance is completely adequate even if marginally worse than in the case of Neo4j (which is expected as this is literally the problem graph databases are designed to solve).

We next move to the Hamiltonian path query. Since it only concerns the underlying graph structure (no reference to properties), we use an even simpler encoding of the graph as a binary relation `G` with two attributes `src` and `tgt` ranging over node identifiers of sources and targets. Candidate Hamiltonian paths are constructed iteratively by initially storing the source and target of each edge in a different array. Whenever another edge can be reached from the edge previously stored in the array, the target of that new edge is added to the array, but only if it was not already in it. At the end of the iteration it only remains to check whether one of those arrays is of the same size as the graph. If so then it contains a Hamiltonian path for the graph. The query is shown below:

```
WITH RECURSIVE paths(startP, endP, path)
AS (SELECT src AS startP,
           tgt AS endP,
           ARRAY[src,tgt] as path
    FROM G
    UNION
    SELECT startP, tgt, array_append(path,tgt)
    FROM G, paths
    WHERE src=endP AND tgt <> ALL(path))
SELECT * FROM paths
WHERE ARRAY_LENGTH(path,1) =
      (SELECT COUNT(DISTINCT src) FROM G)
LIMIT 1;
```

Performance results are shown in Figure 6.6. While for low probabilities and small values of n both Cypher and SQL perform well (with SQL being marginally slower), SQL's coverage is better when it comes to timeouts on a higher number of nodes (e.g., 100% timeout on 19 nodes for $p = 0.3$ as opposed to 10 for Cypher). Regardless, just like Cypher, SQL can only handle tiny graphs here, and for any reasonable size graph

performance would not be adequate independently of the choice of SQL or Cypher to encode the problem.

We next look at the SUBSET SUM query. Here we encode the graph as a ternary relation `G` storing the source, target and weight of each edge. Candidate paths are iteratively constructed as array structures. For each edge in the graph, an array containing its source identifier, weight and target identifier is first initialized. Variables for the first node in the path, the last node and the total weight of the path are also initialized. When another edge can be reached from the edge previously stored in the array, the target of that new edge is added to the array at each iterative stage. As we only run this query on graphs with no cycles (as shown in Figure 6.2), the computation terminates and only gives rise to trails. Finally, the last `SELECT` clause returns among candidate paths those which satisfy the desired condition. The query is shown below.

```
WITH RECURSIVE paths(p_src, p_tgt, path, total_weight)
AS (SELECT src AS p_src, tgt As p_tgt,
           ARRAY[src,weight,tgt] AS path,
           weight AS total_weight
    FROM G
    WHERE src = 0
    UNION
    SELECT p_src, tgt,
        array_append(array_append(path,weight),tgt),
        total_weight+weight AS total_weight
    FROM G, paths
    WHERE src=p_tgt)
SELECT * FROM paths
WHERE total_weight=0 AND p_src=0 AND target=$T$;
```

Performance results for this query are shown in the right graph in Figure 6.5. In this case Cypher and SQL behave very similarly, with the exponential growth starting a bit earlier for Cypher, but then 100% of timeouts is reached on graphs of the same size.

In conclusion, a SQL DBMS does not in general outperform a native graph database on the problematic queries we explored in the previous sections. In fact the results are roughly comparable.

There is one striking difference however. Queries such as SUBSET SUM and HAMILTONIAN PATH that used lists explicitly are *very easy* to write in Cypher. In fact the queries look so deceptively simple that one can expect a reasonably proficient programmer to write them with ease, but then there is no chance they will perform well. In SQL on the other hand these queries are much harder to write, and their general shape may serve as an indication that their performance will not be adequate. Thus again the explicit use of list operations in Cypher queries as its specific design feature can easily lead to significantly degraded performance.

6.7 Conclusion

While performance figures in Section 6.5 may suggest that list processing can ruin everything, it is nonetheless a very convenient device that oftentimes can and will be used without causing significant problems. The main culprit behind the poor performance is not a particular database engine but rather the *design of the language* that makes it possible to write offending queries with ease.

Before outlining a possible remedy, we note that a careless approach to the design of language features and their semantics can lead to even worse circumstances. Consider the following example inspired by Chapter 4. The database is a very simple graph: it has a loop on a node with label `lab1`, and an isolated node with label `lab2` and three properties `a,b,c` with integer values. Next consider the following query.

```
MATCH p=allShortestPaths((:lab1)-[*]->()), (y:lab2)
WITH reduce(s=0,
        v IN [r IN relationships(p) | r.v] | s+v) AS x,
     y.a AS a, y.b AS b, y.c AS c
WHERE a*x*x+b*x+c=0
RETURN p LIMIT 1
```

There are two ways of providing a semantics to this query, depending on the point at which the filter in `WHERE` is applied:

- Assume it is a post-filter, i.e., it is applied after *shortest*. Then a single shortest path of length 1 is computed, and the condition simply checks if $a + b + c = 0$.
- Assume however that it is a *pre-filter*, i.e., *shortest* applies to paths that satisfy the condition. Then the condition checks whether the quadratic equation $ax^2+bx+c = 0$ has a positive integer solution.

The latter means that using `reduce` in pre-filters one can check conditions that are at best done by specialized solvers but in general might even be undecidable. Indeed, if instead of checking the existence of an integer solution to a univariate quadratic polynomial we asked for an integer solution to a multivariate polynomial of degree 4 with a fixed number of variables, this would be an undecidable problem [150], yet encodable with pre-filters.

[150]: Gasarch (2021), "Hilbert's Tenth Problem for Fixed d and n"

If we look at all our examples that led to high complexity of queries, they used post-filers with conditions involving outputs of `reduce`. The above example shows that a simple looking language design decision – using pre-filters instead of post-filters – can make the problem much worse and even lead to undecidability of query evaluation. The question therefore is: what are the lessons for query language design for graph databases?

The **first lesson** is that conditions based on `reduce` should be *disallowed* in `WHERE`. In fact it is easy to trace subexpressions used in `WHERE`, and if any of them used `reduce` in its syntax tree, such an expression should result in a compilation error.

As indicated in Section 6.3, there are multiple examples showing the usefulness of `reduce`. In particular, doing a computation on lists and then returning results rather than using them for filtering is both useful and harmless complexity-wise. Thus, the **second lesson** is that using `reduce` is fine in `RETURN`, `WITH` and similar statements in other languages (as long as there is no violation of lesson one).

This leaves us with an interesting case of using `reduce` to simulate the power of various automata. Our **lesson three** is that for such problems, query languages should provide facilities that *are not based on* `reduce`. This has already been done for RPQs, with GQL and SQL/PGQ providing facilities for expressing them. For more complex path queries, such as RPQs with data or extended CRPQs, so far GQL provides ad hoc facilities. However it is already recognized that the language falls short of some desired expressiveness, and work is under way to enhance GQL's

capabilities ahead of the next release. Our results inform this effort, by explaining what can or cannot (and more importantly should and should not) be done with list facilities that are currently in existence.

References

[11] Pablo Barceló et al. “Expressive languages for path queries over graph-structured data”. In: *ACM Trans. Database Syst.* 37.4 (2012), 31:1–31:46.

[15] Nadime Francis et al. “Cypher: An Evolving Query Language for Property Graphs”. In: *Proceedings of the 2018 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

[19] Marcelo Arenas et al. *Database Theory*. Open source at https://github.com/pdm-book/community, 2022.

[60] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. “Querying Graphs with Data”. In: *Journal of the ACM* 63.2 (2016), 14:1–14:53.

[84] Wim Martens, Matthias Niewerth, and Tina Trautner. “A Trichotomy for Regular Trail Queries”. In: *International Symposium on Theoretical Aspects of Computer Science, (STACS)*. 2020, 7:1–7:16.

[134] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1979.

[135] Leonid Libkin and Limsoon Wong. “On the Power of Aggregation in Relational Query Languages”. In: *Database Programming Languages, 6th International Workshop, DBPL-6, Estes Park, Colorado, USA, August 18-20, 1997, Proceedings*. Ed. by Sophie Cluet and Richard Hull. Vol. 1369. Lecture Notes in Computer Science. Springer, 1997, pp. 260–280. DOI: 10.1007/3-540-64823-2_15.

[136] Graham Hutton. “A tutorial on the universality and expressiveness of fold”. In: *Journal of Functional Programming* 9.4 (1999), pp. 355–372.

[137] Nadime Francis et al. “Cypher: An Evolving Query Language for Property Graphs”. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

[138] Stéphane Grumbach and Tova Milo. “Towards Tractable Algebras for Bags”. In: *J. Comput. Syst. Sci.* 52.3 (1996), pp. 570–588. DOI: 10.1006/JCSS.1996.0042.

[139] Leonid Libkin and Limsoon Wong. “Query Languages for Bags and Aggregate Functions”. In: *J. Comput. Syst. Sci.* 55.2 (1997), pp. 241–272. DOI: 10.1006/JCSS.1997.1523.

[140] Daniel M. Kane. *Unary Subset-Sum is in Logspace*. 2017.

[141] Richard M. Karp. “Reducibility Among Combinatorial Problems”. In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[142] Harry G. Mairson. “Deciding ML typability is complete for deterministic exponential time”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’90. San Francisco, California, USA: Association for Computing Machinery, 1989, pp. 382–401. DOI: 10.1145/96709.96748.

[143] Angela Bonifati, Wim Martens, and Thomas Timm. “An Analytical Study of Large SPARQL Query Logs”. In: *Proc. VLDB Endow.* 11.2 (2017), pp. 149–161. DOI: 10.14778/3149193.3149196.

[144] Marc Najork et al. “Of hammers and nails: an empirical comparison of three paradigms for processing large graphs”. In: *Proceedings of the Fifth International Conference on Web Search and Web Data Mining, WSDM 2012, Seattle, WA, USA, February 8-12, 2012*. Ed. by Eytan Adar et al. ACM, 2012, pp. 103–112.

[145] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. “The Case Against Specialized Graph Analytics Engines”. In: *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[146] Anil Pacaci et al. “Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications”. In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*. Ed. by Peter A. Boncz and Josep Lluís Larriba-Pey. ACM, 2017, 12:1–12:7.

[147] Renzo Angles et al. “Benchmarking database systems for social network applications”. In: *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*. Ed. by Peter A. Boncz and Thomas Neumann. CWI/ACM, 2013, p. 15.

[148] Petri Kotiranta, Marko Junkkari, and Jyrki Nummenmaa. “Performance of Graph and Relational Databases in Complex Queries”. In: *Applied Sciences* 12.13 (2022).
[149] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. O’Reilly Media, 2013.
[150] William Gasarch. “Hilbert’s Tenth Problem for Fixed d and n”. In: *Bull. EATCS* 133 (2021).

7 Conclusion

The main theoretical goal of this thesis was to create a theoretical model for the new standard graph query languages GQL and SQL/PGQ. We started, in Chapter 3, by identifying the main features of the pattern matching mechanisms shared by both languages. We explained and defined these features, trying to stay as close as possible to the definitions given by the standards. While the resulting model is a useful tool to understand the ideas behind GQL and SQL/PGQ, it is not yet suitable for formal study.

The next step, presented in Chapter 4, was to refine the definition of pattern matching down to its core components and present it in a style closer to usual theoretical query languages. This allowed us to prove two basic properties, namely the complexity of query enumeration (PSPACE in data and EXPSPACE in combined) and that the previously studied languages of UC2RPQs, Nested Regular Expressions, Regular Queries and the like are all expressible within this core of pattern matching.

Now equipped with a better understanding of pattern matching, in Chapter 5 we extended the model to not only encompass the full languages of GQL and SQL/PGQ, but also highlight the difference in their style of computation and bring to light their strong link with relational languages. As a result of this new approach, we were also able to show that the design of GQL and SQL/PGQ is lacking, as some simple queries either cannot be expressed in the languages at all, or only through inelegant and twisted means.

To finish, in Chapter 6 we looked at the impact of adding list operations to pattern matching. As expected, we showed that even the most basic aggregation functions come at an unreasonably high cost to both expressive power and practical performance. This observation is yet another argument to justify the need to enhance the pattern matching of GQL and SQL/PGQ as this would remove this need for list operations, at least for some valuable queries.

Thanks to the tools developed in these chapters, we have been able to pin-point some deficiencies in the design of GQL and SQL/PGQ. The high complexity of query enumeration shows that even just the core of these languages contains intractable queries which can make a system crash or timeout. The inability to express some simple queries and the discrepancies in the treatment of nodes and edges are proof that an important piece is missing, and Neo4j's attempt to fill that gap with list operations only results in more intractability.

In comparison, the data complexity of the core of SQL (i.e. Relational Algebra) is only AC^0 for query evaluation and its expressive power is sufficient for all first-order queries, making the language appropriate and efficient for relational databases. One possible explanation for this disparity between SQL and GQL is their inverted development strategies. When the GQL and SQL/PGQ working groups were created, the existing theoretical models were too far removed from the practical graph languages and so could not have an influence on the decisions made by the committees.

The models presented in this thesis provide a basis for developing new theoretical languages with enhanced expressive power and reasonable complexity, that will hopefully lead to better graph query languages.

Index

Bibliography

[1] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685.

[2] D. C. Tsichritzis and F. H. Lochovsky. "Hierarchical Data-Base Management: A Survey". In: *ACM Comput. Surv.* 8.1 (1976), pp. 105–123. DOI: 10.1145/356662.356667.

[3] Robert W. Taylor and Randall L. Frank. "CODASYL Data-Base Management Systems". In: *ACM Comput. Surv.* 8.1 (1976), pp. 67–103. DOI: 10.1145/356662.356666.

[4] Renzo Angles and Claudio Gutierrez. "Survey of graph database models". In: *ACM Comput. Surv.* 40.1 (2008), 1:1–1:39. DOI: 10.1145/1322432.1322433.

[5] https://db-engines.com. https://db-engines.com/en/ranking/graph+dbms.

[6] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. "A Graphical Query Language Supporting Recursion". In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. Ed. by Umeshwar Dayal and Irving L. Traiger. ACM Press, 1987, pp. 323–330. DOI: 10.1145/38713.38749.

[7] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. "G+: Recursive Queries Without Recursion". In: *Expert Database Systems, Proceedings from the Second International Conference, Vienna, Virginia, USA, April 25-27, 1988*. Ed. by Larry Kerschberg. Benjamin/Cummings, 1988, pp. 645–666.

[8] Mariano P. Consens and Alberto O. Mendelzon. "GraphLog: a Visual Formalism for Real Life Recursion". In: *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. ACM Press, 1990, pp. 404–416. DOI: 10.1145/298514.298591.

[9] Pablo Barceló. "Querying graph databases". In: *Principles of Database Systems (PODS)*. 2013, pp. 175–188.

[10] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. "Querying Graphs with Data". In: *J. ACM* 63.2 (2016), 14:1–14:53. DOI: 10.1145/2850413.

[11] Pablo Barceló et al. "Expressive languages for path queries over graph-structured data". In: *ACM Trans. Database Syst.* 37.4 (2012), 31:1–31:46.

[12] Peter T. Wood. "Query languages for graph databases". In: *SIGMOD Record* 41.1 (2012), pp. 50–60.

[13] Diego Figueira. "Containment of UC2RPQ: The Hard and Easy Cases". In: *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*. Ed. by Carsten Lutz and Jean Christoph Jung. Vol. 155. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 9:1–9:18. DOI: 10.4230/LIPICS.ICDT.2020.9.

[14] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. "Relative Expressiveness of Nested Regular Expressions". In: *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27-30, 2012*. 2012, pp. 180–195.

[15] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

[16] Property Graph Query Language. *PGQL 1.4 Specification*. 2021. URL: https://pgql-lang.org/spec/1.4/.

[17] Alin Deutsch et al. "Aggregation Support for Modern Graph Analytics in TigerGraph". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2020, pp. 377–392. DOI: 10.1145/3318464.3386144.

[18] Renzo Angles et al. "G-CORE: A Core for Future Graph Query Languages". In: *SIGMOD*. 2018, pp. 1421–1432.

[19] Marcelo Arenas et al. *Database Theory*. Open source at https://github.com/pdm-book/community, 2022.

[20] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[21] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[22] Wilfrid Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[23] Michael Sipser. “Introduction to the Theory of Computation”. In: *SIGACT News* 27.1 (1996), pp. 27–29. DOI: 10.1145/230514.571645.

[24] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[25] James Jones. “Undecidable diophantine equations”. In: *Bulletin of the American Mathematical Society* 3.2 (1980), pp. 859–862.

[26] Ashok K. Chandra and Philip M. Merlin. “Optimal Implementation of Conjunctive Queries in Relational Data Bases”. In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*. Ed. by John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison. ACM, 1977, pp. 77–90. DOI: 10.1145/800105.803397.

[27] Manuel Lima. *The book of trees : visualizing branches of knowledge*. New York : Princeton Architectural Press, 2014.

[28] openCypher. *Cypher Query Language Reference, Version 9*. 2017. URL: https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf.

[29] Alberto O. Mendelzon and Peter T. Wood. “Finding Regular Simple Paths in Graph Databases”. In: *SIAM J. Comput.* 24.6 (1995), pp. 1235–1258.

[30] Béla Bollobás. *Modern Graph Theory*. Vol. 184. Springer Science & Business Media, 2013.

[31] Olaf Hartig et al. *RDF 1.2 Concepts and Abstract Syntax*. Mar. 2024. URL: https://www.w3.org/TR/2024/WD-rdf12-concepts-20240307/.

[32] Guus Schreiber and Yves Raimond. *RDF 1.1 Primer*. June 2014. URL: https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[33] Claudio Gutierrez, Carlos A. Hurtado, and Alberto O. Mendelzon. “Foundations of Semantic Web Databases”. In: *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*. Ed. by Catriel Beeri and Alin Deutsch. ACM, 2004, pp. 95–106. DOI: 10.1145/1055558.1055573.

[34] Gavin Carothers and Eric Prud’hommeaux. *RDF 1.1 Turtle*. Feb. 2014. URL: https://www.w3.org/TR/2014/REC-turtle-20140225/.

[35] Dominik Tomaszuk and David Hyland-Wood. “RDF 1.1: Knowledge Representation and Data Integration Language for the Web”. In: *Symmetry* 12.1 (2020), p. 84. DOI: 10.3390/SYM12010084.

[36] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. “Reifying RDF: What Works Well With Wikidata?” In: *SSWS@ISWC*. 2015.

[37] Olaf Hartig. “RDF* and SPARQL*: An Alternative Approach to Annotate Statements in RDF”. In: *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 23rd - to - 25th, 2017*. Ed. by Nadeschda Nikitina et al. Vol. 1963. CEUR Workshop Proceedings. CEUR-WS.org, 2017.

[38] Renzo Angles et al. *PG-Schema: Schemas for Property Graphs*. 2022.

[39] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. “RDF and Property Graphs Interoperability: Status and Issues”. In: *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019*. Ed. by Aidan Hogan and Tova Milo. Vol. 2369. CEUR Workshop Proceedings. CEUR-WS.org, 2019.

[40] Serge Abiteboul and Victor Vianu. “Regular Path Queries with Constraints”. In: *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA*. Ed. by Alberto O. Mendelzon and Z. Meral Özsoyoglu. ACM Press, 1997, pp. 122–133. DOI: 10.1145/263661.263676.

[41] Erkki Mäkinen. “On Lexicographic Enumeration of Regular and Context-Free Languages”. In: *Acta Cybern.* 13.1 (1997), pp. 55–61.

[42] Margareta Ackerman and Jeffrey O. Shallit. “Efficient enumeration of words in regular languages”. In: *Theor. Comput. Sci.* 410.37 (2009), pp. 3461–3470. DOI: 10.1016/J.TCS.2009.03.018.

[43] Wim Martens and Tina Trautner. “Evaluation and Enumeration Problems for Regular Path Queries”. In: *International Conference on Database Theory (ICDT)*. Vol. 98. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 19:1–19:21.

[44] Katrin Casel and Markus L. Schmid. “Fine-Grained Complexity of Regular Path Queries”. In: *Log. Methods Comput. Sci.* 19.4 (2023). DOI: 10.46298/LMCS-19(4:15)2023.

[45] Dario Colazzo and Carlo Sartiani. “Typing regular path query languages for data graphs”. In: *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by James Cheney and Thomas Neumann. ACM, 2015, pp. 69–78. DOI: 10.1145/2815072.2815082.

[46] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. "Regular Path Query Evaluation on Streaming Graphs". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier et al. ACM, 2020, pp. 1415–1430. DOI: 10.1145/3318464.3389733.

[47] Diego Calvanese et al. "Reasoning on regular path queries". In: *SIGMOD Rec.* 32.4 (2003), pp. 83–92. DOI: 10.1145/959060.959076.

[48] Diego Calvanese et al. "Answering Regular Path Queries Using Views". In: *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. Ed. by David B. Lomet and Gerhard Weikum. IEEE Computer Society, 2000, pp. 389–398. DOI: 10.1109/ICDE.2000.839439.

[49] Miguel Romero, Pablo Barceló, and Moshe Y. Vardi. "The homomorphism problem for regular graph patterns". In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017, pp. 1–12. DOI: 10.1109/LICS.2017.8005106.

[50] Diego Figueira and Rémi Morvan. "Approximation and Semantic Tree-Width of Conjunctive Regular Path Queries". In: *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*. Ed. by Floris Geerts and Brecht Vandevoort. Vol. 255. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 15:1–15:19. DOI: 10.4230/LIPICS.ICDT.2023.15.

[51] Pablo Barceló, Miguel Romero, and Moshe Y. Vardi. "Semantic Acyclicity on Graph Databases". In: *SIAM J. Comput.* 45.4 (2016), pp. 1339–1376. DOI: 10.1137/15M1034714.

[52] Pablo Barceló, Diego Figueira, and Miguel Romero. "Boundedness of Conjunctive Regular Path Queries". In: *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*. Ed. by Christel Baier et al. Vol. 132. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 104:1–104:15. DOI: 10.4230/LIPICS.ICALP.2019.104.

[53] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. "An Extension of SPARQL for RDFS". In: *Semantic Web, Ontologies and Databases, VLDB Workshop, SWDB-ODBIS 2007, Vienna, Austria, September 24, 2007, Revised Selected Papers*. Ed. by Vassilis Christophides, Martine Collard, and Claudio Gutierrez. Vol. 5005. Lecture Notes in Computer Science. Springer, 2007, pp. 1–20. DOI: 10.1007/978-3-540-70960-2_1.

[54] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "nSPARQL: A navigational language for RDF". In: *J. Web Semant.* 8.4 (2010), pp. 255–270. DOI: 10.1016/J.WEBSEM.2010.01.002.

[55] Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. "How to Best Nest Regular Path Queries". In: *Informal Proceedings of the 27th International Workshop on Description Logics, Vienna, Austria, July 17-20, 2014*. Ed. by Meghyn Bienvenu et al. Vol. 1193. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 404–415.

[56] Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. "Reasonable Highly Expressive Query Languages - IJCAI-15 Distinguished Paper (Honorary Mention)". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Qiang Yang and Michael J. Wooldridge. AAAI Press, 2015, pp. 2826–2832.

[57] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. "Regular Queries on Graph Databases". In: *Theory Comput. Syst.* 61.1 (2017), pp. 31–83. DOI: 10.1007/s00224-016-9676-2.

[58] Diego Figueira and Varun Ramanathan. "When is the Evaluation of Extended CRPQ Tractable?" In: *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid Libkin and Pablo Barceló. ACM, 2022, pp. 203–212. DOI: 10.1145/3517804.3524167.

[59] Pablo Barceló, Diego Figueira, and Leonid Libkin. "Graph Logics with Rational Relations". In: *Log. Methods Comput. Sci.* 9.3 (2013). DOI: 10.2168/LMCS-9(3:1)2013.

[60] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. "Querying Graphs with Data". In: *Journal of the ACM* 63.2 (2016), 14:1–14:53.

[61] Domagoj Vrgoc. "Querying graphs with data". PhD thesis. University of Edinburgh, UK, 2014.

[62] Michael Kaminski and Nissim Francez. "Finite-Memory Automata". In: *Theor. Comput. Sci.* 134.2 (1994), pp. 329–363. DOI: 10.1016/0304-3975(94)90242-9.

[63] Mikolaj Bojanczyk. "Automata for Data Words and Data Trees". In: *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*. Ed. by Christopher Lynch. Vol. 6. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 1–4. DOI: 10.4230/LIPICS.RTA.2010.1.

[64] Patricia Bouyer, Antoine Petit, and Denis Thérien. "An algebraic approach to data languages and timed languages". In: *Inf. Comput.* 182.2 (2003), pp. 137–162. DOI: 10.1016/S0890-5401(03)00038-5.

[65] Mikolaj Bojanczyk et al. "Two-variable logic on data words". In: *ACM Trans. Comput. Log.* 12.4 (2011), 27:1–27:26. DOI: 10.1145/1970398.1970403.

[66] Diego Figueira, Artur Jez, and Anthony W. Lin. "Data Path Queries over Embedded Graph Databases". In: *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid Libkin and Pablo Barceló. ACM, 2022, pp. 189–201. DOI: 10.1145/3517804.3524159.

[67] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. "Querying graph databases with XPath". In: *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. Ed. by Wang-Chiew Tan et al. ACM, 2013, pp. 129–140. DOI: 10.1145/2448496.2448513.

[68] Serge Abiteboul and Victor Vianu. "Regular Path Queries with Constraints". In: *J. Comput. Syst. Sci.* 58.3 (1999), pp. 428–452. DOI: 10.1006/JCSS.1999.1627.

[69] Marcelo Arenas et al. "Temporal Regular Path Queries". In: *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 2412–2425. DOI: 10.1109/ICDE53745.2022.00226.

[70] Gösta Grahne, Alex Thomo, and William W. Wadge. "Preferentially Annotated Regular Path Queries". In: *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings*. Ed. by Thomas Schwentick and Dan Suciu. Vol. 4353. Lecture Notes in Computer Science. Springer, 2007, pp. 314–328. DOI: 10.1007/11965893_22.

[71] Domagoj Vrgoč. *Evaluating regular path queries under the all-shortest paths semantics*. 2022.

[72] Jorge A. Baier et al. "Evaluating Navigational RDF Queries over the Web". In: *HT*. ACM, 2017, pp. 165–174. DOI: 10.1145/3078714.3078731.

[73] Valeria Fionda, Giuseppe Pirrò, and Claudio Gutierrez. "NautiLOD: A Formal Language for the Web of Data Graph". In: *ACM Trans. Web* 9.1 (2015), 5:1–5:43. DOI: 10.1145/2697393.

[74] Alberto O. Mendelzon and Peter T. Wood. "Finding Regular Simple Paths in Graph Databases". In: *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*. 1989, pp. 185–193.

[75] Guillaume Bagan, Angela Bonifati, and Benoît Groz. "A trichotomy for regular simple path queries on graphs". In: *Symposium on Principles of Database Systems (PODS)*. Ed. by Richard Hull and Wenfei Fan. ACM, 2013, pp. 261–272.

[76] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. "Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard". In: *World Wide Web (WWW)*. 2012, pp. 629–638.

[77] Katja Losemann and Wim Martens. "The complexity of regular expressions and property paths in SPARQL". In: *ACM Trans. Database Syst.* 38.4 (2013), p. 24.

[78] Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. "Formal Language Constrained Path Problems". In: *Algorithm Theory - SWAT '98, 6th Scandinavian Workshop on Algorithm Theory, Stockholm, Sweden, July, 8-10, 1998, Proceedings*. Ed. by Stefan Arnborg and Lars Ivansson. Vol. 1432. Lecture Notes in Computer Science. Springer, 1998, pp. 234–245. DOI: 10.1007/BFB0054371.

[79] Andrea S. LaPaugh and Christos H. Papadimitriou. "The even-path problem for graphs and digraphs". In: *Networks* 14.4 (1984), pp. 507–513. DOI: 10.1002/NET.3230140403.

[80] Neil Robertson and Paul D. Seymour. "Graph Minors .XIII. The Disjoint Paths Problem". In: *J. Comb. Theory, Ser. B* 63.1 (1995), pp. 65–110. DOI: 10.1006/JCTB.1995.1006.

[81] Zhivko Prodanov Nedev and Peter T. Wood. "A Polynomial-Time Algorithm for Finding Regular Simple Paths in Outerplanar Graphs". In: *J. Algorithms* 35.2 (2000), pp. 235–259. DOI: 10.1006/JAGM.1999.1072.

[82] Wim Martens and Tina Trautner. "Bridging Theory and Practice with Query Log Analysis". In: *SIGMOD Rec.* 48.1 (2019), pp. 6–13. DOI: 10.1145/3371316.3371319.

[83] Wim Martens and Tina Trautner. "Dichotomies for Evaluating Simple Regular Path Queries". In: *ACM Trans. Database Syst.* 44.4 (2019), 16:1–16:46. DOI: 10.1145/3331446.

[84] Wim Martens, Matthias Niewerth, and Tina Trautner. "A Trichotomy for Regular Trail Queries". In: *International Symposium on Theoretical Aspects of Computer Science, (STACS)*. 2020, 7:1–7:16.

[85] Wim Martens et al. *Representing Paths in Graph Database Pattern Matching*. 2022.

[86] Wim Martens and Tina Popp. "The Complexity of Regular Trail and Simple Path Queries on Undirected Graphs". In: *PODS '22: International Conference on Management of Data, Philadelphia, PA,*

USA, June 12 - 17, 2022. Ed. by Leonid Libkin and Pablo Barceló. ACM, 2022, pp. 165–174. DOI: 10.1145/3517804.3524149.

[87] Tina Popp. "Evaluation and Enumeration of Regular Simple Path and Trail Queries". PhD thesis. University of Bayreuth, Germany, 2022.

[88] Claire David, Nadime Francis, and Victor Marsault. "Run-Based Semantics for RPQs". In: *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*. Ed. by Pierre Marquis, Tran Cao Son, and Gabriele Kern-Isberner. 2023, pp. 178–187. DOI: 10.24963/KR.2023/18.

[89] Diego Figueira and Miguel Romero. "Conjunctive Regular Path Queries under Injective Semantics". In: *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*. Ed. by Floris Geerts, Hung Q. Ngo, and Stavros Sintos. ACM, 2023, pp. 231–240. DOI: 10.1145/3584372.3588664.

[90] Oskar van Rest et al. "PGQL: a property graph query language". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. 2016, pp. 1–6.

[91] Marko A. Rodriguez. "The Gremlin graph traversal machine and language". In: *DBPL*. ACM, 2015, pp. 1–10.

[92] Wikipedia contributors. *GQL Graph Query Language*. 2020. URL: https://en.wikipedia.org/wiki/GQL_Graph_Query_Language.

[93] Shumo Chu et al. "HoTTSQL: proving query rewrites with univalent SQL semantics". In: *PLDI*. ACM, 2017, pp. 510–524. DOI: 10.1145/3062341.3062348.

[94] Paolo Guagliardo and Leonid Libkin. "A Formal Semantics of SQL Queries, Its Validation, and Applications". In: *Proc. VLDB Endow.* 11.1 (2017), pp. 27–39. DOI: 10.14778/3151113.3151116.

[95] Véronique Benzaken and Evelyne Contejean. "A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra". In: *CPP*. ACM, 2019, pp. 249–261. DOI: 10.1145/3293880.3294107.

[96] Alin Deutsch et al. "Graph Pattern Matching in GQL and SQL/PGQ". In: *SIGMOD*. ACM, 2022, pp. 1–12.

[97] *GQL Influence Graph*. https://www.gqlstandards.org/existing-languages. Accessed: 2023-01-17. 2023.

[98] Mary F. Fernandez et al. "A Query Language for a Web-Site Management System". In: *SIGMOD Rec.* 26.3 (1997), pp. 4–11. DOI: 10.1145/262762.262763.

[99] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. "Querying the World Wide Web". In: *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*. IEEE Computer Society, 1996, pp. 80–91. DOI: 10.1109/PDIS.1996.568671.

[100] Diego Calvanese et al. "Containment of Conjunctive Regular Path Queries with Inverse". In: *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000*. 2000, pp. 176–185.

[101] Nadime Francis et al. *Formal Semantics of the Language Cypher*. 2018.

[102] Nadime Francis et al. "GPC: A Pattern Calculus for Property Graphs". In: *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*. Ed. by Floris Geerts, Hung Q. Ngo, and Stavros Sintos. ACM, 2023, pp. 241–250. DOI: 10.1145/3584372.3588662.

[103] Alastair Green et al. "Updating Graph Databases with Cypher". In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2242–2253.

[104] Diego Calvanese et al. "Reasoning on regular path queries". In: *SIGMOD Record* 32.4 (2003), pp. 83–92.

[105] Diego Figueira et al. "Containment of Simple Conjunctive Regular Path Queries". In: *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 2020, pp. 371–380.

[106] Egor V. Kostylev, Juan L. Reutter, and Domagoj Vrgoc. "Containment of queries for graphs with data". In: *J. Comput. Syst. Sci.* 92 (2018), pp. 65–91. DOI: 10.1016/j.jcss.2017.09.005.

[107] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. "Query Planning for Evaluating SPARQL Property Paths". In: *SIGMOD Conference*. ACM, 2016, pp. 1875–1889. DOI: 10.1145/2882903.2882944.

[108] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. "Sparqling Kleene: Fast property paths in RDF-3X". In: *GRADES*. CWI/ACM, 2013. DOI: 10.1145/2484425.2484443.

[109] Dung T. Nguyen et al. "Join Processing for Graph Patterns: An Old Dog with New Tricks". In: *GRADES*. ACM, 2015, 2:1–2:8. DOI: 10.1145/2764947.2764948.

[110] Aidan Hogan et al. “A Worst-Case Optimal Join Algorithm for SPARQL”. In: *ISWC (1)*. Vol. 11778. Lecture Notes in Computer Science. Springer, 2019, pp. 258–275. DOI: 10.1007/978-3-030-30793-6_15.
[111] Renzo Angles et al. “PG-Keys: Keys for Property Graphs”. In: *SIGMOD ’21: International Conference on Management of Data*. ACM, 2021, pp. 2423–2436.
[112] A. Gupta and I.S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.
[113] Denilson Barbosa et al. “Efficient Incremental Validation of XML Documents”. In: *ICDE*. IEEE Computer Society, 2004, pp. 671–682. DOI: 10.1109/ICDE.2004.1320036.
[114] Renzo Angles et al. “Foundations of Modern Query Languages for Graph Databases”. In: *ACM Comput. Surv.* 50.5 (2017), 68:1–68:40.
[115] Mikolaj Bojanczyk et al. “Two-variable logic on data trees and XML reasoning”. In: *PODS*. ACM, 2006, pp. 10–19. DOI: 10.1145/1142351.1142354.
[116] Guillaume Bagan, Angela Bonifati, and Benoît Groz. “A trichotomy for regular simple path queries on graphs”. In: *J. Comput. Syst. Sci.* 108 (2020), pp. 29–48.
[117] Wouter Gelade, Wim Martens, and Frank Neven. “Optimizing Schema Languages for XML: Numerical Constraints and Interleaving”. In: *SIAM J. Comput.* 38.5 (2009), pp. 2021–2043.
[118] Katja Losemann and Wim Martens. “The complexity of regular expressions and property paths in SPARQL”. In: *ACM Trans. Database Syst.* 38.4 (2013), 24:1–24:39.
[119] Marco Console et al. “Coping with Incomplete Data: Recent Advances”. In: *PODS*. ACM, 2020, pp. 33–47. DOI: 10.1145/3375395.3387970.
[120] Albert Atserias, Martin Grohe, and Dániel Marx. “Size Bounds and Query Plans for Relational Joins”. In: *SIAM J. Comput.* 42.4 (2013), pp. 1737–1767. DOI: 10.1137/110859440.
[121] Hung Q. Ngo et al. “Worst-case Optimal Join Algorithms”. In: *J. ACM* 65.3 (2018), 16:1–16:40. DOI: 10.1145/3180143.
[122] Maarten Marx. “Navigation in XML Trees”. In: *Bull. EATCS* 88 (2006), pp. 126–140.
[123] Wim Martens et al. “Expressiveness and complexity of XML Schema”. In: *ACM Trans. Database Syst.* 31.3 (2006), pp. 770–813. DOI: 10.1145/1166074.1166076.
[124] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. “Semantics and Complexity of SPARQL”. In: *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*. Ed. by Isabel F. Cruz et al. Vol. 4273. Lecture Notes in Computer Science. Springer, 2006, pp. 30–43. DOI: 10.1007/11926078_3.
[125] Sergio Abriola et al. “Bisimulations on Data Graphs”. In: *J. Artif. Intell. Res.* 61 (2018), pp. 171–213. DOI: 10.1613/JAIR.5637.
[126] Chandan Sharma, Roopak Sinha, and Kenneth Johnson. “Practical and comprehensive formalisms for modelling contemporary graph query languages”. In: *Inf. Syst.* 102 (2021), p. 101816. DOI: 10.1016/J.IS.2021.101816.
[127] Nadime Francis et al. “A Researcher’s Digest of GQL”. In: *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*. Ed. by Floris Geerts and Brecht Vandevoort. Vol. 255. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 1:1–1:22. DOI: 10.4230/LIPICS.ICDT.2023.1.
[128] PRQL. *Pipelined Relational Query Language*. 2024. URL: https://prql-lang.org.
[129] Hadley Wickham et al. *dplyr: A Grammar of Data Manipulation*. 2023. URL: https://dplyr.tidyverse.org.
[130] Pavol Hell and Jaroslav Nešetřil. *Graphs and homomorphisms*. Oxford University Press, 2004.
[131] Moshe Y. Vardi. “A Theory of Regular Queries”. In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 2016, pp. 1–9. DOI: 10.1145/2902251.2902305.
[132] Neil Immerman. *Descriptive complexity*. Springer, 1999.
[133] Seymour Ginsburg and Edwin H. Spanier. “Semigroups, Presburger formulas, and languages.” In: *Pacific Journal of Mathematics* 16 (1966), pp. 285–296.
[134] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1979.
[135] Leonid Libkin and Limsoon Wong. “On the Power of Aggregation in Relational Query Languages”. In: *Database Programming Languages, 6th International Workshop, DBPL-6, Estes Park, Colorado, USA, August 18-20, 1997, Proceedings*. Ed. by Sophie Cluet and Richard Hull. Vol. 1369. Lecture Notes in Computer Science. Springer, 1997, pp. 260–280. DOI: 10.1007/3-540-64823-2_15.

[136] Graham Hutton. "A tutorial on the universality and expressiveness of fold". In: *Journal of Functional Programming* 9.4 (1999), pp. 355–372.

[137] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

[138] Stéphane Grumbach and Tova Milo. "Towards Tractable Algebras for Bags". In: *J. Comput. Syst. Sci.* 52.3 (1996), pp. 570–588. DOI: 10.1006/JCSS.1996.0042.

[139] Leonid Libkin and Limsoon Wong. "Query Languages for Bags and Aggregate Functions". In: *J. Comput. Syst. Sci.* 55.2 (1997), pp. 241–272. DOI: 10.1006/JCSS.1997.1523.

[140] Daniel M. Kane. *Unary Subset-Sum is in Logspace*. 2017.

[141] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[142] Harry G. Mairson. "Deciding ML typability is complete for deterministic exponential time". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. San Francisco, California, USA: Association for Computing Machinery, 1989, pp. 382–401. DOI: 10.1145/96709.96748.

[143] Angela Bonifati, Wim Martens, and Thomas Timm. "An Analytical Study of Large SPARQL Query Logs". In: *Proc. VLDB Endow.* 11.2 (2017), pp. 149–161. DOI: 10.14778/3149193.3149196.

[144] Marc Najork et al. "Of hammers and nails: an empirical comparison of three paradigms for processing large graphs". In: *Proceedings of the Fifth International Conference on Web Search and Web Data Mining, WSDM 2012, Seattle, WA, USA, February 8-12, 2012*. Ed. by Eytan Adar et al. ACM, 2012, pp. 103–112.

[145] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. "The Case Against Specialized Graph Analytics Engines". In: *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[146] Anil Pacaci et al. "Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications". In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*. Ed. by Peter A. Boncz and Josep Lluís Larriba-Pey. ACM, 2017, 12:1–12:7.

[147] Renzo Angles et al. "Benchmarking database systems for social network applications". In: *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*. Ed. by Peter A. Boncz and Thomas Neumann. CWI/ACM, 2013, p. 15.

[148] Petri Kotiranta, Marko Junkkari, and Jyrki Nummenmaa. "Performance of Graph and Relational Databases in Complex Queries". In: *Applied Sciences* 12.13 (2022).

[149] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. O'Reilly Media, 2013.

[150] William Gasarch. "Hilbert's Tenth Problem for Fixed d and n". In: *Bull. EATCS* 133 (2021).