PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
SCHOOL OF ENGINEERING

EFFICIENT PROCESSING OF RECURSIVE AND FEDERATED QUERIES IN SPARQL

ADRIÁN ANDRÉS SOTO SUÁREZ

Thesis submitted to the Office of Graduate Studies in partial fulfillment of the requirements for the Doctor in Engineering Sciences

Advisor:
JUAN L. REUTTER.

Santiago de Chile, January, 2021

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE

SCHOOL OF ENGINEERING

EFFICIENT PROCESSING OF RECURSIVE AND FEDERATED QUERIES IN SPARQL

ADRIÁN ANDRÉS SOTO SUÁREZ

Members of the Committee:

JUAN REUTTER

CRISTIÁN RIVEROS

MARCELO ARENAS

AIDAN HOGAN

BENNY KIMELFELD

JUAN DE DIOS ORTÚZAR

Thesis submitted to the Office of Graduate Studies in partial fulfillment of the requirements for the Doctor in Engineering Sciences

Santiago de Chile, January, 2021

To my grandparents.

ACKNOWLEDGEMENTS

First of all, I would like to thank to my parents Jessica and Eduardo and my sisters Constanza and Francisca for all their love and support along all these years of work. They gave me the strength to always give my best in both, my academic and my personal life. Also, I would like to thank to all my friends for their unconditional support; mainly to David and Nebil who were part of all this process and gave me a lot of good times during this period. I also have to thank to Juan C. and his podcast ASB, that helped me to stay focused during my writing time.

I also would like to thank to all my teachers and professors who inspired my curiosity through all my life. Mainly, I want to give my gratitude to the members of my committee: Juan de Dios, Benny, Aidan, Cristian and Marcelo, who provided important comments that allowed me to improve my work and also were available to discuss several parts of my research. Also, I would like to thank to Domagoj, Hans and Jorge Baier for their help, meaningful conversations and support.

I would like to thank to my advisor, Juan Reutter, for trusting me and allow me to work with him and also for giving me the opportunity of being a lecturer at the University. Every moment with you was enjoyable and I could learn a lot from you as a person and as a professional. I will not forget the meaning of the seasons change and I am certain that we will keep sharing moments that will be written in a book of *magic realism*.

Finally, I would like to thank to Valentina, who gave me all the strength that I needed during the hardest moments. This work could not be possible without you.

This work was funded by ANID - Millennium Science Initiative Program - Code ICN17_002, by CONICYT FONDECYT Regular Project Number 1170866 and by CONICYT-PCHA Doctorado Nacional 2017-21171731.

Contents

List of Figures

RESUMEN

Han pasado décadas desde los primeros pasos de la Web Semántica, y si bien, los avances han sido considerables, aún hay espacio para mejorar. En esta tesis discutimos una forma de extender SPARQL con funcionalidades recursivas, con el fin de extender el poder expresivo del lenguaje, pero también abarcar casos de uso que aún no están cubiertos. Además proponemos nuevos algoritmos que nos permiten evaluar las funcionalidades recursivas y las consultas generales de SPARQL de forma más eficiente, tanto en entornos locales como distribuidos en la web.

Este trabajo se abre con la presentación de SPARQL Recursivo, una extensión al lenguaje basado en un operador de punto fijo. Luego definimos un fragmento de este lenguaje, que es menos expresivo pero puede ser evaluado de forma más eficiente. Después mostramos cómo la idea de lenguajes recursivos puede ser utilizada para computar procedimientos de analítica de grafos con SPARQL, estudiando qué otros operadores necesita el lenguaje para llevar a cabo esta tarea. Así, proponemos el lenguaje SPARQAL, para hacer analítica de grafos dentro de bases de datos RDF.

Sin embargo, el desarrollo de estas extensiones produce una sobre carga del motor de consultas, por la cantidad de *Basic Graph Patterns* que hay que resolver. Por esta razón es que buscamos técnicas para proponer nuevos algoritmos de evaluación para este fragmento de SPARQL. Nuestras técnicas están basadas en los algoritmos de join *Worst-case optimal*, una nueva familia de algoritmos con buenas propiedades teóricas. De esta forma diseñamos e implementamos un algoritmo basado en el *Leapfrog Triejoin* que, según lo que muestran nuestros experimentos, resuelve los patrones de grafos de forma mucho más eficiente. Luego de esto, buscamos entender cómo estas técnicas de join pueden ser extendidas para entornos Web distribuidos y cómo nos pueden ayudar a integrar datos que actualmente no son accesibles para la Web Semántica.

Palabras Claves: Web semántica, SPARQL, Procesamiento de consultas.

ABSTRACT

Decades have passed since the first steps of the Semantic Web, and although there have been several improvements, there is room to grow. In this thesis we discuss a way to extend SPARQL with recursive functionalities, aiming to extend the expressive power of the language but also to cover new use-cases for it. Moreover, we propose new algorithms for computing our recursive functionalities and general SPARQL queries as efficiently as possible, in both, local and web environments.

This work opens with the presentation of Recursive SPARQL, an extension to the language based over a fixpoint operator. Then we define a less-expressive fragment for our language that can be computed efficiently. After defining this language we discuss about how recursive languages can be useful to compute Graph Analytics procedures with SPARQL, and then we study which elements SPARQL needs to perform these tasks. We finally propose SPARQAL, our language to do In-Database Graph Analytics.

However, the development of our extension produces an overload on the query engine because of the number of Basic Graph Patterns that we need to solve. For this reason we search for techniques that can improve the evaluation of such fragments of SPARQL. Our techniques are based on worst-case optimal join algorithms, a brand new family of join algorithms with good theoretical properties. Thus, we designed and implemented an algorithm based on the Leapfrog Triejoin. As our experiments show, this technique improves dramatically the evaluation of basic graph patterns. Thereafter we discuss how this join techniques can be extended for web environments, and how these techniques can help us to integrate data that currently is not available for the Semantic Web.

Keywords: Semantic web, SPARQL, Query Procesing.

Chapter 1. INTRODUCTION

The Semantic Web is an extension for the web proposed by the World Wide Web Consortium (W3C). Although the idea has been around since the sixties, the first concrete milestone of the Semantic Web was in 1998, with the release of the Resource Description Framework (RDF) and RDF Schema (RDFS) (Hogan, 2020). Since then, the Semantic Web and the tools proposed in this context have been widely adopted. For instance, it is possible to query all the data of Wikipedia by accessing it through a SPARQL endpoint, one of the Semantic Web technologies. Another use-case for the Semantic Web is to publish open data, as Tim Berners-Lee, inventor of the web and one of the main supporters of the Semantic Web, suggested the use of Semantic Web technologies as part of the 5-star deployment scheme (Berners-Lee, 2012). This is a guideline for publishing open data. Since then, several data sources have formed part of the ecosystem of the Semantic Web.

Two of the main axes in this context are RDF and SPARQL. RDF is the data model used by the Semantic Web community. One can understand this model as a graph model where nodes represent entities and the edges represent relationships between entities. SPARQL is the language for querying RDF graphs and the idea behind it is to find patterns within the graph. However, although RDF and SPARQL have been widely adopted there is still room for improvement in order to achieve a solid and reliable framework to work with the Semantic Web technologies. Going towards that direction, in this work we discuss a set of tools that aim to fill some of the current gaps. The techniques presented within this work have theoretical foundations, but we also present practical work that shows that these techniques are also feasible in real world environments.

1.1. Hypothesis and research questions

After the initial proposal of SPARQL, and with more data becoming available in the RDF format, users found use cases requiring more complex features that allow for exploring the data in more detail. For instance, one would like to get all the nodes in

the graph connected by certain path, all the nodes reachable from a starting node with some path that satisfies some constrains, or maybe all the shortest paths between two nodes. In general terms, we would like to add recursive features to SPARQL. Consider the following situation.

EXAMPLE 1.1.1. Suppose that we are visiting Buenos Aires, Argentina, and there is a concert near to the metro station Palermo. Since it is the first time we are there, we want to check all the metro stations from where we can reach the venue in order to make a reservation for our accommodation. The data to answer this query is available on Wikidata (Vrandečić & Krötzsch, 2014). We can take a look to a snapshot of the data in Figure 1.1. Here we are showing adjacent metro stations related by the property `adjacent to` (`P197` in Wikidata). We also have a node representing the Metro Line and a property that relates the metro stations with the line they belong to.

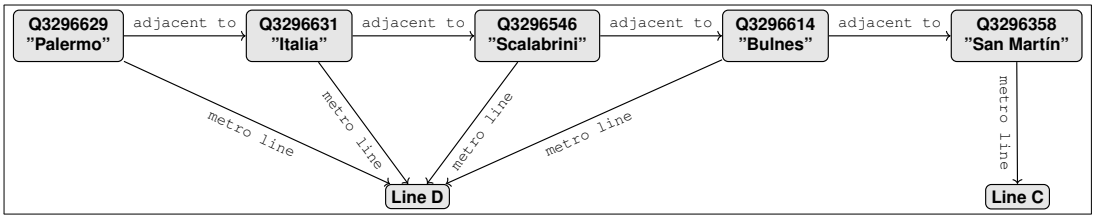


FIGURE 1.1. Adaptation of a snapshot of Wikidata with information about the Buenos Aires Metro.

Since we have access to the data, a natural question is whether or not there is a way to answer our reachability query using SPARQL. Currently it is possible to answer our question using property paths, which is an addition to SPARQL that allows us to ask for paths of arbitrary length between nodes. The resulting query is:

```
1   SELECT ?reachable WHERE {
2     wd:Q3296629 wdt:P197* ?reachable
3   }
```

In this query we note the following elements:

- `wd:Q3296629`: this is the node identifier representing the metro station "Palermo", which is part of the Buenos Aires metro.
- `wdt:P197`: this represents the property adjacent station, which in general, connects two nodes that, in fact, are adjacent metro stations in the real world

(look at Figure 1.1 for a reference). Also note that this property is together with a star ($\star$), which is the way that we indicate that we are asking for a path of arbitrary length.

- `?reachable`: this represents a variable. Thus, we are asking for all the assignations to this variable such that the pattern (after the assignment) exists within the graph.

In general, SPARQL queries work in that way; we ask for a pattern that we want to match over the graph and we return the assignment of the variables that makes the pattern be part of our RDF graph. We discuss the formal syntax and semantics of RDF graphs and SPARQL in a moment.

Another very important use case for RDF is graph analytics. If we take any RDF graph, it would be desirable to obtain extra value from the graph-like model that RDF databases support. For instance, it is reasonable to think that a user would need to obtain centrality measures (such as PageRank) for certain nodes of the graph, to compute a Clustering Coefficient or obtaining the Connected Components of the graph. And since SPARQL is a query language, one would like to execute this procedures in a declarative way.

Although all these problems have been acknowledged by the community and the W3C committee, and as we see, some solutions have been proposed towards solving them, there is room for improvement. For instance, the features added for querying paths are not expressive enough and they struggle when querying large datasets. Consider the following example which is a minimal extension to the case presented above.

EXAMPLE 1.1.2. Suppose that we are in the day of our concert at the Metro Station Palermo, but due to a strike, all the stations of the line C are closed. Now, we would like to know all the stations from where we can reach the metro station Palermo, having in mind that we cannot use stations that are part of the line C. Is there a way to achieve this task with the current SPARQL features?

If we could execute graph analytics tasks in SPARQL, one way to answer the query would be to compute the connected components of the entire metro network. Also one

could think that there is a way to express this query using property paths, but property paths cannot express this query, and moreover, they fall short when trying to express several important properties related to the navigation of RDF documents (cf. (Pérez, Arenas, & Gutierrez, 2010; Barceló, Pérez, & Reutter, 2012; Barceló, Fontaine, & Lin, 2013; Bourhis, Krötzsch, & Rudolph, 2014; Fionda, Pirrò, & Consens, 2015; Fionda, Pirrò, & Gutierrez, 2015; Fionda & Pirrò, 2017)). □

Examples like this suggest a need for improving the current techniques and strategies to process the data of the Semantic Web (Bonatti, Decker, Polleres, & Presutti, 2018). While some analytical frameworks support lightweight query features (Xin, Gonzalez, Franklin, & Stoica, 2013; Rodriguez, 2015), and some query languages support lightweight recursive and analytical features (Harris & Seaborne, 2013; Francis et al., 2018), only specific types of queries or analytics are addressed, or "glue-code" is required to combine both. Thus, we propose our first question:

Is it possible to add recursive and analytics features to SPARQL based on a fixed point operator?

It is important to note that we are seeking for non-invasive techniques, as we do not want to re-engineer the SPARQL engines, but we also need those techniques to be reliable. Thus, the previous question leads us to the following research questions:

Which is the best way to include recursive features in SPARQL? How does such an approach compares to the existing techniques?

We answer this query in Chapter 3, where we propose a language called Recursive SPARQL that adds a recursive operator for SPARQL based on fixpoint semantics. Thus we study how the linear fragment of this operator can be implemented in an efficient and non-intrusive way on top of existing SPARQL engines, and also, our experimental results show that this approach outperforms the existing techniques in several use-cases.

But when studying this recursive operator it is inevitable to think in how it can be adapted to allow SPARQL to compute graph analytics tasks, since several graph analytics procedures can be handled using the aforementioned idea of fixpoint. Thus, we derive the following research questions:

How can we include analytics features to SPARQL? Does a declarative language with analytics features scale?

Chapter A of this work presents a possible way to compute graph analytics algorithms with SPARQL. We propose an extension that is able to express any form of (computable) analytical task of interest for the user. In addition to adding recursive features to the SPARQL query language, that are a key part of our extension, we explore other features that are necessary to make our approach work. We also discuss the implementation of our approach, and for understanding how it works in practice, we design a set of experiments based on real-world use-cases. Our results provide insights into the scale and performance with which an existing SPARQL engine can perform standard graph analytics.

We noted that our approaches may struggle for larger-scale graphs. We note that the addition of recursion into RDF databases can generate an excessive load over the query engine, mainly because the computation of recursive queries implies solving a large number of basic graph patterns. Then, in order to make our techniques more reliable for large datasets, it is imperative to optimize this fragment of SPARQL. Thus, we obtain a third research question:

How can we improve the evaluation of Basic Graph Patterns in SPARQL Engines?

Though current SPARQL implementations now work well for processing large workloads of relatively simple queries (Malyshev, Krötzsch, González, Gonsior, & Bielefeldt, 2018), as we show in later experiments, they still struggle when evaluating queries with more complex patterns; we argue that this is due, in part, to the fact that prominent SPARQL engines rely on traditional algorithms that have not changed for over a decade. While we do not explore a way to resolve this, it would be difficult

for our proposed algorithms to handle large datasets. To fill this gap we explore the idea of adapting the Leapfrog Triejoin (LFTJ) (Veldhuizen, 2014) algorithm to evaluate basic graph patterns which is a worst-case optimal join algorithm with good theoretical properties. The result is a new join algorithm for SPARQL that inherits the good properties of the LFTJ. We present an implementation of this algorithm within an RDF database, and we compare its performance with respect to other engines.

As our last research question, we address the issue that, in the web context, it is rarely the case that one can obtain all the needed information from a single data source, and therefore it is necessary to draw the data from multiple sources. Although in SPARQL we have a specific operator, called SERVICE, that integrates data from distinct RDF sources, most of the data on the web is still not reachable for the Semantic Web technologies. Think about Web APIs: there are several data sources available for applications that are exposed in formats like JSON or XML but those sources cannot be retrieved by the SPARQL query language. Thus, we firmly believe that (1) it is important to have straightforward ways to incorporate these data sources to achieve a truly connected web and (2) those data sources are valuable when computing procedures of graph analytics, since they have information that currently is unavailable for the semantic web. Thus, we arrive at our last research question:

How can we allow our techniques to interact with the data exposed in the web? Is there a way to integrate the data that it is not currently available for the Semantic Web?

Consequently, at the last part of this work, we propose an extension to the SERVICE operator of SPARQL to allow the language to retrieve data from Web APIs in the same way as it connects to other SPARQL endpoints. Our extension offers the option to connect to JSON APIs and incorporate their data into SPARQL query answers. We picked JSON because it is currently one of the most popular data formats used in Web APIs, but the results presented in our work can easily be extended to any API format. By allowing SPARQL to connect to an API we can extend the query answers with data obtained from a Web service, in real time and without any setup. Use cases for such an

extension are numerous and can be particularly practical when the data obtained from the API changes very often (such as weather conditions, state of the traffic, etc.).

In summary, this work proposes new techniques for SPARQL, where these techniques are covering use-cases that in nowadays cannot be handled. Each one of our proposals have strong theoretical foundations, but also, all of our ideas have been implemented, showing that they are also feasible in practice. Now, we detail the contributions of this work.

1.2. Contributions

The main contribution of this work are the extensions that we propose to the Semantic Web technologies. Each one of our extensions is defined formally: that is, we propose the syntax and semantics for our extensions. We also study the complexity and the expressivity of each one of our solutions, giving theoretical guarantees for each one of them. Furthermore, we implement each one of our extensions to prove that our ideas not only have good theoretical properties, but are also feasible solutions for several open problems that exist in the real world. Now we present the contributions that we find in each one of the chapters.

1.2.1. Recursion in SPARQL

As we discussed before, we designed an extension to SPARQL that adds a recursive operator. We call this extension Recursive SPARQL. We first propose a general recursive operator based on fixpoint semantics, defining the syntax and the semantics of it. Then we show how to evaluate it and we study its theoretical properties. We then argue that any implementation of this general form of recursion is unlikely to perform well on real world data, so we restrict it to the so called *linear recursion*, which is well known in the relational context (Abiteboul, Hull, & Vianu, 1995; Green, Huang, Loo, & Zhou, 2013). We then demonstrate that even this restricted class of queries can express most use cases for recursion found in practice.

Next, we develop an elegant algorithm for evaluating this class of recursive queries and show how it can be implemented on top of an existing SPARQL system. For our implementation we use the Apache Jena framework (Jena, 2015) and we implement recursive queries as an add-on to the ARQ SPARQL query engine. We also use Jena TDB, which allows us not to worry about queries whose intermediate results do not fit into main memory, thus resulting in a highly reliable system.

Lastly, we test how this implementation performs on the datasets YAGO, LMDB and Wikidata, using several natural queries over these datasets. We also test our implementation using the GMark Benchmark to compare it against the property paths implementation of Jena and Virtuoso. In general terms, we show that our approach outperforms the existing approaches for computing this kind of query.

1.2.2. In-database Graph Analytics with Recursive SPARQL

As hinted at before, it is possible to extend our recursive implementation to create a (almost) declarative way to perform Graph Analytics over RDF datasets. We discuss which elements, in addition to recursion, are necessary in SPARQL to compute *graph queralytics*: tasks that combine querying and analytics on graphs, allowing to interleave both arbitrarily. We coin the term "*queralytics*" to highlight that these tasks raise new challenges and are not well-supported by existing languages and tools that focus only on querying or analytics. Rather than extending a graph query language with support for specific, built-in analytics, we rather propose to extend SPARQL to be able to express any form of (computable) analytical task of interest to the user. In addition to adding recursive features to the SPARQL query language, we explore a few other features that are necessary for this approach.

We propose a concrete syntax and semantics for this recursive language that allows us to combine querying and analytics for RDF graphs. We call our language the *SPARQL Protocol and RDF Query & Analytics Language* (*SPARQAL*). Since the recursion that we study for this approach goes beyond fixpoint, we study the expressive

power of SPARQAL. We then discuss the implementation of our language on top of a SPARQL query engine, introducing evaluation strategies that aim to find trade-offs between scalability and performance. We present experiments to compare our proposed strategies on real-world datasets, for which we devise a set of benchmark queralytics over Wikidata.

1.2.3. A worst-case optimal join algorithm for SPARQL

Since we firmly believe that such algorithms can improve the performance of RDF engines, we investigate the benefits of worst-case optimal join algorithms for evaluating basic graph patterns. We also aim for worst-case optimal join algorithms to be widely adopted on the Semantic Web in the near future, hence, we select Leapfrog Triejoin (LFTJ) (Veldhuizen, 2014) as our base algorithm since it is relatively straightforward to adapt to the case of SPARQL engine while still providing worst-case optimal guarantees. We propose extensions of the LTFJ algorithm for the SPARQL setting, proving that these adaptations do not affect the theoretical guarantees of the algorithm.

We discuss how the resulting algorithm can be integrated and optimised within a native RDF store that supports multiple index orders and cardinality-based join ordering, reducing the cost of adoption. Analogously, we create a fork of Apache Jena (TDB) (Jena, 2015) that supports worst-case join evaluation, and proceed to evaluate its performance against the unmodified version of the engine, as well as two other prominent SPARQL engines: Virtuoso (Virtuoso, 2015) and Blazegraph (Thompson, Personick, & Cutcher, 2014). We run experiments on the Berlin (Bizer & Schultz, 2009) and WatDiv (Aluç, Hartig, Özsu, & Daudjee, 2014) SPARQL benchmarks, and on a novel benchmark based on Wikidata (Vrandečić & Krötzsch, 2014) from which we generate a large set of SPARQL basic graph patterns exhibiting a variety of increasingly complex join patterns. Our results show that our fork of Apache Jena can reduce the runtimes of queries with non-trivial joins by orders of magnitude versus the baseline systems.

1.2.4. Querying APIs with SPARQL

We propose an extension to the SERVICE operator to integrate data from JSON APIs within a SPARQL query. We discuss a way to formalize the syntax and the semantics of this extension. This is done in a modular way, similar to the SPARQL formalization of (Pérez et al., 2010), making it easy to incorporate this extension into the language standard. We also show that the most likely bottleneck for our queries is the number of API calls. Thus, we design, implement and test a series of optimizations based on techniques that we present for optimizing basic graph patterns. The optimizations proposed give us a worst-case optimal algorithm for evaluating a large fragment of SPARQL patterns that uses remote SERVICE calls.

1.3. Related Work

Before starting to explore in detail each one of these contributions, we have to discuss what has been done towards the direction of our research questions. Here we present the related work for each one of the chapters.

1.3.1. Recursion in SPARQL

Property paths are the most common type of recursion implemented in SPARQL systems. This is not surprising as property paths are a part of the latest language standard and there are now various systems supporting them either in a full capacity (Gubichev, Bedathur, & Seufert, 2013; Yakovets, Godfrey, & Gryz, 2015), or with some limitations that ensure they can be efficiently evaluated, most notable amongst them being Virtuoso (Virtuoso, 2015). The systems that support full property paths are capable of returning all pairs of nodes connected by a property path specified by the query, while Virtuoso needs a starting point in order to execute the query. We would like to note that recursive queries we introduce are capable of expressing the transitive closure of any binary operator (Kostylev, Reutter, & Ugarte, 2015) and can thus be used to express property paths and any of their extensions (Kochut & Janik, 2007; Pérez et al., 2010; Anyanwu & Sheth, 2003; Fionda, Pirrò, & Consens, 2015). Regarding attempts to implement full-fledged recursion as a part of SPARQL, there

have been none as far as we are aware. There were some attempts to use SQL recursion to implement property paths (Yakovets, Godfrey, & Gryz, 2013), or to allow recursion as a programming language construct (Atzori, 2014; Motik, Nenov, Piro, Horrocks, & Olteanu, 2014), however none of them view recursion as a part of the language, but as an outside add-on.

1.3.2. In-database Graph Analytics with Recursive SPARQL

What we propose is a tool for combining graph queries and analytics, focusing on RDF graphs. One such proposal along these lines is Trinity.RDF (Zeng, Yang, Wang, Shao, & Wang, 2013), which stores RDF in a native graph format where nodes store inward and outward adjacency lists, allowing to traverse from a node to its neighbours without the need for index lookup; the system is then implemented in a distributed in-memory index, with query processing and optimization components provided for basic graph patterns. Although the authors discuss how the storage scheme of Trinity.RDF can also be useful for graph algorithms based on random walks, reachability, etc., experiments focus on SPARQL query evaluation from standard benchmarks (Zeng et al., 2013). Later work used the same infrastructure in a system called Trinity (Shao, Wang, & Li, 2013) to implement and perform experiments with respect to PageRank and Breadth-First Search, this time rather focusing on graph analytics without performing queries. Though such an infrastructure could be adapted to perform queries and analytics at scale, the authors do not discuss the combination of them, nor do they propose languages along these lines.

We also have various frameworks that have been proposed outside the scope of RDF to perform tasks of graph analytics at large-scale – involving the Web, social networks, etc. – , including GraphStep (DeLorimier et al., 2006), Pregel (Malewicz et al., 2010), HipG (Krepska, Kielmann, Fokkink, & Bal, 2011), PowerGraph (Gonzalez, Low, Gu, Bickson, & Guestrin, 2012), GraphX (Xin, Gonzalez, et al., 2013), Giraph (Ching, Edunov, Kabiljo, Logothetis, & Muthukrishnan, 2015), Signal/Collect (Stutz, Strebel, & Bernstein, 2016), and more besides. All such frameworks operate on a computational model – sometimes called the systolic model (Low et al., 2014), Gather/Apply/Scatter (GAS) model (Gonzalez et al., 2012), graph-parallel framework (Xin,

Gonzalez, et al., 2013), etc. – that involves each node in a graph recursively computing its state based on data available for its neighbouring nodes according to a given function. Although such frameworks allow for large-scale graph analytics to be applied in a distributed setting, implementing queries on such frameworks, selecting custom sub-graphs to be analysed, etc., is not straightforward. Similar computational models are used in the case of graph neural networks (Scarselli, Gori, Tsoi, Hagenbuchner, & Monfardini, 2009; Wu et al., 2019), which have been shown to be as discriminative as the (incomplete) Weisfeiler–Lehman (WL) graph isomorphism test (Xu, Hu, Leskovec, & Jegelka, 2019): in other words, by basing computation only on local information in each node's neighbourhood, there are certain pairs of non-isomorphic graphs that will return "isomorphic results" for any algorithm implemented in the framework.

1.3.3. A worst-case optimal join algorithm for SPARQL

Worst-case optimal join algorithms are a new family of join algorithms that has received much attention in the recent database literature: the state-of-the-art for join evaluation has moved away from pairwise join evaluation (Ramakrishnan & Gehrke, 2000), towards multiway join evaluation where an arbitrary number of joins can be evaluated "at once". One of the main benefits of the multiway approach is to minimize the number of intermediate results generated.

In fact, a variety of modern multiway join algorithms – including, for example, Leapfrog Triejoin (Veldhuizen, 2014), Minesweeper (Ngo, Nguyen, Re, & Rudra, 2014), Tetris (Khamis, Ngo, Ré, & Rudra, 2016), CacheTrieJoin (Kalinsky, Etsion, & Kimelfeld, 2017), etc. – have been proven to be *worst-case optimal* (Ngo, Porat, Ré, & Rudra, 2012; Ngo, Ré, & Rudra, 2013), meaning that the runtime of the algorithm is bounded by the worst-case cardinality of the query result (i.e. the AGM bound (Atserias, Grohe, & Marx, 2008)); this theoretical guarantee implies that no other join algorithm can exist that is asymptotically faster for all database instances. Several systems (e.g. Logicblox (Aref et al., 2015) and Emptyheaded (Aberger et al., 2017)) have further implemented these worst-case optimal strategies and demonstrated

their superior performance in practice for evaluating queries with complex joins. However, though work has been done on adopting such algorithms for graph queries, to the best of our knowledge, no such work has addressed the evaluation of SPARQL basic graph patterns.

1.3.4. Querying APIs with SPARQL

Pulling data from different sources is a fundamental feature of Semantic Web technologies, and there is an abundance of the literature on the subject. Most notably, SPARQL 1.1. (Harris & Seaborne, 2013) introduces the ability to federate queries using the SERVICE keyword, which permits connecting to different endpoints in order to bring in data from a diverse set of sources distributed over the Web. Fundamental studies in the area (Aranda, Arenas, & Corcho, 2011; Aranda, Arenas, Corcho, & Polleres, 2013; Aranda, Polleres, & Umbrich, 2014; Montoya, Vidal, Corcho, Ruckhaus, & Aranda, 2012; Montoya, Vidal, & Acosta, 2012) lay down the formal foundations for the SERVICE operator, and also identify main challenges for query evaluation in the distributed setting. The main conclusions regarding efficiency in this context resonate with our argument that the amount of calls to external endpoints is the main bottleneck for evaluation.

When it comes to optimizing the evaluation of federated queries, there are several approaches that aim to minimize the amount of SERVICE calls and data transfers between different endpoints. The main objective here is to generate an optimal query plan without having all the data available locally. Most approaches to this problem combine Selinger-style query optimization and a clever use of heuristics and/or data statistics for each endpoint to produce efficient execution plans (Charalambidis, Troumpoukis, & Konstantopoulos, 2015; Montoya, Skaf-Molli, & Hose, 2017; Saleem, Potocki, Soru, Hartig, & Ngomo, 2018). This is in contrast to our approach, since we are agnostic to any statistics on the queried APIs, and simply aim at minimizing the number of API calls in a worst-case optimal manner. The work in (Yannakis, Fafalios, & Tzitzikas, 2018) starts from the same assumption regarding the lack of information about the data in the endpoints, but query planning is done based on heuristics only, without providing any formal guarantees of their efficiency. Overall, what differentiates us most

from this work is that in contrast to the classical relational join optimization based on statistics and heuristics, we build our execution plans to be optimal in the worst case.

Additionally, one can view the evaluation of federated queries in a wider setting. For instance, (Hartig, Bizer, & Freytag, 2009), already proposes a way to execute SPARQL queries on the linked Web, before federation became part of the language standard, and identifies network latency as a potential stumbling stone to query execution. Similarly, (Beek, Rietveld, Schlobach, & van Harmelen, 2016), casts these ideas in a more general way, while (Verborgh et al., 2016) changes the execution paradigm by shifting the processing load from the server executing the query to the client side.

Complementary to federation in SPARQL, there is also a lot of work on bringing data from different sources into SPARQL, in a similar way as we do in this paper. Some of these approaches are based on the idea of building RDF wrappers for other formats (Kobayashi et al., 2011; Müller, Cabral, Morshed, & Shu, 2013; Rietveld & Hoekstra, 2013; Dimou et al., 2014), which is somewhat orthogonal to our approach, and can be prohibitively expensive when the API data changes often (like in Example 6.1.1). The most similar to our work are the approaches of (Fafalios & Tzitzikas, 2015; Fafalios, Yannakis, & Tzitzikas, 2016; Battle & Benson, 2008; Stringer, Meroño-Peñuela, Loizou, Abeln, & Heringa, 2015) that incorporate API data directly into SPARQL, but do not provide query execution guarantees.

Going in the other direction, there is also a lot of work on bringing SPARQL query results into Web APIs. For instance, (Lisena, Meroño-Peñuela, Kuhn, & Troncy, 2019) defines a transformation language for turning SPARQL results into JSON, while (Meroño-Peñuela & Hoekstra, 2017) allows for building generic APIs based on RDF and Linked Data.

1.4. Outline

Before starting we review the outline of this work. In Chapter 2 we present all the preliminaries that we must have in mind throughout this work. In Chapter 3 we present Recursive SPARQL, our extension for recursive queries in SPARQL. In Chapter 4 we present SPARQAL, a modified version of Recursive SPARQL to run Graph Analytics

procedures with SPARQL. In Chapter 5 we present our Worst-case Optimal Join algorithm for SPARQL, which is a variation of the Leapfrog Triejoin Algorithm (we refer to this algorithm as Jena-LFTJ). In Chapter 6 we present our algorithm SERVICE-to-API, which enables SPARQL to query JSON APIs with an algorithm that minimizes the number of calls done to the API. Finally in Chapter 7 we discuss our conclusions and we present the lessons learned.

1.5. Previous Publications

Several results presented in this work have been previously published. Here we present a detailed list of the publications related to each chapter. Also, it is possible to check the publications at `https://adriansoto.cl/research/`.

Recursion in SPARQL. A first paper was published in the conference ISWC 2015 (Reutter, Soto, & Vrgoč, 2015). We explore the idea of a recursive algebra in (Libkin, Reutter, Soto, & Vrgoč, 2018), a paper published in TODS 2018. An extended version of this work is accepted as a journal paper at The Semantic Web Journal (Reutter, Soto, & Vrgoc, 2020).

In-Database Graph Analytics with Recursive SPARQL. A conference paper (Hogan, Reutter, & Soto, 2020) for this work was presented at ISWC 2020.

A Worst-case Optimal Join Algorithm for SPARQL. A conference paper of this work was presented in ISWC 2019 (Hogan, Riveros, Rojas, & Soto, 2019).

Querying APIs with SPARQL. A conference paper of this work was presented in ESWC 2018 (Mosser, Pieressa, Reutter, Soto, & Vrgoc, 2018). An extended version of this work is accepted as a paper by the Information Systems journal (Mosser, Pieressa, Reutter, Soto, & Vrgoc, 2020). Also a demo and a workshop paper of this work were presented in ISWC 2016 (Junemann, Reutter, Soto, & Vrgoc, 2016) and AMW 2019 (Mosser, Pieressa, Reutter, Soto, & Vrgoc, 2019).

Chapter 2. PRELIMINARIES

Before starting, we present the fragment of RDF and SPARQL we work throughout this work. We first define what an RDF graph is, what operators we support and then we define their syntax and semantics.

2.1. RDF Graphs and Datasets

RDF graphs can be seen as edge-labeled graphs where edge labels can be nodes themselves, and an RDF dataset is a collection of RDF graphs. Formally, let $\mathbf{I}$ be an infinite set of *IRIs* and $\mathbf{L}$ an infinite set of *Literals*[1]. An *RDF triple* is a tuple (s, p, o) from $(\mathbf{I} \cup \mathbf{L}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{L})$, where s is called the *subject*, p the *predicate*, and o the *object*. We recall the definition of IRI given in (Arenas, Gutierrez, & Pérez, 2010). An IRI is an identifier of resources that extends the syntax of URIs to a much wider repertoire of characters for internationalization purposes. In the context of Semantic Web, IRIs are used for denoting resources.

An *RDF graph* is a finite set of RDF triples, and an *RDF dataset* is a set:

$$\{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$$

where $G_0, \ldots, G_n$ are RDF graphs and $u_1, \ldots, u_n$ are distinct IRIs (for the sake of presentation we do not consider blanks). The graph G_0 is called the *default graph*, and $G_1, \ldots, G_n$ are called *named graphs* with *names* $u_1, \ldots, u_n$, respectively. For a dataset D and IRI u we define $\mathsf{gr}_D(u) = G$ if $\langle u, G\rangle \in D$ and $\mathsf{gr}_D(u) = \emptyset$ otherwise. We also use $\mathcal{G}$ and $\mathcal{D}$ to denote the sets of all RDF graphs and datasets, correspondingly.

Given two datasets D and D' with default graphs G_0 and G'_0, we define the union $D \cup D'$ as the dataset with the default graph $G_0 \cup G'_0$ and $\mathsf{gr}_{D\cup D'}(u) = \mathsf{gr}_D(u) \cup \mathsf{gr}_{D'}(u)$ for any IRI u. Unions of datasets without default graphs is defined in the same way, i.e., as if the default graph was empty. Finally, we say that a dataset D is *contained* in a dataset D', and write $D \subseteq D'$ if (1) the default graph G in D is contained in the

[1]For clarity of presentation we do not include blank nodes in our definitions.

default graph G' in D', and (2) for every graph $\langle u, G\rangle$ in D, there is a graph $\langle u, G'\rangle$ in D' such that $G \subseteq G'$.

2.2. SPARQL

Now we define the syntax and semantics of the fragment of SPARQL that we consider in this work, which is based on the one presented in (Hernández, 2020). However, we introduce some additional fragments of SPARQL in each chapter if it is needed.

2.2.1. SPARQL Syntax

We assume a countable infinite set $\mathbf{V}$, called the set of variables, $\varnothing$ the unbound (as we discuss soon, this is a symbol to represent variables without an assignation) value and a set $\mathbf{F}$, called the set of functions, that consists in functions of the form $f : (\mathbf{I} \cup \mathbf{L} \cup \{\varnothing\})^n \to \mathbf{I} \cup \mathbf{L} \cup \{\varnothing\}$. The prefix "?" is used to denote variables (e.g., ?x).

The set of SPARQL queries is defined recursively as follows:

- An element of $(\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ is a query. Queries of this form are called *triple patterns*.
- If Q_1, Q_2 are queries, then:
  - $(Q_1 \ \mathsf{UNION} \ Q_2)$ is a query called a UNION query.
  - $(Q_1 \ \mathsf{AND} \ Q_2)$ is a query called an AND query.
  - $(Q_1 \ \mathsf{OPTIONAL} \ Q_2)$ is a query called an OPTIONAL query.
  - $(Q_1 \ \mathsf{MINUS} \ Q_2)$ is a query called a MINUS query.
- If Q is a query and g is a variable or an IRI, then $(\mathsf{GRAPH} \ g \ Q)$ is a query called a GRAPH query.
- If Q is a query and $\mathcal{X} \subset \mathbf{V}$ is a finite set of variables, then $(\mathsf{SELECT} \ \mathcal{X} \ \mathsf{WHERE} \ Q)$ is a query called a SELECT query.
- If Q is a query and φ is a SPARQL buit-in condition (see below), then $(Q \ \ \mathsf{FILTER} \ \ \varphi)$ is a query called a FILTER query. A SPARQL built-in condition (or simply a filter-condition) is defined recursively as follows:

  - An equality $t_1 = t_2$, where t_1, t_2 are elements of $\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}$, is a filter-condition.
  - If $?\mathrm{x}$ is a variable then $\mathrm{bound}(?\mathrm{x})$ is a filter-condition.
  - A Boolean combination of filter-conditions (with operators $\wedge$, $\vee$, and $\neg$) is a filter-condition.
- If Q is a query, $f : (\mathbf{I} \cup \mathbf{L} \cup \{\varnothing\})^n \to (\mathbf{I} \cup \mathbf{L} \cup \{\varnothing\})$ is a function in $\mathbf{F}$, and $?\mathrm{y}, ?\mathrm{x}_1, \ldots, ?\mathrm{x}_n$ are variables such that $?\mathrm{y}$ does not occur in Q nor in $\{?\mathrm{x}_1, \ldots, ?\mathrm{x}_n\}$, then $(Q \ \mathsf{BIND} \ f(?\mathrm{x}_1, \ldots, ?\mathrm{x}_n) \ \mathsf{AS} \ ?\mathrm{y})$ is a BIND query.

Notably, we have chosen to rule out the EXISTS keyword. This is because of a disagreement in the semantics of this operator (see the discussion in (Hernández, 2020)). However, extending all of our results for queries with EXISTS is a straightforward task.

2.2.2. Mappings and Mappings Sets

Since we have all the syntax of the language, we have to discuss the semantics, but before, we need to introduce the notion of *mappings* and some operators over *sets of mappings*. A *SPARQL mapping* is a partial function $\mu : \mathbf{V} \to \mathbf{I} \cup \mathbf{L}$. If $\mu(?x)$ is not defined, then $\mu(?x) = \varnothing$. Abusing notation, for a triple pattern t we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. Additionally, when $\mathcal{X}$ is a set of variables, and μ a mapping, we denote by $\mu_{\mathcal{X}}$ the mapping with the domain $\mathrm{dom}(\mu) \cap \mathcal{X}$, and such that $\mu_{\mathcal{X}}(?x) = \mu(?x)$, for every $?x$ in its domain.

Two mappings μ_1 and μ_2 are said to be compatible, denoted $\mu_1 \sim \mu_2$ if and only if for every common variable X holds $\mu_1(X) = \mu_2(X)$. Given two compatible mappings μ_1 and μ_2, the join of μ_1 and μ_2, denoted $\mu_1 \smile \mu_2$, is the mapping with domain $\mathrm{dom}(\mu_1) \cup \mathrm{dom}(\mu_2)$ that is compatible with μ_1 and μ_2.

Let Ω_1 and Ω_2 be two sets of mappings. Then, the operators $\bowtie$, $\cup$, $-$, and $\sqsupset\!\bowtie$ are defined over sets of mappings as follows:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \backsim \mu_2 \mid \mu_1 \in \Omega_1,\ \mu_2 \in \Omega_2,\ \mu_1 \sim \mu_2\},$$

$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},$$

$$\Omega_1 - \Omega_2 = \{\mu_1 \mid \mu_1 \in \Omega_1, \text{ and there does not exist } \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\},$$

$$\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2).$$

Given a query Q we write var(Q) to denote the set of variables occurring in Q. We use this notation also for filter-conditions, i.e., var(φ) are the variables occurring in the filter-condition φ.

2.2.3. SPARQL Semantics

Let $D = \{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$ be a dataset, G a graph among $G_0, G_1, \ldots, G_n$ and Q a SPARQL query. Then, the evaluation of Q over D (Arenas et al., 2010) with respect to G, denoted $[\![Q]\!]_G^D$, is the set of mappings recursively defined as follows:

- If t is a triple pattern, then $[\![t]\!]_G^D$ is the set of all mappings μ such that $dom(\mu) = dom(t)$ and $\mu(t) \in G$.
- If Q is $(Q_1 \text{ AND } Q_2)$ then $[\![Q]\!]_G^D = [\![Q_1]\!]_G^D \bowtie [\![Q_2]\!]_G^D$.
- If Q is $(Q_1 \text{ UNION } Q_2)$ then $[\![Q]\!]_G^D = [\![Q_1]\!]_G^D \cup [\![Q_2]\!]_G^D$.
- If Q is $(Q_1 \text{ MINUS } Q_2)$ then $[\![Q]\!]_G^D = [\![Q_1]\!]_G^D - [\![Q_2]\!]_G^D$. Note that there is a disagreement between this definition of the difference and the one adopted by the standard. The standard adopts the difference presented in (Angles & Gutiérrez, 2016) as $(Q_1 \text{ DIFF } Q_2)$, where a mapping in $[\![Q_1]\!]_G^D$ that do not share any variable with a mapping in $[\![Q_2]\!]_G^D$ is not eliminated. Instead, our definition eliminates such a mapping.
- If Q is $(Q_1 \text{ OPTIONAL } Q_2)$ then $[\![Q]\!]_G^D = [\![Q_1]\!]_G^D \mathbin{⟕} [\![Q_2]\!]_G^D$.
- If Q is $(\text{GRAPH } g\ Q')$, then $[\![Q]\!]_G^D = [\![Q']\!]_{\mathsf{gr}_D(g)}^D$, if $g \in \mathbf{I}$, or $[\![Q]\!]_G^D = \bigcup_{u \in \mathbf{I}} \left([\![Q']\!]_{\mathsf{gr}_D(u)}^D \bowtie \{g \mapsto u\} \right)$ in case of $g \in \mathbf{V}$.
- If Q is $(\text{SELECT } \mathcal{X} \text{ WHERE } Q_1)$ then $[\![Q]\!]_G^D = \{\mu_{\mathcal{X}} \mid \mu \in [\![Q_1]\!]_G^D\}$.

- If Q is $(Q_1 \text{ FILTER } \varphi)$ then $[\![Q]\!]_G^D$ is the set of mappings $\mu \in [\![Q_1]\!]_G$ such that $\mu(\varphi)$ is true. The truth value of $\mu(\varphi)$ can be **true**, **false** or **error**, and is defined as follows. Let μ be a finite mapping from variables to elements in $\mathbf{I} \cup \mathbf{L}$, φ be a SPARQL filter-condition, and t_1, t_2 be elements in $\mathbf{V} \cup \mathbf{I} \cup \mathbf{L}$. Let $\nu_\mu : \mathbf{V} \cup \mathbf{I} \cup \mathbf{L} \to \mathbf{V} \cup \mathbf{I} \cup \mathbf{L}$ be the function defined as

$$\nu_\mu(t) = \begin{cases} \mu(t) & \text{if } t \in \text{dom}(\mu), \\ \varnothing & \text{if } t \in \mathbf{V} \setminus \text{dom}(\mu), \\ t & \text{if } t \notin \mathbf{V}. \end{cases}$$

  The truth value of $\mu(\varphi)$ is defined recursively as follows:
  - If φ is an equality $t_1 = t_2$ then:
    (i) $\mu(\varphi)$ is error if $\nu_\mu(t_1) = \varnothing$ or $\nu_\mu(t_1) = \varnothing$.
    (ii) $\mu(\varphi)$ is true if $\nu_\mu(t_1) \neq \varnothing$, $\nu_\mu(t_1) \neq \varnothing$, and $\nu_\mu(t_1) = \nu_\mu(t_2)$.
    (iii) $\mu(\varphi)$ is false if $\nu(t_1) \neq \varnothing$, $\nu_\mu(t_1) \neq \varnothing$, and $\nu_\mu(t_1) \neq \nu_\mu(t_2)$.
  - If φ is an expression of the form $\text{bound}(?\text{x})$ then $\mu(\varphi)$ is true if $?\text{x} \in \text{dom}(\mu)$. Otherwise, $\mu(\varphi)$ is false.
  - If φ is a Boolean combination of conditions using operators $\wedge$, $\vee$ and $\neg$, then the truth value of $\mu(\varphi)$ is the usual for 3-valued logic (where error is interpreted as unknown).
- If Q is $(Q_1 \text{ BIND } f(?\text{x}_1, \ldots, ?\text{x}_n) \text{ AS } ?\text{y})$ then let μ be a mapping and y_μ be the value $f(\mu(?\text{x}_1), \ldots, \mu(?\text{x}_n))$. Then, $[\![Q]\!]_G^D$ is the set of SPARQL mappings

$$\{\mu \in [\![Q_1]\!]_G^D \mid \text{when } y_\mu \text{ is not defined}\} \cup$$
$$\{\mu \backsim \{?\text{y} \mapsto y_\mu\} \mid \mu \in [\![Q_1]\!]_G^D \text{ and } y_\mu \text{ is defined}\}.$$

2.3. RDF Datasets and Engines

Throughout this work we use several data sources and RDF engines. In this section we present a brief introduction of all the datasets, benchmarks and engines that we use, and we also mention in which section are they mentioned.

2.3.1. RDF Datasets

To prepare the experiments for each one of the techniques presented in this work we used several datasets. Our primary testing dataset is the graph of Wikidata, which is a knowledge graph that contains all the information available in Wikipedia. We also tested with two other widely used knowledge graphs: Yago and LMDB. The first one stores several facts related to multiple domains, while the second stores information about the film industry. In terms of benchmarks we used Berlin, WatDiv, GMark and Graphalytics. The first two are benchmarks for RDF databases, while the third is a benchmark to test path queries in several data models. The last one is a benchmark to test the performance of graph analytics procedures, such as PageRank, BFS and Clustering Coefficient. Now we detail each one of the datasets and benchmark.

Wikidata. Wikidata is a knowledge graph created in 2012 and hosted by the Wikimedia Foundation (Vrandečić & Krötzsch, 2014). It is based on all the data that one can find within Wikipedia. As hinted before, this graph is stored as an RDF graph and one can access it through a public endpoint (*Wikidata Query Service*, n.d.).

In this graph we have entities connected by properties. For instance, the movie **Interstellar** can be found as the node:

<http://www.wikidata.org/entity/Q13417189>.

As we may expect, this node is connected to the node

<http://www.wikidata.org/entity/Q11424>,

which means being a **film**. The property that goes from **Interstellar** to the node **film** is the property **instance of**, represented by the URI

<http://www.wikidata.org/prop/direct/P31>.

In general, all the entities are URIs identified by a number preceded by the letter Q, while the properties connecting nodes are identified by numbers preceded by the letter P.

Since this graph contains billions of edges, we use it several times to test how our ideas work with large datasets. Also, as this graph is one of the most famous RDF datasets, we use it to give several examples.

Yago and LMDB. Yago (YAGO, n.d.) is a knowledge graph stored as an RDF dataset that contains information about facts of different domains, while LMDB (LMDB, n.d.) is a graph that stores information about movies, actors, directors and many other entities related to cinema. We use these datasets to perform some experiments for our recursive language.

Graphalytics. Graphalytics (LDBC, 2019) is a benchmark used to test graph analysis platforms. The benchmark consists of several datasets that are graphs from the real world and synthetic data. The size of the graphs goes from thousands to trillions of nodes. They propose a set of six algorithms and they provide the answer of those algorithms for each one of the graphs.

GMark. GMark (Bagan et al., 2017) is a domain and query language-independent framework that generates graph instances and graph query workloads. The main purpose of this benchmark is to test property path queries. Thus, we mainly use this benchmark to compare Recursive SPARQL against the existing approaches for recursive queries based on property paths.

The Berlin Benchmark and WatDiv. The Berlin Benchmark (Bizer & Schultz, 2009) is a benchmark for RDF systems that contains a fixed set of query templates to test several features of SPARQL. The data is generated synthetically, and it is based on an e-commerce use case. We use this benchmark to test our worst-case optimal algorithm for BGPs and to try our extension of the service operator.

WatDiv (Aluç et al., 2014) is a benchmark to test the evaluation of BGPs in RDF databases. This benchmark consists on a set of several query templates, and each

query template can generate various queries with the same structure. The data for this benchmark is synthetical, and the size can be customized by the user.

A summary of the used datasets can be found in table 2.1.

Data Source	Rec-SPARQL	SPARQAL	Jena-LFTJ	SERVICE-to-API
Wikidata	✓	✓	✓	✓
LMDB & Yago	✓			
GMark	✓			
Graphalytics		✓		
WatDiv			✓	
Berlin			✓	✓

TABLE 2.1. Data sources used throughout this work.

We note that we use Wikidata in each one of the chapters, since there are several properties of this dataset that makes it a good choice for testing our prototypes: it has billions of nodes, it has heterogeneous data from different domains and it is one of the most used datasets within the Semantic Web community. With respect to Yago and LMDB, we used them for testing our recursive implementation since they contain several path-like patterns and they are lightweight datasets for testing our in-memory approach.

Regarding the benchmarks, we use GMark to compare our recursive extension against the existing approaches, since it is a benchmark for comparing path functionalities across different data models, such as RDF, Property Graphs and Relational Databases. As we mentioned before, Graphalytics is a benchmark for testing graph analytics procedures, thus we use it for testing our extension for computing Graph Analytics with SPARQL. WatDiv is a benchmark that aims to test the evaluation of BGPs in RDF databases, which we use to compare our WCO join algorithm with respect to the join algorithms used by the most used RDF databases. Finally, the Berlin benchmark is a benchmark to test several components of an RDF database. Thus we used it to understand how our WCO join algorithm and our SERVICE extension interacted with the other components of the query engine.

2.3.2. RDF Engines

Since the RDF model has increased in popularity through the years, there are several engines that can handle RDF graphs and SPARQL queries. We use 3 these engines in this work:

- **Apache Jena**: this engine is an open-source Semantic Web framework written in Java (Jena, 2015). It provides several features such as persistent storage, a query engine and an HTTP interface. All the prototypes created for this work were implemented by modifying and extending the source code of this engine.
- **Openlink Virtuoso**: this is a database engine that combines functionalities from relational databases and RDF databases (Virtuoso, 2015), among other data models. It is written in C and it has an open-source version. It is one of the most widely used systems for working with knowledge graphs.
- **Blazegraph**: Blazegraph is both, an RDF store platform and a graph databases (Thompson et al., 2014). It is mainly known for being the database system used for exposing the Wikidata SPARQL endpoint.

A summary of engines used per chapter can be found in table 2.2.

Data Source	Rec-SPARQL	SPARQAL	Jena-LFTJ	SERVICE-to-API
Apache Jena	✓	✓	✓	✓
Virtuoso	✓	✓	✓	
Blazegraph			✓	

TABLE 2.2. RDF engines used throughout this work.

As we note, Blazegraph is used only in Chapter 5. This is because in such section we do experiments of our worst-case optimal algorithm for BGPs with the graph of Wikidata, and, since Blazegraph is the database used to expose the endpoint of this graph, we had to compare our algorithm against this engine. We also note that in Chapter A we only use Apache Jena. This is because we are implementing a feature that goes beyond to any existing feature of RDF engines, and thus, there is no straightforward way to compare our implementation with other engines.

Chapter 3. RECURSION IN SPARQL

In this chapter we talk about our **Recursive SPARQL**, our extension to SPARQL with a recursive operator based on fixpoint semantics. We opt for this alternative because this approach is the one that SQL databases have, it has been used successfully since the release of SQL 99 and its implementation can be done in a non-intrusive way.

First we show how we can answer the query of the Example 1.1.2 using Recursive SPARQL, and then, we present our formalization for the general recursive operator. We also present an algorithm for evaluating recursive queries. We also study the theoretical properties of our operator: we identify the fragment where our operator converges and we include a complexity analysis which shows that our recursive queries are PTIME-complete in data complexity. Then we study an optimization for the recursive operator by restricting it to the "Linear Recursion" fragment. Linear recursive queries has been studied before, since it is the approach that SQL databases use for computing recursive queries. Thus, we develop an algorithm for computing this class of recursive queries and then we show how to implement it on top of any RDF database.

Our implementation relies on the ARQ query engine, which is part of the open-source database Apache Jena. To understand how our approach compares with respect to property paths, we develop several experiments. First we tried our implementation with queries from real use-cases over the datasets of YAGO and LMDB. We then test the implementation using a benchmark designed for testing property paths: the GMark Benchmark. We finally stress out our implementation by doing experiments over the entire graph of Wikidata. All our experiments show the same trend: although our implementation offers more expressive power than the current techniques, it performs as well as those techniques. Moreover, our approach outperforms property paths in several use-cases, showing that it is worth to have a recursive operator within SPARQL.

3.1. Motivating the need of full recursion

As we discussed in the introduction, we can express certain reachability queries in SPARQL using techniques such as property paths, but there are some tasks where those

```
1  WITH RECURSIVE http://db.ing.puc.cl/connected AS {
2    CONSTRUCT { ?x ex:conn ?y } WHERE {
3      ?x ex:adjacent_to ?y . ?x ex:metro_line ?line .
4      ?y ex:metro_line ?line2 .
5      FILTER ( ?line != "Line C" && ?line2 != "Line C" )
6    }
7  }
8  WITH RECURSIVE http://db.ing.puc.cl/reachable AS {
9    CONSTRUCT { ?x ex:reachable ?y } WHERE {
10     { GRAPH <http://db.ing.puc.cl/connected> { ?x ex:conn ?y } }
11     UNION {
12       GRAPH <http://db.ing.puc.cl/connected> { ?x ex:conn ?aux } .
13       GRAPH <http://db.ing.puc.cl/reachable> { ?aux ex:conn ?y }
14     }
15   }
16 }
17 SELECT * WHERE {
18   GRAPH <http://db.ing.puc.cl/reachable> { ?x ?y ?z }
19 }
```

FIGURE 3.1. Recursive query that answers the question of Example 1.1.2.

approaches fall short; recall the Example 1.1.2 presented before. We have the metro network of Buenos Aires (a snapshot of such network is in Figure 1.1) and we want to know whether is a way to reach the station Palermo without using the stations of line C. We already know that this query cannot be answered with the current specification of SPARQL, so, we designed a full recursive operator based on fixpoint semantics that, in fact, has more expressive power and can answer the query.

EXAMPLE 3.1.1. We first explain the intuition of how our recursive operator works and then we begin to formalize this approach. One can answer the question of 1.1.2 with the recursive query of Figure 3.1.

Each **WITH RECURSIVE** clause is constructing a temporary graph. The first one is named `http://db.ing.puc.cl/connected` and is a graph that contains all the nodes that are adjacent stations such that they do not belong to Line C. While the second one is `http://db.ing.puc.cl/reachable`, that basically is computing all the pair of nodes reachable from one to the other according to the graph `http://db.ing.puc.cl/connected`. Finally we do a SELECT query that is

calling to the second temporary graph. Thus, we are answering the question of Example 1.1.2.

Note that within the second **WITH RECURSIVE** clause we are calling to the graph that we are defining, in this case the graph `http://db.ing.puc.cl/reachable`. In fact, that is the recursive part: to define a graph in terms of itself. Another things to note is (1) that the first **WITH RECURSIVE** clause is not recursive, since we are not using the graph `http://db.ing.puc.cl/connected` within the clause and (2) we are not using SELECT queries to compute recursive graphs. Instead we are using CONSTRUCT queries. We discuss this later in this chapter.

Recursive queries has been introduced before for others query languages. In 1999 the `WITH RECURSIVE` clause was added to SQL since it was not possible to express certain kind of queries, such as the ones about reachability. The idea behind it is to define a temporary table T in terms of the result of a query Q, but this query uses the table T (this is why this procedure is *Recursive*). The table T is initially empty, but when we execute Q for the first time, and if the result is not empty, then the instance of the temporary table will be equals to the result of Q. We can repeat this procedure several times, and in each one of the iterations the value of T is going to be overwritten. Normally, when Q has the same result in two consecutive iterations, we return the value of the recursive query as the last instance of T. It is important to note that this addition to SQL was straightforward since the result of a query is a table and therefore, a temporary table can be defined in terms of a SQL query.

One possible reason why recursion was not previously considered as an integral operator of SPARQL could be the fact that in order to compute recursive queries we need to apply the query to the result of a previous computation. However, typical SPARQL queries do not have this capability as their inputs are RDF graphs but their outputs are mappings. This hinders the possibility of a fixed point recursion as the result of a SPARQL query cannot be subsequently queried. However, as hinted by the example, the existence of the CONSTRUCT operator allows us to handle this issue.

3.2. The CONSTRUCT operator

The CONSTRUCT operator receives a graph as input, but the output is an RDF graphs as well, and indeed (Kostylev, Reutter, & Ugarte, 2015) has proposed a way of defining a fixed point like extension for SPARQL based on this idea. A SPARQL CONSTRUCT query, or *c-query* for short, is an expression

$$\text{CONSTRUCT } H \text{ WHERE } Q,$$

where H is a set of triples from $(\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V})$, called a *template*[1], and Q is a SPARQL query.

The idea behind the CONSTRUCT operator is that the mappings resulting of evaluating Q over the dataset are used to construct an RDF graph according to the template H: for each such mapping μ, one replaces each variable $?x$ in H for $\mu(?x)$, and add the resulting triples to the answer. Since all the patterns in the template are triples we will end up with an RDF graph as desired. We illustrate how they work by means of an example.

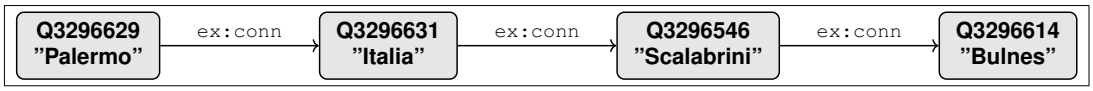


FIGURE 3.2. Result of the CONSTRUCT query from the Example 3.2.1.

EXAMPLE 3.2.1. Let G be the graph in Figure 1.1. Recall the first part of the recursive query that we discussed before: we want a graph that contains only the adjacent stations that do not belong to the Line C. This would be achieved by the following SPARQL CONSTRUCT query q:

```
1 CONSTRUCT { ?x ex:conn ?y } WHERE {
2   ?x ex:adjacent_to ?y . ?x ex:metro_line ?line .
3   ?y ex:metro_line ?line2 .
4   FILTER ( ?line != "Line C" && ?line2 != "Line C" )
5 }
```

[1]For brevity we leave out FROM NAMED constructs, and we leave the study of blanks in templates as future work.

We call $\mathsf{ans}(\boldsymbol{q}, D)$ the result of evaluating $\boldsymbol{q}$ over D. In this case, it is depicted in Figure 3.2.

Formally, the *answer* $\mathsf{ans}(q, D)$ to a c-query of the form

$$q = \mathsf{CONSTRUCT}\ H\ \mathsf{WHERE}\ Q$$

over a dataset D with default graph G_0 is the RDF graph consisting of all triples in $\mu(H)$, for each mapping μ in $[\![Q]\!]^D_{G_0}$.

For readability, we will use Q to refer to SPARQL queries using the SELECT form, and $\boldsymbol{q}$ for c-queries whenever it is convenient to use this distinction. However, we often deal with queries that can be of either form. In this case, we use the notation $\mathsf{ans}(q, D)$ to speak of the answer of $\boldsymbol{q}$ over dataset D. it is defined as above if $\boldsymbol{q}$ is a c-query, and we define $\mathsf{ans}(q, D) = [\![\boldsymbol{q}]\!]^D_{G_0}$, for G_0 the default graph of D, when $\boldsymbol{q}$ is a query of the SELECT form.

3.3. Adding recursion to SPARQL

The most basic example of a recursive query in the RDF context is that of reachability: given a resource x, compute all the resources that are reachable from x via a path of arbitrary length. However, a full recursive operator would allow us to perform several other queries. To understand what can we can do with our operator, now we define it formally.

3.3.1. A Fixed Point Based Recursive Operator

We have decided to implement a different approach and define a more expressive recursive operator that allows us compute the fixed point of a wide range of SPARQL queries. As we mention earlier, this is based on the recursive operator that was added to SQL when considering similar challenges. We cannot define this type of operator for SPARQL SELECT queries, since these returns mappings and thus no query can be applied to the result of a previous query, but we can do it for CONSTRUCT queries, since these return RDF graphs. Following (Kostylev, Reutter, & Ugarte, 2015), we now define the language of *Recursive Queries*. Before proceeding with the formal

definition we illustrate the idea recalling the Example 3.1.1. Let us explain in detail how this query works.

The recursive query of Figure 3.1 has three parts.

(i) From lines 1 to 7 we have the first recursive query. We are creating a temporary graph called `http://db.ing.puc.cl/connected` which is equal to the result of the construct query described from line 2 to 6. We note that this CONSTRUCT is the same that was presented in 3.2.1, and its result contains all the triples (M_1, C, M_2) such that M_1 and M_2 are metro stations that do not belong to Line C and are adjacent one from the other.

(ii) From lines 8 to 16 we have the second recursive query which is defined in terms of itself. First we initialize an empty temporary graph called:

`http://db.ing.puc.cl/reachable`

but at the first iteration we initialize this recursive graph with all the triples from the graph:

`http://db.ing.puc.cl/connected`

Then comes the recursive part: if (M_1, C, X) and (X, C, M_2) are triples in the temporary graph, then we also add (M_1, C, M_2) to the temporary graph. We continue iterating until a fixed point is reached, and finally we obtain a graph that contains all the triples (M_1, C, M_2) such that the station M_1 is linked to the station M_2 and we do not use any stations of the line C.

(iii) Finally, the SELECT query extracts all such pairs of metro stations from the constructed graph.

As hinted at the example, the following is the syntax for recursive queries:

DEFINITION 3.3.1 (Syntax of recursive queries). A *recursive SPARQL query*, or just recursive query, is either a SPARQL query or an expression of the form

$$\mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2, \tag{3.1}$$

where t is an IRI from $\mathbf{I}$, $\boldsymbol{q}_1$ is a c-query, and $\boldsymbol{q}_2$ is a recursive query. The set of all recursive queries is denoted rec-SPARQL.

Note that in this definition $\boldsymbol{q}_1$ is allowed to use the temporary graph t, which leads to recursive iterations. Furthermore, the query $\boldsymbol{q}_2$ could be recursive itself, which allows us to compose recursive definitions.

As usual with this type of query, the semantics is given via a fixed point iteration.

DEFINITION 3.3.2 (Semantics of recursive queries). Let $\boldsymbol{q}$ be a recursive query of the form (3.1) and D an RDF dataset. If $\boldsymbol{q}$ is a non recursive query then $\mathsf{ans}(\boldsymbol{q}, D)$ is defined as usual. Otherwise the *answer* $\mathsf{ans}(\boldsymbol{q}, D)$ is equal to $\mathsf{ans}(\boldsymbol{q}_2, D_{\text{LFP}})$, where D_{LFP} is the least fixed point of the sequence $D_0, D_1, \ldots$ with $D_0 = D$ and

$$D_{i+1} = D \cup \text{RG}_i, \text{ for } i \geq 0.$$

where RG_i is equals to the graph with IRI t and the triples are given by the result of $\mathsf{ans}(\boldsymbol{q}_1, D_i)$. When D_{LFP} exists and is a finite set (of finite graphs), we say that the recursive query $\boldsymbol{q}$ *converges* over D.

Since we are working with finite graphs and datasets, the notion of a fixed point here is simpler than the one defined for complete lattices. Here we can order datasets using the subset ($\subseteq$) relation, and thus, we can define a fixed point. This has been studied before in (Libkin, 2004).

In this definition, D_1 is the union of D with a temporary graph t that corresponds to the evaluation of q_1 over D, D_2 is the union of D with a temporary graph t that corresponds to the evaluation of q_1 over D_1, and so on until $D_{i+1} = D_i$. Note that the temporary graph is completely rewritten after each iteration. This definition suggests the pseudocode of Algorithm 1 for computing the answers of a recursive query $\boldsymbol{q}$ of the form (3.1) over a dataset D[2].

[2]For readability we assume that t is not a named graph in D. If this is not the case then the pseudocode needs to be modified to meet the definition above

Algorithm 1 Computing the answer for recursive c-queries of the form (3.1).

Input: Query Q of the form (3.1), dataset D
Output: Evaluation $\mathsf{ans}(Q, D)$ of Q over D

```
1: G_temp ← ∅ named after IRI t
2: loop
3:     G_Temp ← ans(q_1, D ∪ {⟨t, G_Temp⟩})
4:     if ans(q_1, D ∪ {⟨t, G_Temp⟩}) = G_Temp then
5:         break
6:     end if
7: end loop
8: return ans(q_2, D ∪ {⟨t, G_Temp⟩})
```

To clarify Definition 3.3.2, we show how the temporary graph

<http://db.ing.puc.cl/reachable>

evolves during the execution of the query from Figure 3.1, when evaluated the graph in Figure 1.1.2. The different values of temporary graph are show in Figure 3.3. Here we call G^i_{Temp} the instance of G_{Temp} at the i^{th} iteration of the loop presented in the Algorithm 1. We have two things to note:

(i) G^0_{Temp} is an empty graph.

(ii) G^1_{Temp} has the same triples as the graph:

<http://db.ing.puc.cl/connected>

which is the result of the construct query presented in the Example 3.2.1. We can think of this instance as the paths of length one.

(iii) We can think about the graphs G^2_{Temp} and G^3_{Temp} as as graphs where we add the paths of length two and three respectively.

(iv) Note that since we are working with graphs, there are no duplicated triples. Finally, we have $G^3_{\text{Temp}} = G^4_{\text{Temp}}$, and thus we stop the loop at the fourth iteration.

Obviously, the semantics of recursive queries only makes sense as long as the required fixed point exists. Unfortunately, we show in the following section that there are queries for which this operator indeed does not have a fixed point. Thus, we need to

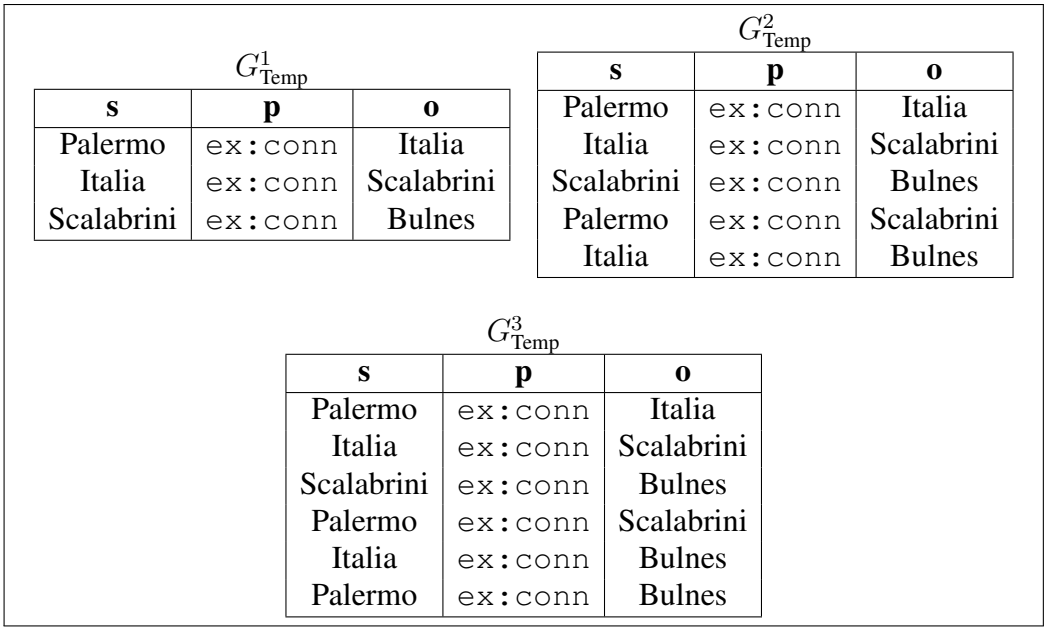


FIGURE 3.3. The step-by-step evaluation of a recursive graph.

restrict the language that can be applied to such inner queries[3]. We also discuss other possibilities to allow us to use any operator we want.

3.3.2. Ensuring fixed point of queries

If we want to guarantee the termination of Algorithm 1, we need to impose two conditions. The first, and most widely studied (Libkin, 2004), is that query $\boldsymbol{q}_1$ must be *monotone*: a c-query $\boldsymbol{q}$ is monotone if for all pair of datasets D_1, D_2 where $D_1 \subseteq D_2$ it holds that $\mathsf{ans}(\boldsymbol{q}, D_1) \subseteq \mathsf{ans}(\boldsymbol{q}, D_2)$. However, we also need to impose that the recursive c-query $\boldsymbol{q}_1$ preserves the domain: we say that a c-query $\boldsymbol{q}$ preserves the domain if there is a finite set S of IRIs such that, for every dataset D, the IRIs in $\boldsymbol{q}(D)$ either come from S or are already present in D. Let us provide some insight about the need for these conditions.

Monotonicity. The most typical example of a problematic non-monotonic behaviour is when we use negation to alternate the presence of some triples in each iteration of the recursion, and therefore come up with recursive queries where the fixed point does not exists.

[3]It should be noted that the recursive SQL operator has the same problem, and indeed the SQL standard restricts which SQL features can appear inside a recursive operator.

EXAMPLE 3.3.1. Consider the following query that contains a MINUS clause.

```
WITH RECURSIVE http://db.puc.cl/temp
AS {
  CONSTRUCT {?x ?y "a"}
  WHERE {
    { ?x ?y ?z } MINUS {
      GRAPH <http://db.puc.cl/temp> {
        { ?x ?y "a" }
      }
    }
  }
}
SELECT * WHERE {
  GRAPH <http://db.puc.cl/temp> {
    ?x ?y ?z
  }
}
```

Also consider a instance for the default graph with only one triple:

```
:s :p "b"
```

In the first iteration, the graph `<temp>` would have the triple:

```
:s :p "a"
```

but in the next iteration the graph `<temp>` will be empty because of the MINUS clause. Then, the `<temp>` graph will be alternating between an empty graph and a graph with the triple `:s :p "a"`. Thus, the fixed point does not exist for this query.

Similar examples can be obtained with other SPARQL operators that can simulate negation, such as MINUS or even arbitrary OPTIONAL (Angles & Gutierrez, 2016; Kaminski & Kostylev, 2016).

Preserving the domain. The BIND clause allows us to generate new values that were not in the domain of the database before executing a recursive query. Since completely

new values may be generated for the temporary graph at each iteration, this may also imply that a (finite) fixed point may not exists, even if the query is monotone.

EXAMPLE 3.3.2. Consider the following query that makes use of the BIND clause.

```
1  WITH RECURSIVE http://db.puc.cl/temp
2  AS {
3    CONSTRUCT {?x :number ?b}
4    WHERE {
5      { ?x :type :person .
6        ?x :age ?a . BIND (?a AS ?b) }
7      UNION {
8        GRAPH <http://db.puc.cl/temp> {
9          ?x :number ?aux .
10         BIND(?aux + 1 AS ?b)
11       }
12     }
13   }
14 }
15 SELECT * WHERE {
16   GRAPH <http://db.puc.cl/temp>{
17     ?x ?y ?z
18   }
19 }
```

The base graph stores the age for all the people in the database. In each iteration, we will increase by one all the objects in our graph and then we will store triples with those new values. As we mentioned, in each iteration the query will try to insert new triples into the database, and will thus never terminate adding new triples into the dataset.

On the other hand, it is easy to verify that the query from Example 3.3.2 is monotone. By the Knaster-Tarski theorem (Tarski, 1955), a monotone query always has a fixed point, however, such a fixed point need not be a finite dataset. Indeed, the fixed point for the query in Example 3.3.2 would have to contain triples linking each person

in the original dataset with all the numbers larger than her or his initial age. This would make the fixed point infinite and thus not be a valid RDF graph.

One way to ensure that a fixed point of a monotone query is necessarily finite, is to make the underlying domain over which it operates finite. For instance, in Example 3.3.2, we are assuming that the domain over which queries operate is the set of all possible triple over $\mathbf{I} \cup \mathbf{L}$, and not just the ones using IRIs and literals from the queried dataset. On the other hand, in the case of domain preserving queries, when considering the sequence $(D_i)_i$ from Definition 3.3.2, our monotone queries can only construct triples over a finite set of IRIs and literals (the initial dataset, plus another finite set), thus making the fixed point necessarily finite. Besides the BIND operator, we can also simulate the creation of new values by means of blanks in the construct templates, or even with blanks inside queries or subqueries.

Existence of a fixed point. If we are working with queries that are both monotone and domain preserving, we can guarantee that the sequence $(D_i)_i$ from Definition 3.3.2 always converges to a well defined RDF dataset. More precisely, as an immediate consequence of the Knaster-Tarski theorem (Tarski, 1955), we can obtain the following:

PROPOSITION 3.3.1. Let D be a dataset and $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ two monotone queries that are domain preserving, and let $\boldsymbol{q} = \mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2$ be a recursive query. Then $\boldsymbol{q}$ converges over D, and we can use Algorithm 1 to evaluate $\boldsymbol{q}$.

Next, we study which SPARQL queries are both domain preserving and monotone, in order to restrict the recursion to such queries.

3.3.3. Fragments where the recursion converges

We know that monotonicity and domain preservation allows us to define a class of recursive queries which will always have a least fixed point. The question then is: how to define a fragment of SPARQL that is both monotonic and domain preserving?

First, how can we guarantee that queries are monotonic? An easy option here is to simply disallow all the operators which can simulate negation such as OPTIONAL, MINUS, or negative FILTER conditions. Second, when it comes to guaranteeing that

queries are domain preserving, we can simply prohibit them to use operators that can create new values such as BIND, or to use blanks in construct templates, queries or subqueries. This leads us to a first subclass of SPARQL queries for which can be used inside recursive queries.

DEFINITION 3.3.3 (positive SPARQL and rec-SPARQL). A SPARQL query is *positive* if it does not use any of the following operators:

(i) It does not use operators OPTIONAL, MINUS, BIND or blanks in construct templates;

(ii) Every construct $(Q \text{ FILTER } \varphi)$ is such that φ uses only equalities and positive boolean combinations using $\wedge$ and $\vee$; and

(iii) In every subquery $(\text{SELECT } \mathcal{X} \text{ WHERE } Q)$ we have that Q is also positive.

Likewise, a positive c-query is a c-query using positive SPARQL in its definition. The language of positive rec-SPARQL comprises every positive SPARQL query, and also queries of the form

$$\text{WITH RECURSIVE } t \text{ AS } \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2, \tag{3.2}$$

where t is an IRI from $\mathbf{I}$, $\boldsymbol{q}_1$ is a positive c-query, and $\boldsymbol{q}_2$ is a positive rec-SPARQL query.

Given that positive SPARQL queries are both monotone and domain preserving, as an easy corollary of Proposition 3.3.1, we obtain the following:

PROPOSITION 3.3.2. If we have a recursive query

$$\boldsymbol{q} = \text{WITH RECURSIVE } t \text{ AS } \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2$$

where $\boldsymbol{q}_1$ a positive query, and $\boldsymbol{q}_2$ is a positive rec-SPARQL query, then $\boldsymbol{q}$ converges over D.

While positive queries do the trick, one might argue that they are quite restrictive. We can therefore wonder, whether it is possible to allow some form of negation, or the use of OPTIONAL?

In fact, literature has pointed out some more milder restrictions, that still do the trick. For instance, even though we disallow OPTIONAL, we note that our fragment of SPARQL is expressive enough to express rec-SPARQL queries in which $\boldsymbol{q}_1$ is given by positive SPARQL with well-designed optionals (Pérez et al., 2010). This is because for CONSTRUCT queries, the fragment we consider has been shown to contain queries defined by union of well designed graph patterns (Kostylev, Reutter, & Ugarte, 2015)).

The second, more expressive fragment to consider, loosens the restrictions we put on the use of negation. The idea is that bad behaviour of negation, such as the one shown in Example 3.3.1, only occurs when negation involves the same graph that is being constructed in the fixed point. Therefore, we will consider a fragment of rec-SPARQL queries which allow negation whenever it does not involve the graph being constructed in the recursive portion of the query.

DEFINITION 3.3.4 (Stratified positive rec-SPARQL). Stratified positive rec-SPARQL extends the language of positive rec-SPARQL by allowing queries of the form (3.2) above in which $\boldsymbol{q}_1$ can use constructs of the form $(Q_1 \ \mathsf{MINUS} \ Q_2)$, as long as every expression $(\mathsf{GRAPH}\ g\ Q)$ in Q_2 is such that g is an IRI different from t.

Essentially, with stratified positive rec-SPARQL we allow some degree of negation, but we need to make sure this negation does not involve the temporal graph under construction.

While stratified positive rec-SPARQL need not be monotone with respect to the domain of all possible RDF datasets, it is monotone in the context of the sequence $(D_i)_i$ from Definition 3.3.2. More precisely, any stratified positive rec-SPARQL of the form (3.2) is such that $\boldsymbol{q}_1$ is monotone with respect to the named graph t, meaning that, for datasets D and D' that only differ in the named graph t, if $\langle t, G\rangle$ is the graph named t in D, $\langle t, G'\rangle$ is the graph t in D', and $G \subseteq G'$, then $\mathsf{ans}(\boldsymbol{q}_1, D) \subseteq \mathsf{ans}(\boldsymbol{q}_1, D')$. We can then confirm that our semantics is well-defined for positive or stratified positive rec-SPARQL as an easy corollary of Proposition 3.3.1.

PROPOSITION 3.3.3. Let D be a dataset and $\boldsymbol{q}$ a stratified positive (or just positive) recursive c-query of the form WITH RECURSIVE t AS $\{\boldsymbol{q}_1\}$ $\boldsymbol{q}_2$. Then $\boldsymbol{q}$ converges over D.

3.3.4. Complexity Analysis

Since recursive queries can use either the SELECT or the CONSTRUCT result form, there are two decision problems we need to analyze. For SELECT queries, we define the problem SELECTQUERYANSWERING, that receives as an input a recursive query $\boldsymbol{q}$ using the SELECT result form, a tuple $\bar{a}$ of IRIs from $\mathbf{I}$ and a dataset D with default graph G_0, and asks whether $\bar{a}$ is in $\mathsf{ans}(\boldsymbol{q}, D)$. For CONSTRUCT queries, the problem CONSTRUCTQUERYANSWERING receives a recursive query $\boldsymbol{q}$ using the CONSTRUCT result form, a triple (s, p, o) over $\mathbf{I} \times \mathbf{I} \times \mathbf{I}$ and a dataset D, and asks whether this triple is in $\mathsf{ans}(\boldsymbol{q}, D)$. We also clarify that for this analysis we only consider positive SPARQL for the recursive CONSTRUCT queries.

PROPOSITION 3.3.4. The problem

SELECTQUERYANSWERING

is PSPACE-complete and

CONSTRUCTQUERYANSWERING

is NP-complete. The complexity of SELECTQUERYANSWERING drops to Π_2^p if one only consider SELECT queries given by unions of well-designed graph patterns[4].

PROOF. It was proved in (Kostylev, Reutter, & Ugarte, 2015) that the problem CONSTRUCTQUERYANSWERING is NP-complete for non recursive c-queries, and Pérez et al. show in (Pérez, Arenas, & Gutierrez, 2009) that the problem SELECTQUERYANSWERING if PSPACE-complete for non-recursive SPARQL queries, and Π_2^p for non-recursive SPARQL queries given by unions of well-designed graph patterns. This immediately gives us hardness for all three problems when recursion is allowed.

[4]Note that this applies only for the possible last SELECT query at the end of a recursive query.

To see that the upper bound is maintained, note that for each nested query, the temporal graph can have at most $|D|^3$ triples. Since we are computing the least fixed point, this means that in every iteration we add at least one triple, and thus the number of iterations is polynomial. This in turn implies that the answer can be found by composing a polynomial number of NP problems, to construct the temporal graph corresponding to the fixed point, followed by the problem of answering the outer query over this fixed point database, which is in PSPACE for SELECTQUERYANSWERING, in Π_2^p for SELECTQUERYANSWERING assuming queries given by unions of well designed patterns and in NP for CONSTRUCTQUERYANSWERING. The first two classes are closed under composition with NP, and the last NP bound can be obtained by just guessing all meaningful queries, triples to be added and witnesses for the outer query at the same time. □

Thus, at least from the point of view of computational complexity, our class of recursive queries are not more complex than standard select queries (Pérez et al., 2009) or construct queries (Kostylev, Reutter, & Ugarte, 2015). We also note that the complexity of similar recursive queries in most data models is typically complete for exponential time; what lowers our complexity is the fact that our temporary graphs are RDF graphs themselves, instead of arbitrary sets of mappings or relations.

For databases it is also common to study the data complexity of the query answering problem, that is, the same decision problems as above but considering the input query to be fixed. We denote this problems as SELECTQUERYANSWERING($\boldsymbol{q}$) and CONSTRUCTQUERYANSWERING($\boldsymbol{q}$), for select and result queries, respectively. The following shows that the problem remains in polynomial time for data complexity, albeit in a higher class than for non recursive queries.

PROPOSITION 3.3.5. The problems:

- SELECTQUERYANSWERING($\boldsymbol{q}$) and
- CONSTRUCTQUERYANSWERING($\boldsymbol{q}$)

are PTIME-complete.

PROOF. Following the same idea as in the proof of Proposition 3.3.4, we see that the number of iterations needed to construct the fixed point database is polynomial. But, if queries are fixed, the problem of evaluating SELECT and CONSTRUCT queries is always in NLOGSPACE (see again (Pérez et al., 2009) and (Kostylev, Reutter, & Ugarte, 2015)). The PTIME upper bound then follows by composing a polynomial number of NLOGSPACE algorithms.

We prove the lower bound by a reduction from the *path systems* problem, which is a well known PTIME-complete problem (c.f. (Vardi, 1995)). The problem is as follows. Consider a a set of nodes V and a unary relation $C(x) \subseteq V$ that indicates whether a node is *coloured* or not. Let $R(x, y, z) \subseteq V \times V \times V$ be a relation of reachable elements, and the following rule for colouring additional elements: if there are coloured elements a, b such that a triple (a, b, c) is coloured, then c should also be coloured. Finally consider a *target* relation $T \subseteq V$. The problem of *path systems* is to decide if some element in T is coloured by our rule.

For our reduction we construct a database instance and a (fixed) recursive query according to the instance of *path systems* such that the result of the query is not empty if and only if some element of T is coloured by our rule in the *path system* problem. The construction is as follows.

The database instance contains the information of which vertex is coloured, which vertex is part of the target relation T and the elements of the R relation:

- We define the function u which maps every vertex to a unique URI.
- For each element $v \in C$, we add the triple $(u(v), \texttt{:p}, \texttt{"C"})$ to a named graph `gr:C` of the database instance.
- For each element $v \in T$, we add the triple $(u(v), \texttt{:p}, \texttt{"T"})$ to a named graph `gr:T` of the database instance.
- For each element $(x, y, z) \in R$ we add the triple $(u(x), u(y), u(z))$ to the default graph of the database instance.

Thus, the recursive query needs to compute all the coloured elements in order to check if the target relation is covered. This can be done in the following way:

```
1 PREFIX gr: <http://example.org/graph>
```

```
WITH RECURSIVE http://db.puc.cl/temp
AS {
  CONSTRUCT { ?z :p "C" }
  WHERE {
    { GRAPH gr:C { ?z :p "C" } }
    UNION {
      { ?x ?y ?z } .
      GRAPH <http://db.puc.cl/temp> {
        ?x :p "C" } .
      GRAPH <http://db.puc.cl/temp> {
        ?y :p "C" }
      }
    }
  }
}
ASK WHERE {
  GRAPH gr:T {
    ?x :p "T"
  } .
  GRAPH <http://db.puc.cl/temp> {
    ?x :p "C"
  }
}
```

It is clear that the recursive part of the query is computing all the coloured nodes according to the R relation. Then in the ASK query[5], its result will be false iff none of the nodes in T are reachable. Note that this reduction can be immediately adapted to reflect hardness for queries using CONSTRUCT or SELECT. □

From a practical point of view, and even if theoretically the problems have the same combined complexity as queries without recursion and are polynomial in data complexity, any implementation of the Algorithm 1 is likely to run excessively slow due to a high demand on computational resources (computing the temporary graph

[5]Note that an ASK query is a boolean query with no variables to be projected.

over and over again) and would thus not be useful in practice. For this reason, instead of implementing full-fledged recursion, we decided to support a fragment of recursive queries based on what is commonly known as *linear recursive queries* (Green et al., 2013; Abiteboul et al., 1995). This restriction is common when implementing recursive operators in other database languages, most notably in SQL (PostgreSQL, n.d.), but also in graph databases (Consens & Mendelzon, 1990), as it offers a wider option of evaluation algorithms while maintaining the ability of expressing almost any recursive query that one could come up with in practice. For instance, as demonstrated in the following section, linear recursion captures all the examples we have considered thus far and it can also define any query that uses property paths. Furthermore, it can be implemented in an efficient way on top of any existing SPARQL engine using a simple and easy to understand algorithm. All of this is defined in the following section.

3.4. Realistic Recursion in SPARQL

Having defined our recursive language, the next step is to outline a strategy for implementing it inside of a SPARQL system. In this section we show how this can be done by focusing on *linear* queries. While the use of linear queries is well established in the SQL context, here we show how this approach can be lifted to SPARQL. In doing so, we will argue that not only do linear queries allow for much faster evaluation algorithms than generic recursive queries, but they also contain many queries of practical interest. Additionally, we outline some alternatives of recursion which can support the use of negation or BIND operators.

3.4.1. Linear recursive queries

The concept of *linear recursion* is widely used as a restriction for fixed point operators in relational query languages, because it presents a good trade-off between the expressive power of recursive operators and their practical applicability.

Commonly defined for logic programs, the idea of linear queries is that each recursive construct can refer to the recursive graph or predicate being constructed only once. To achieve this, our queries are made from the union of a graph pattern that does

not use the temporary IRI, denoted as $\boldsymbol{p}_{\text{base}}$ and a graph pattern $\boldsymbol{p}_{\text{rec}}$ that does mention the temporary IRI. Formally, a *linear recursive query* is an expression of the form

$$\begin{aligned} &\mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{ \\ &\quad \mathsf{CONSTRUCT}\ H \\ &\quad \mathsf{WHERE}\ \boldsymbol{p}_{\text{base}}\ \mathsf{UNION}\ \boldsymbol{p}_{\text{rec}}\ \}\ \boldsymbol{q}_{\text{out}} \end{aligned} \tag{3.3}$$

with H is a construct template as usual, $\boldsymbol{q}_{\text{out}}$ a linear recursive query, $\boldsymbol{p}_{\text{base}}$ and $\boldsymbol{p}_{\text{rec}}$ positive SPARQL queries, and where only $\boldsymbol{p}_{\text{rec}}$ is allowed to mention the IRI t. We further require that the recursive part $\boldsymbol{p}_{\text{rec}}$ mentions the temporary IRI at most once, and the graph clause contains only a singe triple pattern. Consequently, we define linear positive rec-SPARQL and linear stratified positive rec-SPARQL by restricting the operators in $\boldsymbol{p}_{\text{base}}$ and $\boldsymbol{p}_{\text{rec}}$. The semantics of linear positive and stratified positive recursive queries is inherited from Definition 3.3.2.

Notice that we enforce a syntactic separation between base and recursive query. This is done so that we can keep track of changes made in the temporary graph without the need for computing the difference of two graphs, as discussed in Subsection 3.4.2. This simple yet powerful syntax resembles the design choices taken in most SQL commercial systems supporting recursion[6], and is also present in graph databases (Consens & Mendelzon, 1990). To give an example of a linear query, we take a look to the query from Figure 3.4 that uses one level of nesting (meaning that the query $\boldsymbol{q}_{\text{out}}$ is again a linear recursive query).

We note that the union in the first query can obviously be omitted, and is there only for clarity (our implementation supports queries where either $\boldsymbol{p}_{\text{base}}$ or $\boldsymbol{p}_{\text{rec}}$ is empty). The idea of this query is to first dump all meaningful triples from the original dataset into a new graph named `http://db.ing.puc.cl/temp1`, and then use

[6]In SQL one cannot execute a recursive query which is not divided by UNION into a base query (inner query) and the recursive step (outer query) (PostgreSQL, n.d.).

```
1  PREFIX prov: <http://www.w3.org/ns/prov#>
2  WITH RECURSIVE http://db.ing.puc.cl/temp1 AS {
3      CONSTRUCT { ?x ?u ?y }
4      WHERE{
5          { ?x prov:wasRevisionOf ?z .
6            ?x prov:wasGeneratedBy ?w .
7            ?w prov:used ?z .
8            ?w prov:wasAssociatedWith ?u }
9      UNION
10         {}}
11 }
12 WITH RECURSIVE http://db.ing.puc.cl/temp2 AS {
13     CONSTRUCT { ?x ?u ?y }
14     WHERE
15         { GRAPH <http://db.ing.puc.cl/temp1> { ?x ?u ?y } }
16     UNION {
17         GRAPH <http://db.ing.puc.cl/temp1> { ?x ?u ?z }.
18         GRAPH <http://db.ing.puc.cl/temp2> { ?z ?u ?y } }
19 }
20 SELECT ?x ?y WHERE {
21   GRAPH <http://db.ing.puc.cl/temp> {
22     ?x ?u ?y
23   }
24 }
```

FIGURE 3.4. Example of a linear recursion.

this graph as a basis for computing the required reachability condition, that will be dumped into a second temporary graph `http://db.ing.puc.cl/temp2`[7].

3.4.2. Algorithm for linear recursive queries

The main reason why linear queries are widely used in practice is the fact that they can be computed piece by piece, without ever invoking the complete database being constructed. More precisely, if a query $Q = \mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2$ is linear, then for every dataset D, the answer $\mathsf{ans}(Q, D)$ of the query can be computed as the least fixed point of the sequence given by

$$D_0 = D, \quad D_{-1} = \emptyset,$$

$$D_{i+1} = D_i \cup \{\langle t, \mathsf{ans}(\boldsymbol{q}_1, (D \cup D_i \setminus D_{i-1}))\rangle\}.$$

[7]Interestingly, one can show that in this case the nesting in this query can be avoided.

In other words, in order to compute the $i+1$-th iteration of the recursion, we only need the original dataset plus the tuples that were added to the temporary graph t in the i-th iteration. Considering that the temporary graph t might be of size comparable to the original dataset, linear queries save us from evaluating the query several times over an ever increasing dataset: instead we only need to take into account what was added in the previous iteration, which is generally much smaller.

Unfortunately, it is undecidable to check whether a given recursive query satisfies the property outlined above (under usual complexity-theoretic assumptions, see (Gaifman, Mairson, Sagiv, & Vardi, 1993)), so this is why we must guarantee it with syntactic restrictions. We also note that most of the recursive extensions proposed for SPARQL have the aforementioned property: from property paths (Harris & Seaborne, 2013) to nSPARQL (Pérez et al., 2010), SPARQLeR (Kochut & Janik, 2007), regular queries (Reutter, Romero, & Vardi, 2017) or Trial (Libkin et al., 2018), as well as our example.

As for the algorithm, we have decided to implement what is known as *seminaive evaluation*, although several other alternatives have been proposed for the evaluation of these types of queries (see (Green et al., 2013) for a good survey). In order to describe our algorithm for evaluating a query of the shape (3.3), we abuse the notation and speak of $\boldsymbol{q}_{\text{base}}$ to denote the query $\mathsf{CONSTRUCT}\ H\ \ \mathsf{WHERE}\ \boldsymbol{p}_{\text{base}}$ and $\boldsymbol{q}_{\text{rec}}$ to denote the query $\mathsf{CONSTRUCT}\ H\ \ \mathsf{WHERE}\ \boldsymbol{p}_{\text{rec}}$. Our algorithm for query evaluation is presented in Algorithm 2.

So what have we gained? By looking at Algorithm 2 one realizes that in each iteration we only evaluate the query over the union of the dataset and the intermediate graph G_{temp}, instead of the previous algorithm where one needed the whole graph being constructed (in this case G_{ans}). Furthermore, $\boldsymbol{q}_{\text{base}}$ is evaluated only once, using $\boldsymbol{q}_{\text{rec}}$ in the rest of the iterations. Considering that the temporary graph may be large, and that no indexing scheme could be available, this often results in a considerable speedup for query computation.

Algorithm 2 Computing the answer for linear recursive c-queries of the form (3.3).

Input: Query Q of the form (3.3), dataset D
Output: Evaluation $\mathsf{ans}(Q, D)$ of Q over D

1: $G_{temp} \leftarrow \mathsf{ans}(\boldsymbol{q}_{\text{base}}, D)$
2: $G_{ans} \leftarrow G_{temp}$
3: $size = |G_{ans}|$
4: **loop**
5: $\quad G_{temp} \leftarrow \mathsf{ans}(\boldsymbol{q}_{\text{rec}}, D \cup \{(t, G_{temp})\})$
6: $\quad G_{ans} \leftarrow G_{ans} \cup G_{temp}$
7: $\quad$ **if** $size = |G_{ans}|$ **then**
8: $\quad\quad$ **break**
9: $\quad$ **else**
10: $\quad\quad size \leftarrow |G_{ans}|$
11: $\quad$ **end if**
12: **end loop**
13: **return** $\mathsf{ans}(\boldsymbol{q}_{\text{out}}, D \cup \{\langle t, G_{ans}\rangle\})$

3.4.3. Limiting the recursion depth

Although we show in Section 3.3 that recursive queries which include some form of negation can be impossible to evaluate, there is no doubt that queries including negation are very useful in practice. In this section we briefly discuss how such queries can be mixed with linear recursion.

Limiting the recursion depth. In practice it could happen that an user may not be interested in having all the answers for a recursive query. Instead, the user could prefer to have only the answers until a certain number of iterations are performed. We propose the following syntax for to restrict the depth of recursion to a user specified number k:

$$\begin{aligned} &\mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{ \\ &\quad \mathsf{CONSTRUCT}\ H \\ &\quad \mathsf{WHERE}\ \boldsymbol{p}_{\text{base}}\ \mathsf{UNION}\ \boldsymbol{p}_{\text{rec}} \\ &\quad \}\ \mathsf{MAXRECURSION}\ k\ \boldsymbol{q}_{\text{out}} \end{aligned} \tag{3.4}$$

Here all the keywords are the same as when defining linear recursion, and $k \geq 1$ is a natural number. The semantics of such queries is defined using Algorithm 2, where the loop between steps 4 and 12 is executed precisely $k - 1$ times.

It is easy to see that this extension is useful for handling queries which include negation, or which create values by means of blanks or a BIND clause. Namely, if we fix the number of iterations of a recursive query, we can ensure that these queries terminate their execution, regardless of the existence of a fixed point.

Other ways of supporting negation. Limiting the number of iterations can also give us a way of allowing more complex c-queries inside the recursive part of recursive queries. This is not an elegant solution, but can be made to work: since the number of iterations is bounded, we do not longer need restrictions over our queries to ensure that our graph has a fixed point.

We also mention that there are other, more elegant solutions, but we do not investigate them further as they drive us out of what can be implemented on top of SPARQL systems, and is out of the scope of this paper. For example, a possible solution to support BIND and negation is to extend the semantics by borrowing the notion of stable models from logic programs (see e.g. (Niemelä, 1999)). Moreover, one could redefine rec-SPARQL to consider a *partial fixed point* in Definition 3.3.2 instead of the least-fixed point. This approach simply assumes a query that does not converge gives an empty result.

Studying these extensions to rec-SPARQL is an important topic for future work, and in particular the stable model semantics approach may require an interesting combination of techniques from both databases and logic programming.

3.5. Experimental Evaluation

In this section we will discuss how our implementation performs in practice and how it compares to alternative approaches that are supported by existing RDF Systems. Though our implementation has more expressive power, we will see that the response time of our approach is similar to the response time of existing approaches, and also our implementation outperforms the existing solutions in several use cases.

Technical details. Our implementation of linear recursive queries was carried out using the Apache Jena framework (version 3.7.0)(Jena, 2015) as an add-on to the ARQ SPARQL query engine. It allows the user to run queries either in main memory, or using disk storage when needed. The disk storage was managed by Jena TDB (version 1). As previously mentioned, since the query evaluation algorithms we develop make use of the same operations that already exist in current SPARQL engines, we can use those as a basis for the recursive extension to SPARQL we propose. In fact, as we show by implementing recursion on top of Jena, this capability can be added to an existing engine in an elegant and non-intrusive way. To check the source code and the supplementary material for Recursive SPARQL see Appendix A.

Datasets. We test our implementation using four different datasets. The first one is Linked Movie Database (LMDB) (LMDB, n.d.), an RDF dataset containing information about movies and actors. The second dataset we use is a part of the YAGO ontology (YAGO, n.d.) and consists of all the facts that hold between instances. For the experiments the version from May 2018 was used. In order to test the performance of our implementation on synthetic data, we turn to the GMark benchmark (Bagan et al., 2017), and generate data with different characteristics using this tool. Finally, we use the Wikidata "truthy" dump from 2018/11/15 containing over 3 billion triples, in order to test whether our implementation scales. All the datasets apart from Wikidata can be found at `https://alanezz.github.io/RecSPARQL/`.

Experiments. The experiments we run are divided into four batches:

- **Common use cases.** In the first round of experiments we turn to YAGO and LMDB datasets, which allow for defining recursive queries rather naturally. The main objective of these experiments is to show that our implementation can handle complex recursive patterns in reasonable time over real world datasets.
- **Comparison with SPARQL engines.** In order to compare with the recursive properties supported by SPARQL, we turn to the GMark (Bagan et

al., 2017) property path benchmark, and compare our implementation with pache Jena and Openlink Virtuoso, two popular SPARQL systems.

- **Performance over large datasets.** To verify whether our solution scales, we run a sequence of recursive queries over the Wikidata dataset containing over 3 billon triples, and compare our response times with the ones provided by the Wikidata endpoint.
- **Limiting recursion depth.** Finally, we test the solution proposed in Section 3.4.3, which stops the recursive iteration after a predetermined number of steps. Here we show that this approach is not only useful fro dealing with recursion, but also when evaluating repeated joins.

The experiments involving smaller datasets (LMDB, YAGO, and GMark) were run on a MacBook Pro with an Intel Core i5 2.6 GHz processor and 8GB of main memory. To handle the size of Wikidata, we used a server Devuan GNU/Linux 3 (beowulf) with an Intel Xeon Silver 4110 CPU @ 2.10GHz processor and 120GB of memory.

Next, we elaborate on each batch of experiments, as specified above.

3.5.1. Evaluating real use cases

The first thing we do is to test our implementation against realistic use cases. As we have mentioned, we do not aim to obtain the fastest possible algorithms for these particular use cases (this is out of the scope of this paper), but rather aim for an implementation whose execution times are reasonable. For this, we took the LMDB and the YAGO datasets, and built a series of queries asking for relationships between entities. Since YAGO also contains information about movies, we have the advantage of being able to test the same queries over different datasets (their ontology differs). The specifications for each database can be found in the Figure 3.5. Note that the size is the one used by Jena TDB to store the datasets.

Graph	Number of triples	Size
LMDB	6147996	1.09 Gb
Yago	6215350	1.54 Gb

FIGURE 3.5. Specifications for the LMDB and Yago datasets.

To the best of our knowledge, it is not possible to compare the full scope of our approach against other implementations. While it is true that our formalism is similar to the recursive part of SQL, all of the RDF systems that we checked were either running RDF natively, or running on top of a relational DBMS that did not support the recursion with common table expressions functionality, which is part of the SQL standard. OpenLink Virtuoso does have a *transitive closure* operator that can be used with its SQL engine, but this operator is quite limited in the sense that it can only compute transitivity when starting at a given IRI. Our queries were more general than this, and thus we could not compare them directly. For this reason, in this set of experiments we will only discuss about the practical applicability of the results.

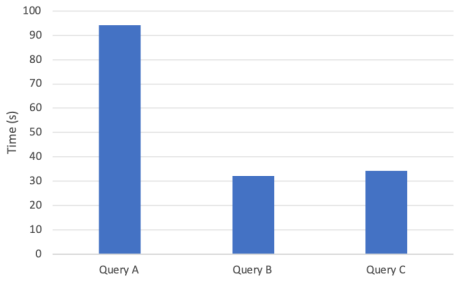
Our round of experiments consists of three movie-related queries, which will be executed both on LMDB and YAGO, and two additional queries that are only run in YAGO, because LMDB does not contain this information. All of these queries are similar to that of Example 3.1.1. The queries executed in both datasets are the following:

- QA: the first query returns all the actors in the database that have a finite Bacon number[8], meaning that they co-starred in the same movie with Kevin Bacon, or another actor with a finite Bacon number. A similar notion, well known in mathematics, is that of an Erdős number.
- QB: the second query returns all actors with a finite Bacon number such that all the collaborations were done in movies with the same director.
- QC: the third query tests if an actor is connected to Kevin Bacon through movies where the director is also an actor (not necessarily in the same movie).

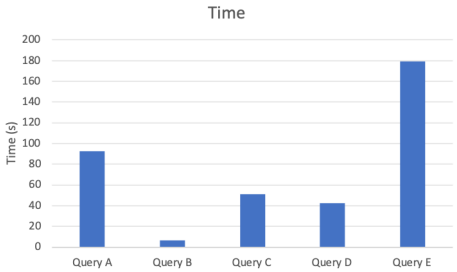
The queries executed only in the YAGO dataset where the following:

- QD: the fourth query answers with the places where the city Berlin is located in from a transitive point of view, starting from Germany, then Europe and so forth.

[8] `http://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon`.



(a) Query times on LMDB dataset



(b) Query times on YAGO dataset

QA	QB	QC
37349	1172	14568

(c) The number of output tuples for LMDB queries

QA	QB	QC	QD	QE
29930	85	3617	7	44

(d) The number of output tuples for Yago queries

FIGURE 3.6. Running times and the number of output tuples for the three datasets.

- QE: the fifth query returns all the people who are transitively related to someone, through the `isMarriedTo` relation, living in the United States or some place located within the United States.

Note that QA, QD and QE are also expressible as property paths. To fully test recursive capabilities of our implementation we use another two queries, QB and QC, that apply various tests along the paths computing the Bacon number. Recall that the structure of queries QB and QC is similar to the query from Example 3.1.1 and cannot be expressed in SPARQL 1.1 either.

The results of the evaluation can be found in Figures 3.6(a) and 3.6(b). As we can see the running times, although high, are reasonable considering the size of the datasets and the number of output tuples (Figures 3.6(c) and 3.6(d)). The query QE is the only query with a small size in its output and a high time of execution. This fact can be explained because the query is a combination of 2 property paths that required to instantiate 2 recursive graphs before computing the answer. Also, as a reference, we compared our engine with respect to the implementation of the WITH RECURSIVE operator of PostgresSQL. We remark that the performance may vary depending on the serialization of the RDF graph in a relational model. The results can be found at the Appendix B.

3.5.2. Comparison with Property Paths using the GMark benchmark

As mentioned previously, since to the best of our knowledge no SPARQL engine implements general recursive queries, we cannot really compare the performance of our implementation with the existing systems. The only form of recursion mandated by the latest language standard are property paths, so in this section we show the results of comparing the execution of property paths queries in our implementation using our recursive language against the implementation of property paths in popular systems.

We used the GMark benchmark (Bagan et al., 2017) to measure the running time of property paths queries using Recursive SPARQL, and to compare such times with respect to Apache Jena and Openlink Virtuoso.

The GMark benchmark allows generating queries and datasets to test property paths, and one of its advantages is that the size of the datasets and the patterns described by the queries are parametrized by the user. Using the benchmark we generated three different graphs of increasing size, named G_1, G_2 and G_3. The specifications for each graph can be found in Figure 3.7. We also generated 10 SPARQL queries that could have one or more property paths of different complexities. The run times for our queries are presented in Figure 3.8 for the graph G_1, in Figure 3.9 for G_2, and in Figure 3.10 for G_3.

Graph	Number of triples	Size
Graph 1	220564	271 mb
Graph 2	447851	535 mb
Graph 3	671712	605 mb

FIGURE 3.7. Specifications for the graphs generated by GMark.

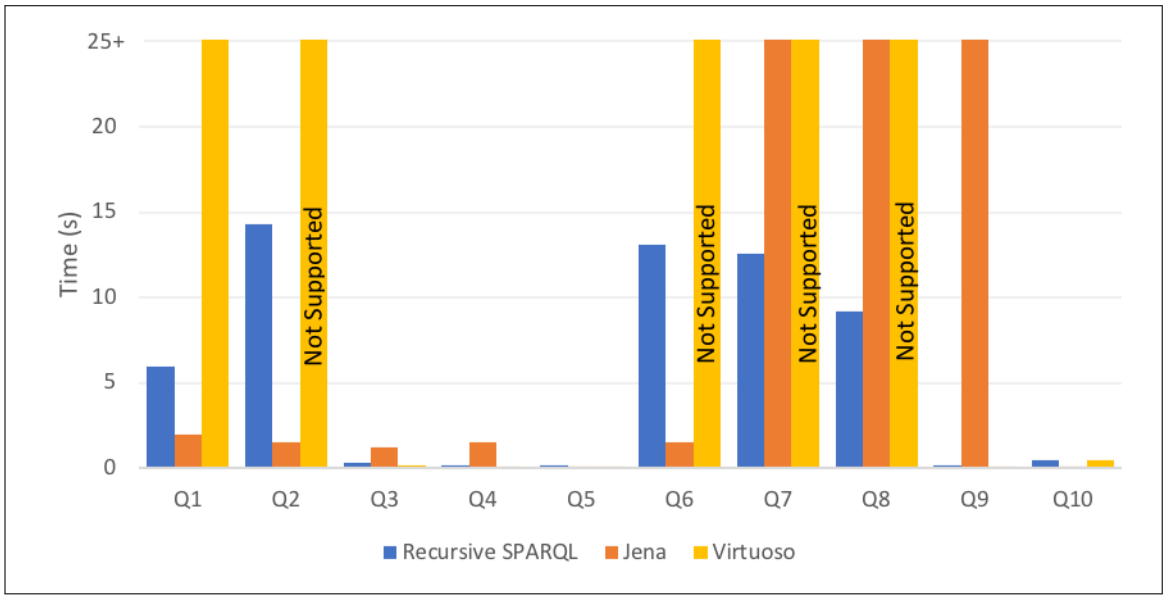


FIGURE 3.8. Times for $G1$.

Note first that every property path query is easily expressible using linear recursion. With this observation in mind we must also remark that we are comparing the performance of our more general recursive engine with property paths, which are a much less expressive language. For this reason highly efficient systems like Virtuoso should run property paths queries faster: they do not need to worry about being able to compute more recursive queries. Of course, it would also be interesting to compare our engine with specific ad-hoc techniques for computing property paths.

Comparison with Virtuoso. Virtuoso cannot run queries 2, 6, 7 and 8, because the SPARQL engine requires an starting point for property paths queries, which was not possible to give for such queries. We can see that Virtuoso outperforms Jena and the Recursive implementation in almost all the queries that they can run, except for Query 1, where the running time goes beyond 25 seconds. As we will discuss later, this can

be explained because of the semantic they use to evaluate property paths, which makes Virtuoso to have many duplicated answers. For the remaining queries, we can see that the execution time is almost equals.

Comparison with Jena. Apache Jena can also answer all queries. However, our recursive implementation is only clearly outperformed in Query 2 and Query 6. This is mainly because those queries have patterns of the form:

```
?x (:p1|:p2)* ?z
```

and our system is not optimized for working with unions of predicates. Remarkably, and even though all of the generated queries are relatively simple, our implementation reports a faster running time in half of the queries we test. Note that Q7, Q8 and Q9 have an answer time considerably worse in Jena than in our recursive implementation, where the time goes beyond the 25 seconds. We can only speculate that this is because the property paths has many paths of short length and because Apache Jena cannot manage properly the queries with two or more star triple patterns.

When we increase the size of the graph, the results have the same behaviour. It is also more evident which queries are easier and harder to evaluate for the existing systems. The result for the increased size of the graph can be found in the Figures 3.9 and 3.10.

Number of outputs. As we said before, one interesting thing that we note from the previous experiments is the time that Virtuoso took to answer the query Q1 in the three dataset. This occur because Virtuoso generates many duplicate results, thus the output should be higher with respect to rec-SPARQL and Jena. We count the number of outputs for the queries ran over the first graph. The results can be seen in Figure 3.11.

The first thing to note is that we avoid duplicate answers in our language, mainly because of the `UNION` operator used in linear recursion, which deletes the duplicates

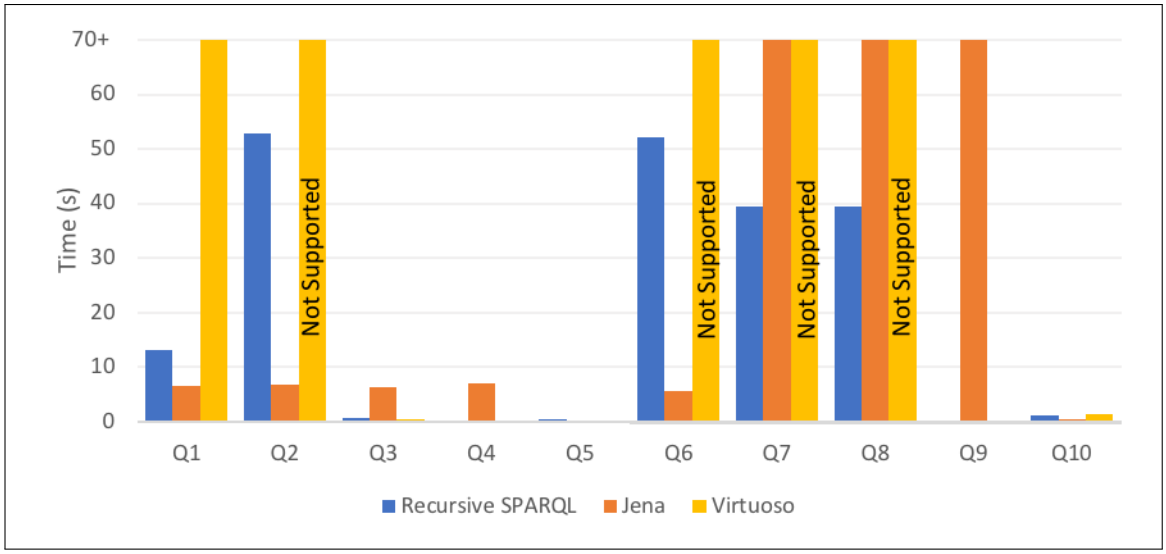


FIGURE 3.9. Times for $G2$.

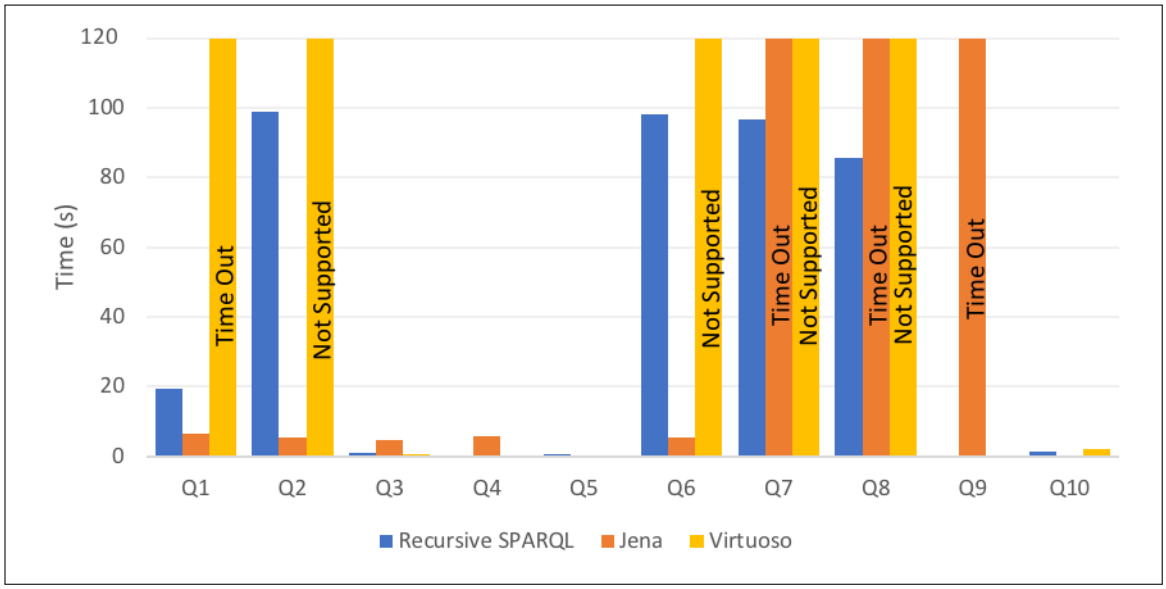


FIGURE 3.10. Times for $G3$.

answers. Also we do not consider paths of length 0, because those results do not give us any relevant information about the answer. Then we can see that Apache Jena has always more results than us, this is mainly because they consider paths of length 0. We did not rewrite the queries because we wanted keep them as close as possible to the benchmark. In Virtuoso one needs a starting point for property path queries, so this system does not consider paths of length 0 and for that reason in some queries they have fewer outputs than Apache Jena. However, in most of the queries they give

Query	RecSPARQL	Jena	Virtuoso
Q1	24723	103814	1849915
Q2	3964	90398	-
Q3	1455	89128	2267
Q4	9	198	198
Q5	169	802	804
Q6	3964	90398	-
Q7	2604	10838	-
Q8	126	94638	-
Q9	2	89523	454
Q10	906	5373	6345

FIGURE 3.11. Number of resulsts for the GMark queries over the first graph.

more results than Apache Jena and RecSPARQL, because they produce many duplicate answers and thus, the answer time becomes considerably worse. This happens mainly in the first query, which is the simplest one. The same effect happens for the 2 bigger graphs. The number of outputs for the bigger graphs can be found in Figures B.1 and B.2.

Query	RecSPARQL	Jena	Virtuoso
Q1	50190	208437	3624482
Q2	8153	181043	-
Q3	3188	178465	5256
Q4	7	409	409
Q5	345	1547	1561
Q6	8153	181043	-
Q7	6116	16730	-
Q8	308	189628	-
Q9	4	179168	968
Q10	2134	11022	13002

FIGURE 3.12. Number of outputs for the GMark queries over the second graph.

3.5.3. Tests over Wikidata

We want to know how our recursive operator works when the queries are executed over large datasets. Thus, we try our implementation with queries over the graph of Wikidata. This dataset is the biggest knowledge-base maintained by the Wikimedia

Query	RecSPARQL	Jena	Virtuoso
Q1	74967	311559	-
Q2	12015	270506	-
Q3	4719	266777	7743
Q4	17	766	766
Q5	533	2674	2716
Q6	12015	270506	-
Q7	16040	-	-
Q8	487	-	-
Q9	4	-	1384
Q10	2942	15951	18726

FIGURE 3.13. Number of outputs for the GMark queries over the third graph.

foundation and edited by a community of thousands of users (Vrandečić & Krötzsch, 2014). The goal of Wikidata is to provide a common interoperable source of factual information represented as RDF data. We load the "truthy" dump from 2018/11/15. This dump contains 3,303,288,386 triples. For this set of experiments we use a server Devuan GNU/Linux 3 (beowulf) with an Intel Xeon Silver 4110 CPU @ 2.10GHz processor and 120GB of memory.

We create property paths queries based on (1) the example queries showed at the Wikidata Endpoint and (2) the LMDB queries from Subsection 3.5.1. The queries are the following:

- **Q1**: Sub-properties of property `P276`.
- **Q2**: All the instances of *horse* or a subclass of *horse*.
- **Q3**: Parent taxon of the Blue Whale.
- **Q4**: Metro stations reachable from Palermo Station in *Metro de Buenos Aires*.
- **Q5**: Actors with finite Bacon number:

Queries **Q1**, and **Q4** are simple star $\star$ queries, while **Q3** is a star query where in each iteration only one triple is added to the recursive graph. **Q2** is a query of the form `(wdt:p1/wdt:p2*)`, while **Q5** combines two properties within a star.

As hinted by the previous experiments, we rewrite the property paths as `WITH RECURSIVE` queries and we run them on our server setup. We display the results in

Figure 3.14. As a reference, we also put the time that the queries took at the Wikidata endpoint `https://query.wikidata.org/`. We note that this is not an exact comparison as the endpoint dataset might slightly differ from the one we use, and the server running the endpoint is likely different from ours. The values are expressed in seconds.

Query	RecSPARQL	Endpoint
Q1	2.23	0.28
Q2	2.45	1.02
Q3	2.15	0.56
Q4	2.11	0.73
Q5	101.60	Timeout

FIGURE 3.14. Time in seconds taken by the queries over the Wikidata Graph.

As we see, the trend shown with the previous experiments is repeated again with a large dataset: Recursive SPARQL is almost as competitive as the existing solutions. Since the dataset of Wikidata is larger than previous datasets and our solution implies to do several joins, we expected easier queries to have better running times in the endpoint than in our implementation. Finally, we remark the result for **Q5**, where our implementation could answer the query in a reasonable time, and the endpoint times out.

3.5.4. Limiting the number of iterations

In section 3.4.3 we presented a way of limiting the depth of the recursion. We argue that this functionality should find good practical uses, because users are often interested in running recursive queries only for a predefined number of iterations. For instance, very long paths between nodes are seldom of interest and in a vast majority of use cases we will be interested in using property paths only up to depth four or five. It is straightforward to see that every query defined using recursion with a predefined number of iterations can be rewritten in SPARQL by explicitly specifying each step of the recursion and joining them using the concatenation operator. The question then is, why is specifying the recursion depth beneficial?

One apparent reason is that it makes queries much easier to write and understand. The second reason we would like to argue for is that, when implemented using Algorithm 2, recursive queries with a predetermined number of steps result in faster query evaluation times than evaluating an equivalent query with lots of joins. The intuitive reason behind this is that computing q_{base}, although expensive initially, acts as a sort of index to iterate upon, resulting in fast evaluation times as the number of iterations increases. On the other hand, for even a moderately complex query using lots of joins, the execution plan will seldom be optimal and will often resort to simply trying all the possible matchings to the variables, thus recomputing the same information several times.

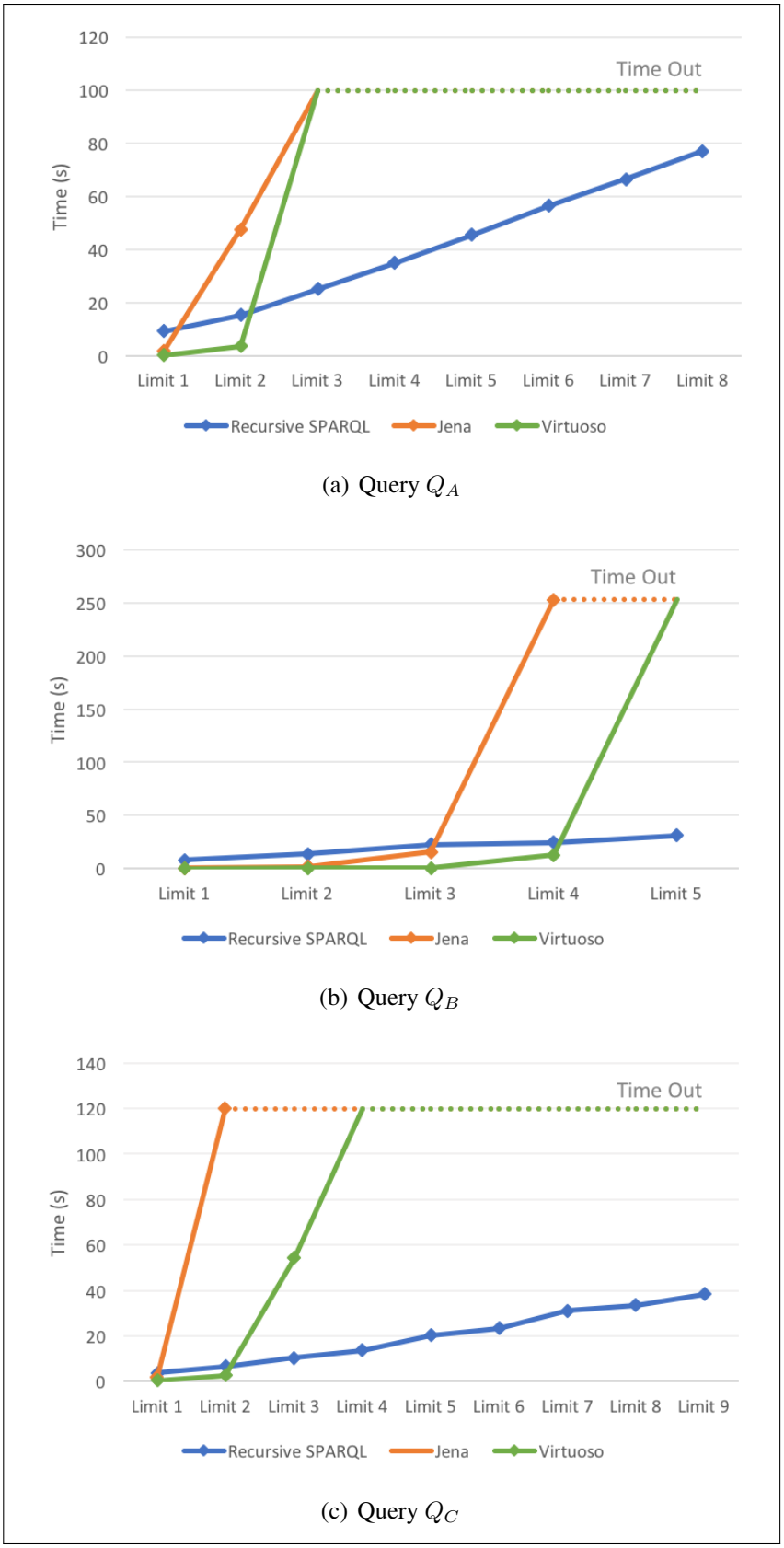
We substantiate this claim by running two rounds of experiments on LMDB and YAGO datasets, using queries QA, QB and QC from Subsection 3.5.1 and running them for an increasing number of steps. We evaluate each of the queries using Algorithm 2 and run it for a fixed number of steps until the algorithm saturates. Then we use a SPARQL rewriting of a recursive query where the depth of recursion is fixed and evaluate it in Jena and Virtuoso.

Figure 3.15 shows the results over LMDB and Figure 3.16 shows the results over YAGO. The time out here is again set to two minutes. As we can see, the initial cost is much higher if we are using recursive queries, however as the number of steps increases we can see that they show much better performance and in fact, the queries that use only SPARQL operators time out after a small number of iterations. Note that we did not run the second query over the YAGO dataset, because it ends in two iterations, and it would not show any trend. We also did not run queries QD and QE. Query QD was timing out also after two iterations on Jena and Virtuoso, and query QE is composed of two property paths, so there is no straightforward way to transform it in a query with unions.

As illustrated by several use cases, there is a need for recursive functionalities in SPARQL that go beyond the scope of property paths. To tackle this issue we propose a recursive operator to be added to the language and show how it can be implemented efficiently on top of existing SPARQL systems.

Furthermore, we believe that rec-SPARQL may also be used for doing Graph Analytics, since the idea of fix point can be extended for such kind of tasks. Consider the algorithm of PageRank. The idea behind this algorithm is to determine the centrality of nodes in an iterative way: all the nodes start with the same PageRank (this is to assign the same number for all the nodes of a graph) and, based on the previous computation, we change the value of the PageRank for the next iteration. This algorithm stops when the difference of the centrality of the nodes from iteration to iteration is minimal. If we take an RDF graph and we append a new property to all the nodes (that represents the current PageRank of a single node), we could compute the algorithm of PageRank by executing a SPARQL query until a "fixpoint" is reached. However, the "fixpoint" here would be the criteria for PageRank to stop.

In the next chapter we will explore this idea, by presenting a Recursive Language that allows us to compute several Graph Analytics tasks. In addition to recursion, we explore other features that are necessary to enable SPARQL in order to compute Graph Analytics tasks.

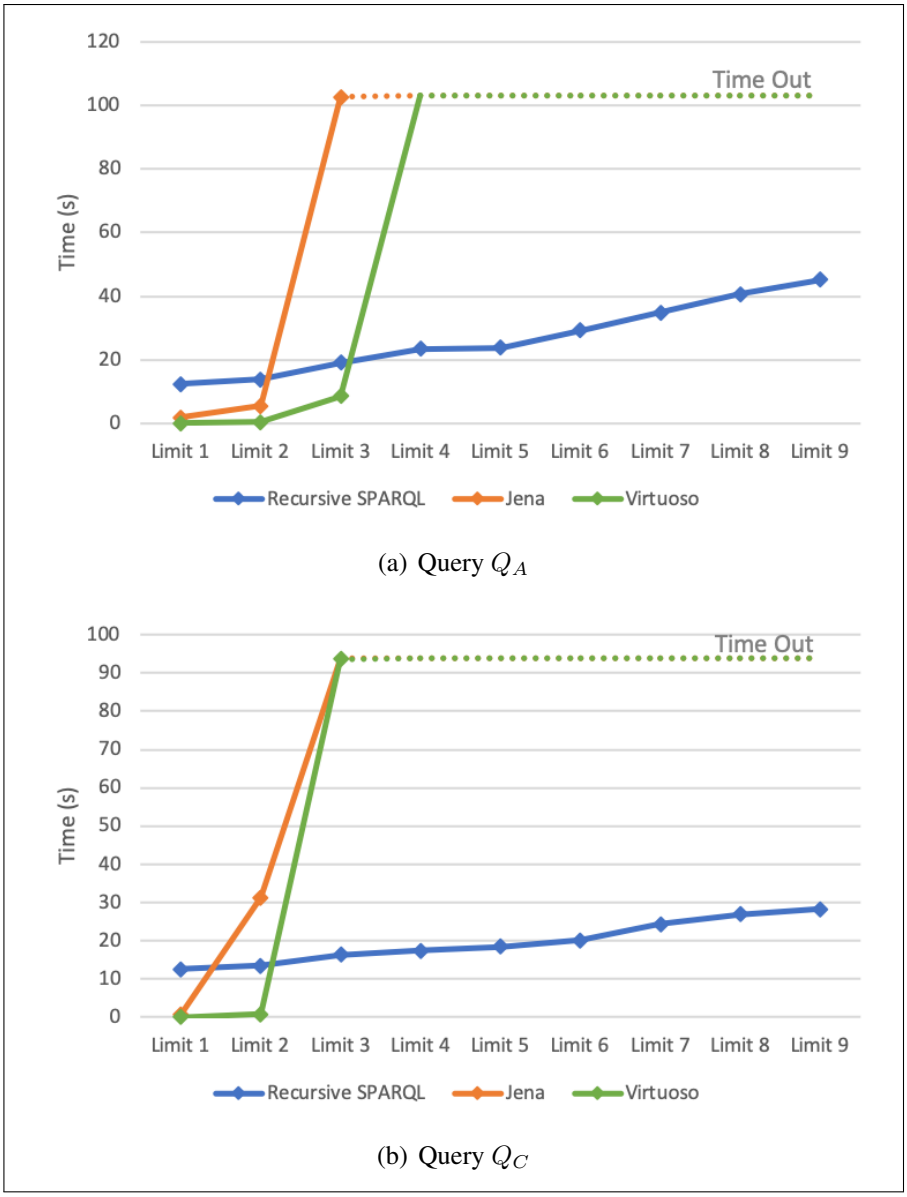


(a) Query Q_A

(b) Query Q_B

(c) Query Q_C

FIGURE 3.15. Limiting the number of iterations for the evaluation of Q_A, Q_B and Q_C over LMDB.



(a) Query Q_A

(b) Query Q_C

FIGURE 3.16. Limiting the number of iterations for the evaluation of Q_A and Q_C over Yago.

Chapter 4. IN-DATABASE GRAPH ANALYTICS WITH RECURSIVE SPARQL

Recent years have seen a surge in interest in graph data management, learning and analytics within different sub-communities, particularly under the title of "knowledge graphs" (Hogan, Blomqvist, et al., 2020). However, more work is needed to combine complementary techniques from different areas (Bonatti et al., 2018). As a prominent example, while numerous query languages have been proposed for graphs (Harris & Seaborne, 2013; Rodriguez, 2015; Francis et al., 2018; Angles et al., 2018, 2017), and numerous frameworks have been proposed for graph analytics (Malewicz et al., 2010; Xin, Gonzalez, et al., 2013; Stutz et al., 2016), few works aim to combine both: while some analytical frameworks support lightweight query features (Xin, Gonzalez, et al., 2013; Rodriguez, 2015), and some query languages support lightweight analytical features (Harris & Seaborne, 2013; Francis et al., 2018), only specific types of queries or analytics are addressed, or imperative "glue-code" is required to combine both.

As we discussed at the end of the previous chapter, one possible application a recursive query language is to use it for computing Graph Analytics tasks. Suppose the following simple task that we wish to compute over the Wikidata graph: *find the top author of scientific articles about the Zika virus according to their p-index within the topic*. The p-index of authors is calculated by computing the PageRank of papers in the citation network, and then summing the scores of the papers for each respective author (Senanayake, Piraveenan, & Zomaya, 2015). One way this could currently be achieved is to: (1) perform a SPARQL query to extract the citation graph of scientific articles about the Zika virus; (2) load the graph into an external tool to compute PageRank scores; (3) perform another query to extract the (bipartite) authorship graph for the articles; (4) load the authorship graph into the external tool to join authors with papers, aggregate the p-index score per author, sort by score, and output the top result. Here the user must ship data back and forth between different tools to solve the task. Another strategy might be to load the Wikidata dump into a graph-analytics framework, writing code to extract the required graphs, analyze them, and aggregate

the results; in this case, we lose the convenience of a declarative query language and database optimizations for extracting (only) the relevant data.

It is important to understand how this approach is useful to create a (mostly) declarative language that supports *graph queralytics*: tasks that combine querying and analytics on graphs, allowing to interleave both arbitrarily. We coin the term "*queralytics*" to highlight that these tasks raise new challenges and are not well-supported by existing languages and tools that focus only on querying or analytics. Rather than extending a graph query language with support for specific, built-in analytics, we rather propose to extend a graph query language to be able to express any form of (computable) analytical task of interest to the user. In addition to adding recursive features to the SPARQL query language, we explore a few other features that are necessary for this approach.

In this chapter we propose a concrete syntax and semantics for a recursive language that allows us to combine querying and analytics for graphs. We call our language the *SPARQL Protocol and RDF Query & Analtyics Language* (*SPARQAL*). We study the expressive power of SPARQAL and we compare it with the language discussed in the previous chapter. We then discuss the implementation of our language on top of a SPARQL query engine, introducing evaluation strategies that aim to find trade-offs between scalability and performance. We present experiments to compare our proposed strategies on real-world datasets, for which we devise a set of benchmark queralytics over Wikidata. Our results provide insights into the scale and performance with which an existing SPARQL engine can perform standard graph analytics, showing that for queralytics wherein a selective sub-graph is extracted for analysis, interactive performance is feasible; on the other hand, the current implementation struggles for larger-scale graphs, opening avenues for future research.

However, a natural question so far could be: why not to use Recursive SPARQL? In comparison with the proposal of Recursive SPARQL and Urzua and Gutiérrez (Urzua & Gutiérrez, 2019), we allow recursion over SELECT queries, which adds flexibility by not requiring to maintain intermediate results as (RDF) graphs: for example, allowing us to maintain multiple intermediate relations of arbitrary arity (without requiring

some form of reification); we further allow for terminating a loop based on a boolean condition (an ASK query), which can more easily express termination conditions in cases where an analytics task is infinitary and/or requires approximation (e.g., PageRank). Recall that Recursive SPARQL is not design to work with negation (such as MINUS or OPTIONAL) and with non-domain preserving values (such as BIND). Finally, as we show in this chapter, the theoretical analysis of SPARQAL gives us the result that this language is in fact more expressive.

4.1. The metro example revisited

First of all, we introduce the language SPARQAL with an example inspired in a query that we have used before, when we introduced Recursive SPARQL.

EXAMPLE 4.1.1. Suppose that there is a concert close to Palermo metro station in Buenos Aires; however, Line C of the metro is closed due to a strike. As mentioned in the introduction, we would like to know from which metro stations we can still reach Palermo. The data to answer this query are available on Wikidata (Vrandečić & Krötzsch, 2014). We can express this request in our SPARQL-based language, as shown in Figure 4.1. Two adjacent stations are given by the property `wdt:P197` and the metro line by `wdt:P81`; the entities `wd:Q3296629` and `wd:Q1157050` refer to Palermo metro station and Line C, respectively. From lines 1 to 7, we first define a *solution variable* called `reachable` whose value is the result of computing all stations directly adjacent to Palermo that are not on Line C. From lines 8 to 22 we have a loop that executes two instructions: the first, starting at line 10, computes all stations directly adjacent to the current reachable stations not on Line C; here the **`QVALUES`**`(reachable)` clause is used to invoke all solutions stored in variable `reachable`. The second, starting at line 17, adds the new adjacent stations to the list of known reachable stations with a union. The loop is finished when the set of solutions assigned to the variable `reachable` does not change from one iteration to another (a fixpoint is thus reached). Finally, on line 23, we return reachable stations. □

As we see, this language is similar to Recursive SPARQL, in the sense that we execute a query until we reach a fix point. However, here we allow recursion over SELECT

```
# stations directly adjacent to Palermo not on Line C
LET reachable = (
  SELECT ?s WHERE {
    wd:Q3296629 wdt:P197 ?s .
    MINUS { ?s wdt:P81 wd:Q1157050 }
  }
);
DO (
  # stations adjacent to stations in variable reachable
  LET adjacent = (
    SELECT (?adj AS ?s) WHERE {
      ?s wdt:P197 ?adj .
      MINUS { ?adj wdt:P81 wd:Q1157050 } QVALUES(reachable)
    }
  );
  # add stations in variable adjacent to variable reachable
  LET reachable = (
    SELECT DISTINCT ?s WHERE {
      { QVALUES(adjacent) } UNION { QVALUES(reachable) }
    }
  );
) UNTIL( FIXPOINT(reachable) );
RETURN(reachable);
```

FIGURE 4.1. Procedure to find metro stations from which Palermo can be reached.

queries, which adds flexibility by not requiring to maintain intermediate results as (RDF) graphs: for example, with SELECT we can maintain a table of four columns/-variables representing a weighted RDF graph, where the first three columns denote an RDF graph and the fourth column denotes weights on individual triples; in the case of CONSTRUCT, we would rather require some form of reification to capture weighted triples. Furthermore, while we support fixpoint recursion, we also support other forms of recursion; in particular, we allow for terminating a loop based on a boolean condition (an ASK query), which offers greater flexibility for defining termination conditions in cases where, for example, an analytics task is infinitary and/or requires approximation in practice (e.g., PageRank). In comparison with LDScript (Corby, Faron-Zucker, & Gandon, 2017) – which also supports recursion on SELECT queries – our focus is rather on supporting graph analytics with such a language, supporting features, such as fixpoint, that are useful in this setting. However, before defining formally our language, we review other approaches related to *queralytics* tasks.

4.2. Existing approaches for Graph Analytics

In terms of related works, we first discuss frameworks and languages for applying graph analytics. We then discuss prior proposals for combining graph querying and graph analytics. We then introduce works on extending graph query languages with recursion. We end by highlighting the novelty of our approach.

Frameworks for Graph Analytics. Given the growing need to perform graph analytics at large-scale – involving the Web, social networks, etc. – various frameworks have been proposed for such settings, including GraphStep (DeLorimier et al., 2006), Pregel (Malewicz et al., 2010), HipG (Krepska et al., 2011), PowerGraph (Gonzalez et al., 2012), GraphX (Xin, Gonzalez, et al., 2013), Giraph (Ching et al., 2015), Signal/Collect (Stutz et al., 2016), and more besides. All such frameworks operate on a computational model – sometimes called the systolic model (Low et al., 2014), Gather/Apply/Scatter (GAS) model (Gonzalez et al., 2012), graph-parallel framework (Xin, Gonzalez, et al., 2013), etc. – that involves each node in a graph recursively computing its state based on data available for its neighbouring nodes according to a given function. Although such frameworks allow for large-scale graph analytics to be applied in a distributed setting, implementing queries on such frameworks, selecting custom sub-graphs to be analysed, etc., is not straightforward. Similar computational models are used in the case of graph neural networks (Scarselli et al., 2009; Wu et al., 2019), which have been shown to be as discriminative as the (incomplete) Weisfeiler–Lehman (WL) graph isomorphism test (Xu et al., 2019): in other words, by basing computation only on local information in each node's neighbourhood, there are certain pairs of non-isomorphic graphs that will return "isomorphic results" for any algorithm implemented in the framework.

Graph Queries and Analytics. Our work aims to combine graph queries and analytics, focusing on RDF graphs. One such proposal along these lines is Trinity.RDF (Zeng et al., 2013), which stores RDF in a native graph format where nodes store inward and outward adjacency lists, allowing to traverse from a node to its neighbours without the need for index lookup; the system is then implemented in a distributed in-memory index, with query processing and optimisation components provided for basic graph

patterns. Although the authors discuss how Trinity.RDF's storage scheme can also be useful for graph algorithms based on random walks, reachability, etc., experiments focus on SPARQL query evaluation from standard benchmarks (Zeng et al., 2013). Later work used the same infrastructure in a system called Trinity (Shao et al., 2013) to implement and perform experiments with respect to PageRank and Breadth-First Search, this time rather focusing on graph analytics without performing queries. Though such an infrastructure could be adapted to apply graph queralytics at scale, the authors do not discuss the combination of queries and analytics, nor do they propose languages along these lines.

Most modern graph query languages directly support some built-in analytical features. SPARQL 1.1 (Harris & Seaborne, 2013) introduced *property paths* (Kostylev, Reutter, Romero, & Vrgoc, 2015) that allow for specifying regular expressions on paths; these can then be used in the context of a SPARQL query to find pairs of nodes connected by some path matching the regular expression. The Cypher query language for property graphs (Francis et al., 2018) (used by the Neo4j graph database (Miller, 2013)) also allows for querying on paths; though limited in terms of the regular expressions it allows on paths when compared to SPARQL 1.1, it offers features that SPARQL 1.1 does not, including shortest paths, returning paths, etc. The G-CORE query language (Angles et al., 2018) also supports features relating to paths, allowing to store and label paths, find weighted shortest paths, and more besides. In general, however, graph query languages tend to only support analytics relating to path finding and reachability (Angles et al., 2017).

The Gremlin language (Rodriguez, 2015) is more imperative in style than the aforementioned query languages, allowing to express analytical tasks through graph traversals. Per the Trinity.RDF system (Zeng et al., 2013), graph traversals, when combined with variables, can be used to express and evaluate, for example, basic graph patterns (Angles et al., 2018). Gremlin (Rodriguez, 2015) also supports some declarative query operators, such as union, projection, negation, path expressions, and so forth, along with recursion, which allows to capture general analytical tasks; in fact, the Gremlin language is Turing complete (Rodriguez, 2015).

In the context of SQL, languages such as Shark (Xin, Rosen, et al., 2013) have been proposed that allow SQL queries to be embedded and executed in the context of distributed frameworks (in this case Spark (Zaharia et al., 2016)) within which analytics can also be imperatively coded. Aside from embedding SQL into imperative languages, a number of languages have recently been proposed to combine relational algebra with linear algebra – including LARA (Hutchison, Howe, & Suciu, 2017) and MATLANG (Brijder, Geerts, den Bussche, & Weerwag, 2018) – based on the observation that although relational algebra is often used for declarative querying, and linear algebra for learning and analytics, many operations in relational algebra can be simulated with linear algebra, and vice-versa, where it is thus of interest to understand the expressive power of both and how they complement each other (Geerts, 2019).

Recursive Graph Queries. In addition to Recursive SPARQL, there are other approaches (such as Property Paths) for computing recursive queries. As we have discussed before, most query languages support recursively matching path expressions in a graph; however, per Example 4.1.1, more powerful forms of recursion are needed in the context of graph query languages to support the general class of analytics that we target here[1].

In later work, Corby et al. (Corby et al., 2017) proposed the LDScript language, which supports the definition of functions using SPARQL expressions; local variables that can store individual values, lists or the results of queries; and iteration over lists of values using loops, as well as recursive function calls. Recently Urzúa and Gutierrez (Urzua & Gutiérrez, 2019) proposed an extension of the G-CORE language to support linear recursion, and show how the resulting language can be used in principle to express various graph algorithms, such as a topological sort, which cannot be expressed in G-CORE without recursion.

4.3. The definition of SPARQAL

Recursion stands out in the literature as a key feature for supporting graph analytics. Our proposal – called SPARQAL – also extends SPARQL (1.1) with recursion by

[1]Though more complex forms of "navigational patterns" have been proposed in the literature, they are mostly limited to path-finding and reachability (Angles et al., 2017).

allowing to iteratively evaluate queries (optionally) joined with solution sequences of prior queries until some condition is met. In order to support this form of iteration, we need two key operators. First, we extend SPARQL with *solution variables* to which the results of a SELECT query can be assigned, and which can then be used within other queries to join solutions. Second, we extend SPARQL with `do-until` *loops* to support iteratively repeating a sequence of SPARQL queries until some termination condition is met; this condition may satisfy a fixed number of iterations, a boolean `ASK` query, or a fixpoint on a solution variable (terminating when the set of solutions do not change).

We refer back to Example 4.1.1, which illustrates how our language can be used to address a relatively simple queralytic task. We now present the syntax of our language, and thereafter proceed to define the formal semantics. We finish the section with a second, more involved example for computing the p-index of authors in an area.

To formally define our language and give our examples we assume familiarity with SPARQL and basic notions of graph analytics algorithms. We use the standard syntax and semantics of SPARQL that we presented before.

4.3.1. Syntax

SPARQAL aims to be a minimalistic extension of the SPARQL language that allows to express queralytic tasks. Specifically, a task is defined as a *procedure*, which is a sequence of *statements*. A statement can be an *assignment*, *loop* or *return* statement.

Assignment: Assigns the solution sequence of a query to a solution variable. The syntax of an assignment statement is `LET var = (Q);` where `var` is a variable name and `Q` is a SPARQL SELECT query that may use constructs of the form `QVALUES(var)`.

Loop: Executes a sequence of statements until a termination condition holds. The syntax of a loop statement is `DO (S) UNTIL (condition);` where `S` is a sequence of statements and `condition` is one of the following three forms of termination condition: (1) `TIMES t`, where `t` is an integer greater than 0; (2) `FIXPOINT (var)`, where `var` is a solution variable; (3) `AQ`, an ASK query that may use `QVALUES`. Also, if the condition is an ASK query, we can negate (`!AQ`) the result of the query. We note that

the **FIXPOINT** condition may be simulated with an ASK query, however, it is simpler to write some procedures with this instruction.

Return: Specifies the solution sequence to be returned by the procedure. The syntax of a return statement is **RETURN** `(var);` where `var` is a solution variable.

We note that some instructions may use the **QVALUES** clause. This instruction works in the same way than the usual VALUES clause of SPARQL. The **QVALUES** allow us to prove inline data which is combined with the result of the query.

Finally, a SPARQAL *procedure* is a sequence of statements satisfying the following two conditions: (1) the last statement, and only the last statement, is a return statement; (2) all solution variables used in **QVALUES**, **FIXPOINT** and **RETURN** have been assigned by **LET** in a previous statement (or a nested statement thereof).

EXAMPLE 4.3.1. Figure 4.1 illustrated a SPARQAL procedure with three statements: an assignment statement (lines 1–7); a loop statement with a fixpoint termination condition and two nested assignments (lines 8–22); and a final return statement (line 23). □

4.3.2. Semantics

We now give the semantics of statements that form procedures in SPARQAL. More formally, let $P = s_1; \ldots; s_n$ be a sequence of statements, and let

$$\texttt{var\_1}, \ldots, \texttt{var\_k}$$

be all variables mentioned in any statement in P (including in nested statements). For a tuple $\text{val}_0 = (r_1, \ldots, r_k)$ of initial assignments of (possibly empty) solution sequences to variables $\texttt{var\_1}, \ldots, \texttt{var\_k}$, we will construct a sequence $\text{val}_0, \ldots, \text{val}_n$ of k-tuples, where each val_i represents the value of all variables after executing statement s_i.

The construction is done inductively. Assume that $\text{val}_{i-1} = (r_1, \ldots, r_k)$. The value of val_i depends on whether s_i is an assignment, loop or return statement.

First, if s_i is the assignment statement **LET** `var_j = (Q);`, then tuple val_i is constructed as follows. Define SPARQL query $Q[(\texttt{var\_1}, \ldots, \texttt{var\_k}) \mapsto (r_1, \ldots, r_k)]$ as

the result of substituting each subquery {**QVALUES** (var_j)} in Q for the solution sequence r_j[2], and let r^* be the result of evaluating this extended query over the database. Then, substituting r_j for r^* in the tuple val_{i-1}, we define

$$\text{val}_i = (r_1, \ldots, r_{j-1}, r^*, r_{j+1}, r_k).$$

Next, if s_i is the loop statement **DO** (S) **UNTIL** (condition); the tuple val_i is constructed as follows. Assume that S is the sequence $s'_1, \ldots, s'_\ell$ and notice that (by definition) S must use a subset of the k solution variables in P. Repeat the following steps until the terminating condition is met:

(i) Initialise $\text{val}'_0 :- \text{val}_{i-1}$.

(ii) Compute the tuple val'_ℓ that represents the result of executing statements $s'_1, \ldots, s'_\ell$.

(iii) If val'_ℓ does not satisfy the condition, set $\text{val}'_0 :- \text{val}'_\ell$ and repeat step 2 above.

(iv) Otherwise finish, and set $\text{val}_i :- \text{val}'_\ell$.

To define when a tuple val'_ℓ over k variables satisfies a condition, we have three cases:

- If the condition is **TIMES** t, then the condition is met once the loop above has repeated t times.
- If the condition is **FIXPOINT** (var_j), then the condition is met when the j-th component of val'_ℓ contains the same set of solutions as the j-th component of val'_0.
- If the condition is AQ, then the condition is met when the ASK query

$$AQ[(\texttt{var\_1}, \ldots, \texttt{var\_k}) \mapsto \text{val}'_\ell]$$

evaluates to true. If the condition is a negated ASK query (!AQ), then the condition is met when the ASK query evaluates to false.

Finally, if s_i is the return statement **RETURN** (var_j), then the program terminates and returns the solution sequence r_j that is the j-th component of val_i.

[2] A syntactic way of doing this is to use a VALUES command in SPARQL.

Note that we assume all solution variables to have a global scope as it makes the semantics simpler to define; one could define local solution variables analogously. Moreover, some SPARQAL statements may incur infinite loops; later we will discuss fragments for which every program can be shown to terminate (as in, e.g., Datalog or recursive SPARQL). Currently we do not consider blank nodes when checking **FIXPOINT** conditions; these could be supported in a future version using the labelling of (Hogan, 2017), which has been shown to be efficient for a wide variety of graphs.

EXAMPLE 4.3.2. We recall Example 4.1.1, this time to illustrate the semantics of SPARQAL. In the first **LET** statement, we assign the solution sequence of the given SPARQL query to the variable `reachable`. Then the procedure enters a loop. We assign `adjacent` to the results of a SPARQL query that embeds the current solutions of `reachable` as a sub-query, leading to a join between current `reachable` stations and pairs of adjacent stations not on Line C. We then update the `reachable` solutions, adding `adjacent` solutions; here we can use `reachable` in the **LET** and **QVALUES** of the same statement since it was assigned before (line 1). In each iteration the solutions for `reachable` will increase, discovering new stations adjacent to previous ones, until a fixpoint. Finally, the **RETURN** clause specifies the solutions to be given as a result of the procedure. □

4.3.3. Example with PageRank

We now illustrate a procedure for a more complex queralytic.

EXAMPLE 4.3.3. Suppose we have the citation network of articles on a topic of interest and, we want to compute a centrality algorithm in order to know which articles of the network are the most important. Thereafter we wish to use these scores to find the most prominent authors in the area. We can express this task using SPARQAL. In this case we will consider the citation network of all the articles about the Zika virus on Wikidata, where we then encode and apply the PageRank algorithm over the citation network, using the resulting article scores to compute p-indexes for the respective authors. We show a procedure in our language for solving this task in Figure 4.2.

```
LET zika = ( # directed graph of citations between Zika articles
  SELECT ?node ?cite WHERE {
    ?node wdt:P31 wd:Q13442814 ; wdt:P921 wd:Q202864 ; wdt:P2860 ?cite .
    ?cite wdt:P31 wd:Q13442814 ; wdt:P921 wd:Q202864 .
  }
);
LET nodes = ( # all nodes of Zika graph
  SELECT DISTINCT ?node WHERE {
    { QVALUES(zika) } UNION
    { SELECT (?cite AS ?node) WHERE { QVALUES(zika) } }
  }
);
LET n = ( # number of nodes in Zika graph
  SELECT (COUNT(*) AS ?n) WHERE { QVALUES(nodes) }
);
# out-degree (>1) of nodes in Zika graph
LET degree = (
  SELECT ?node (COUNT(?cite) AS ?degree) WHERE { QVALUES(zika) } GROUP BY ?node
);
LET rank = ( # initial rank
  SELECT ?node (1.0/?n AS ?rank) WHERE { QVALUES(nodes) . QVALUES(n) }
);
DO ( # begin 10 iterations of PageRank
  LET rank_edge = ( # spread rank to neighbours via edges
    SELECT (?cite AS ?node) (SUM(?rank*0.85/?degree) AS ?rankEdge)
    WHERE {
      QVALUES(degree) . QVALUES(rank) . QVALUES(zika)
    } GROUP BY ?cite
  );
  LET unshared = ( # compute total rank not shared via edges
    SELECT (1-SUM(?rankEdge) AS ?unshared) WHERE { QVALUES(rank_edge) }
  );
  LET rank = ( # split and add unshared rank to each node
    SELECT ?node (COALESCE(?rankEdge,0)+(?unshared/?n) AS ?rank)
    WHERE {
      QVALUES(nodes) . QVALUES(n) . QVALUES(unshared) .
      OPTIONAL { QVALUES(rank_edge) }
    }
  );
) UNTIL (TIMES 10);
LET p_index_top = ( # compute p-index for authors, select top author
  SELECT ?author (SUM(?rank) AS ?p_index) WHERE {
    QVALUES(rank) . ?node wdt:P50 ?author .
  } GROUP BY ?author ORDER BY DESC(?p_index) LIMIT 1
);
RETURN(p_index_top);
```

FIGURE 4.2. Procedure to compute the top author in terms of p-index for articles about the Zika virus.

In this procedure we start by defining a variable that contains a solution sequence with pairs (`?node`, `?cite`) such that both `?node` and `?cite` are instances of (`P31`) scientific articles (`Q13442814`) about (`P921`) the Zika virus (`Q202864`) and `?node` cites (`P2860`) `?cite`. The solutions for this query are assigned to `zika`. We can think

of this variable as the representation of a directed subgraph extracted from Wikidata. We also define the variables `nodes` with all nodes in the subgraph, `n` with the number of nodes, and `degree` with the out-degree of all nodes in the graph (with some out-edge).

After extracting the graph and preparing some data structures for it, we then start the process of computing PageRank. First we assign the variable `rank` with initial ranks for all nodes of $\frac{1}{n}$. We then start a loop where we will execute 10 iterations of PageRank.[3] In each iteration we will first compute and assign to `rank_edge` the PageRank that each node shares with its neighbours; here we assume a damping factor $d = 0.85$ as typical for PageRank (Page, Brin, Motwani, & Winograd, 1999), denoting the ratio of rank that a node shares with its neighbours. Next we compute and assign to `unshared` the total rank not shared with neighbours in the previous step (this arises from nodes with no out-edges and the $1 - d$ factor not used previously for other nodes). We conclude the iteration by allocating the unshared rank to each node equally, updating the results for `rank`. The loop is applied 10 times.

Subsequently, we join the PageRank scores for articles with their authors, and use aggregation to sum the scores for each author, applying ordering and a limit to select the top author according to that sum, assigning the solution to `p_index_top`. Finally, the procedure returns the solution for `p_index_top` denoting the top author. □

4.3.4. Graph Updates

Although there is a straightforward way to implement our language on top of any engine using the VALUES clause, this option does not scale for larger data. Hence we define a recursive approach for graphs that serves as an alternative to SPARQAL for expressing queralytics; this approach is based on the techniques presented in Chapter 3 for creating temporary graphs. As a motivating example, consider the declaration of variables `zika` and `degree`, in lines 1 and 15 respectively of Figure 4.2. These statements initialize these variables, but we can view them as queries constructing two graphs. More precisely, we use graph `ex:zika` to store the result of the query:

[3] We select this termination condition for simplicity; we could also implement, for example, conditions based on residual norm, correlation coefficients, etc.

```
1 CONSTRUCT { ?node ex:zikacites ?cite } WHERE {
2  ?node wdt:P31 wd:Q13442814; wdt:P921 wd:Q202864; wdt:P2860 ?cite .
3  ?cite wdt:P31 wd:Q13442814; wdt:P921 wd:Q202864
4 }
```

Thus, instead of storing pairs of values for `<node> <cite>` in a SPARQAL solution variable `zika`, we store them as triples of the form `<node> ex:zikacites <cite>` in a graph named `ex:zika`. Using this graph we can now store the result of `degree` in graph `ex:degree` by means of the following query:

```
1 CONSTRUCT { ?node ex:zikadegree ?degree } WHERE {
2   SELECT ?node (COUNT(?cite) AS ?degree) WHERE {
3     GRAPH ex:zika {?node ?p ? cite}
4   } GROUP BY ?node
5 }
```

The approach of Graph Updates: Let $\mathcal{G} = \{(n_1, G_1), \ldots, (n_k, G_k)\}$ be a set of named graphs with IRIs $\{n_1, \ldots, n_k\}$ and RDF graphs $\{G_1, \ldots, G_k\}$ such that $n_i = n_j$ if and only if $i = j$. Let Q be a `CONSTRUCT` query. Given an IRI n, we use $n \leftarrow Q$ to express the action of storing the result G of $Q(\mathcal{G})$ as the named graph (n, G) in $\mathcal{G}$, overwriting the graph previously named n if necessary. Our approach of updates consists of (1) update expressions of the form $n \leftarrow Q$, for n an IRI and Q a `CONSTRUCT` query that may reference any of the existing graphs in $\mathcal{G}$, (2) loop expressions of the form **`DO`** `A` **`WHILE`** `(condition)` where `A` is a sequence of expressions and `(condition)` is again one of **`TIMES`** `t`; **`FIXPOINT`** `n`, where `n` is a graph name in $\mathcal{G}$; or `AQ`, an `ASK` query that may reference graphs in $\mathcal{G}$.

With respect to the semantics of this approach, starting with the initial set $\mathcal{G}$, an expression modifies graphs in $\mathcal{G}$ as follows. An assignment expression $n \leftarrow Q$ removes the graph (n, G) from $\mathcal{G}$ (if it exists), and adds $(n, Q(\mathcal{G}))$, where $Q(\mathcal{G})$ denotes the evaluation of Q over $\mathcal{G}$. A loop expression **`DO`** `A` **`WHILE`** `(condition)` applies iteration, evaluating the sequence `A`: t times if `condition` is **`TIMES`** `t`, or until the named graph $(n, G) \in \mathcal{G}$ did not change at the end of two subsequent iterations if

`condition` is **`FIXPOINT`** `n`, or until the evaluation of query `AQ` over $\mathcal{G}$ returns false if `condition` is `AQ`.

Given an expression A dealing with graphs in $\mathcal{G}$, we use $A(\mathcal{G})$ to denote the result of evaluating A over $\mathcal{G}$. Looking at our motivating example, one sees that transforming our procedural language into the approach of Graph Updates is not difficult, and neither is transforming Graph Updates expressions into our procedural language. The following proposition summarises the claim that both languages have the same expressive power.

PROPOSITION 4.3.1. Let P be a SPARQAL procedure, with v the solution variable returned by P. Then one can construct an expression A of Graph Updates mentioning a set $\mathcal{G}$ of graphs, and a SELECT query Q, such that evaluating Q over $A(\mathcal{G})$ yields the same solutions as those stored by v after evaluating P over $\mathcal{G}$. Likewise, for a Graph Updates expression A mentioning graphs $\mathcal{G}$, and any named graph $(n, G) \in \mathcal{G}$, one can construct a SPARQAL procedure P returning a solution variable v over $\mathcal{G}$, and a `CONSTRUCT` query Q, such that evaluating Q over the solutions stored by v yields the graph G.

Thus, we now have two strategies for implementing SPARQAL procedures: we can implement them directly by translating **`QVALUES`** clauses as VALUES statements while running the procedures, or we can compile the procedure into an expression of Graph Updates and evaluate it directly. We will analyze these two possibilities in Section 4.5.3, but first we study the expressive power of these formalisms.

4.4. Expressive Power

In this section we review the expressive power of procedures in SPARQAL. It is clear that our language is more expressive that Recursive SPARQL, since we have fixpoint but we can also store arbitrary queries in variables, that can potentially include BIND clauses or negation. But which is the expressive power of SPARQAL? In this section we prove that our language is Turing-complete. We also give complexity results.

4.4.1. Turing-completeness

Although `do-until` loops may appear to be just a mild extension to a query language, our first result states that this is actually enough to achieve Turing-completeness. Formally, we say that a query language $\mathcal{L}$ is Turing-complete if for every Turing machine M over an alphabet Σ one can construct a query Q in $\mathcal{L}$ and define a computable function f that takes a word in Σ^* and produces an RDF graph, and such that a word $w \in \Sigma^*$ is accepted by M if and only if the evaluation of Q over the graph $f(w)$ produces a non-empty result. Along these lines, we prove the following result:

THEOREM 4.4.1. SPARQAL is Turing-complete

PROOF. The proof of this theorem relies on the combination of `do-while` loops and the ability to create new values in the base SPARQL language through BIND statements and algebraic functions (Harris & Seaborne, 2013). Of course, for the proof one must assume that there is no limit on the memory used by the evaluation algorithm; however, the proof reveals a linear correspondence between the memory used by the query and the number of cells visited by the machine M. This proof is inspired by previous work of Churchill et. al. showing the Turing-completeness of a particular card game (Magic: The Gathering) (Churchill, Biderman, & Herrick, 2019).

Let $M = (Q, F, \Sigma \cup \{B\}, q_0, \delta)$ be a deterministic Turing machine, where $Q = \{q_o, \ldots, q_m\}$ is the set of states, there is a single final state $F = \{q_m\}$, Σ is the alphabet, B is the blank node covering all cells and δ is the transition function. Without loss of generality, and for readability, we assume that $\Sigma = \{0, 1\}$ and that δ does not define transitions for q_m. Let also $w = a_0, \ldots, a_n$ be a binary string. We construct a graph G and a SPARQAL procedure P such that M accepts w if and only if P returns a non-empty mapping.

Let us first assume that all states in Q and characters 0, 1, B are represented by IRIs, and that we use IRIs `:right` and `:left`. Define T_δ as a set of tuples of arity 5 containing one tuple (q, a, q', b, d) for each transition in δ of the form $\delta(q, a) = (q', b, d)$, for $d \in$ `:right`, `:left`.

For readability we will not make the distinction between graph and program, and rather initialize everything in the program. But the construction can be easily adapted so that the input is not coded directly in the program but is queried from a graph. The procedure P consists of the following groups of statements.

Initialization:

First group of statements are in charge of initializing some of the solution variables. The idea of variable `transition` is to store the transitions of M. The solution variable `current` stores the content of the current cell that M is pointing at, and the current state of the run. Solution variables `positive_cells` and `negative_cells` store, respectively, all cells to the right of the head of M and all cells to the left of the head of M. Of course, the tape is infinite, but we only need to store cells we have already visited.

```
1 LET transition = (
2   SELECT ?oldstate ?oldsymbol ?newstate ?newsymbol ?direction
3   WHERE {
4     VALUES (?oldstate ?oldsymbol ?newstate ?newsymbol ?direction) {
5       T_δ
6     }
7   }
8 );
```

```
1 LET current = (
2   SELECT  ?c_symbol ?c_state WHERE {
3     VALUES ( ?c_symbol ?c_state) {(a0, q0}
4   }
5 );
```

```
1 LET positive_cells = (
2   SELECT ?p_pos ?p_symbol WHERE {
3     VALUES (?p_pos ?p_symbol ) {(1, a1),...,(n,an)}
4   }
5 );
```

Loop: The loop phase of the procedure is as follows:

```
DO (
  S1
  S2
  S3
  S4
) UNTIL ( C );
```

Where all inner statements and conditions are defined next. The idea is that queries are used to check when the transition demands moving to the right or to the left, and depending on these values we update the cells accordingly. We use `new_current` as a temporal variable that will store the pointed cell and state of the machine in the next step of the run.

Statement `S1`: This statement stores the new symbol and the new state in which the machine shall advance on the next step of the computation.

```
LET new_current = (
  SELECT ?c_symbol ?c_state WHERE {
    SELECT (?newstate AS ?c_state) WHERE {
      QVALUES(transition)
      QVALUES(current)
      FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
    } .
    SELECT (?symbol AS ?c_symbol) WHERE {
      QVALUES(positive_cells)
      FILTER(?p_pos = 1)
      BIND(IF(!bound(?p_pos),"B",?p_symbol) AS ?symbol)
    }
  }
);
```

Here the first part of the query simply states that the new state is obtained by looking at what new state in the set of transitions fit the current symbols and states; note how the subquery

```
QVALUES(transition)
```

```
2  QVALUES(current)
3  FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
```

retrieves a single mapping if there is a transition for the current state and symbol, or is empty otherwise. This single mapping stores the new symbol, state and direction in variables `?newstate`, `?newsymbol` and `?direction`, respectively. The second part of the query produces the new symbol. If variable `positive_cells` is non-empty, this means that there are symbols to the right of the tape in positions we have previously visited, and so we just select the first of those symbols (position 1). The BIND statement ensures that if `positive_cells` is empty, then the current symbol is a blank, because we enter a portion of the tape we have not yet visited.

Statement `S2`: This statement is in charge of updating the set of positions and symbols to the right of the new position of the tape.

```
1  LET positive_cells = (
2    SELECT ?p_pos ?p_symbol WHERE {
3      {
4        SELECT (?p_pos -1 AS ?p_pos) ?p_symbol WHERE {
5          QVALUES(positive_cells)
6          QVALUES(transition)
7        QVALUES(current)
8          FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
9          FILTER(?direction=:right)
10       FILTER(?p_pos>1)
11       }
12     } UNION
13     {
14       SELECT (?p_pos + 1 AS ?p_pos) ?p_symbol WHERE {
15         QVALUES(positive_cells)
16         QVALUES(transition)
17         QVALUES(current)
18         FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
19         FILTER(?direction=:left)
20       }
```

```
21      } UNION
22      {
23        SELECT (1 AS ?p_pos) (?newsymbol as ?p_symbol) WHERE {
24          QVALUES(transition)
25          QVALUES(current)
26          FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
27          FILTER(?direction=:left)
28        }
29      }
30    }
31  );
```

The first part of the query has to do when the machine moves right (note that if the transition moves left then this will be empty, which is the intended behavior). In this case, it adds into `positive_cells` all those mappings representing pairs (position,symbol) already in `positive_cells`, but such that they are not in position 1 (because that position will become the new current position). Consequently, the new position is updated to reflect that the head now moves closer to every cell already in `positive_cells`.

The second part has to do when the machine is due to move left. In this case, there are two things that must be done. First, every mapping already in `positive_cells` remains in `positive_cells`, but the position is incremented by 1, to reflect that the head moved left. Next, the symbol that is due to be written by the current transition is also added as a new mapping into `positive_cells`, with position 1 .

Statement `S3`: This statement is the analogous of `S2`, this time dealing with negative cells.

```
1  LET negative_cells = (
2    SELECT ?n_pos ?n_symbol WHERE {
3      {
4        SELECT (?n_pos + 1 AS ?n_pos) ?n_symbol WHERE {
5          QVALUES(negative_cells)
6          QVALUES(transition)
```

```
7           QVALUES(current)
8           FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
9           FILTER(?direction=:left)
10          FILTER(?n_pos<-1)
11        }
12      } UNION
13      {
14        SELECT (?n_pos - 1 AS ?n_pos) ?n_symbol WHERE {
15          QVALUES(negative_cells)
16          QVALUES(transition)
17          QVALUES(current)
18          FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
19          FILTER(?direction =:right)
20        }
21      } UNION
22      {
23        SELECT (-1 AS ?n_pos) (?newsymbol AS ?n_symbol) WHERE {
24          QVALUES(transition)
25          QVALUES(current)
26          FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
27          FILTER(?direction =:right)
28        }
29      }
30    }
31  );
```

Statement S4: Once both `positive_cells` and `negative_cells` have been updated, we can store the new symbol and state in variable `current`, therefore executing the transition.

```
1  LET current = (
2    SELECT ?c_symbol ?c_state WHERE { QVALUES(new_current) }
3  );
```

Condition C: The condition simply verifies that one is actually able to find a suitable transition for the current symbols. If it is not possible, then we stop.

```
!(ASK {
  QVALUES(transition)
  QVALUES(current)
  FILTER(?oldstate=?c_stat && ?oldsymbol=?c_symbol)
})
```

Return: Finally, after the loop, we return the state.

```
LET state = (
  SELECT ?state WHERE { QVALUES(current) FILTER(?c_state = :qm) }
);
RETURN(state);
```

One can now check that this program effectively returns a non-empty mapping if and only if the procedure P terminates and the variable `current` stores the state q_m. In turn, this happens if and only if M accepts on the input. This finishes the proof. □

Traditional theoretical results have tended to study languages assuming that the creation of new values is not possible, or, if possible, that there is a bound on the number of values that are created. But this is not the case with SPARQAL procedures; for starters, we can iterate and sum to create arbitrarily big numbers. However, for the purpose of comparing SPARQAL procedures against other traditional database languages, we ask, what would be its expressive power if one disallows the creation of new values? In fact, `do-while` loops have been studied previously in the literature, especially in the context of relational algebra (see e.g. (Abiteboul et al., 1995)). In our context, we ask what happens if we disallow the invention of new values in the procedure: more formally, we say that a procedure P *does not invent new values* if for every graph G and every variable `var` defined in P, all mappings in any solution sequence associated to `var` always binds variables to values already present in G. In this case, there is a limit on the maximum number of mappings in the solution sequence of any variable at any point in time during evaluation of the procedure, and this limit depends polynomially on the size of the graph. This implies that the evaluation of this procedure can

be performed in PSPACE (in data complexity), and we can also show that this bound is tight. To formally state this result, let P be a SPARQAL procedure. The evaluation problem for P receives a graph as an input, and asks whether the evaluation of P over G is not empty.[4] We can then state the following:

PROPOSITION 4.4.1. The evaluation problem for SPARQAL procedures that do not invent new values is PSPACE-complete.

PROOF. The proof goes in three steps: for a SPARQAL procedure that do not invent new values, we need to store only (1) the current state of all variables, (2) the previous state of variables in fixed-point clauses, and (3) the current number of iterations for the case of loops with a max number (which is bounded by the query, as we do not need more iterations that the number stated). Additionally, SPARQL queries can themselves be computed in PSPACE, giving us the upper bound.

For the lower bound we can use the construction in Theorem 4.4.1. Because we know that the machine M runs in PSPACE, the number of cells visited is bounded by a number which depends on the elements on the graph. Let then $|G|$ be the size of the graph, and assume that $n = |G|^k$ is the number of maximum cells visited in any computation of M over a graph with size $|G|$. The first thing we need is to construct a linear order from the elements of the graph, which we will store in a solution variable `order`. We can do this with a do–until iteration that keeps adding elements until there are no more to add. We can then extend this linear order into an order of $2k$ tuples, which will be stored in a solution variable `full-order`. With this full order we can now pre-compute all possible n cells that may be visited by M in solution variables `positive_cells` and `negative_cells`. We cannot use a numeric position anymore, but we can use our tuples in full order as the position. With these cells pre-computed, we need to invoke the rest of the procedure. However, the last modification we make is that all arithmetic is replaced by the appropriate operation that uses our linear order. □

[4]This corresponds to boolean evaluation. This is without loss of generality because the standard evaluation problem where one considers a tuple of values as an input can be simulated by means of filters.

Note that this result gives us an interesting comparison between Recursive SPARQL and SPARQAL. We showed in Section 3.3.5 that the evaluation of Recursive Query is a PTIME-complete problem. Then, recursive SPARQL cannot simulate SPARQAL procedures that do not invent new values unless PTIME= PSPACE, which is widely assumed to be false.

4.5. Experiments

In this section we present our prototypical implementation of a queralytics engine based on the SPARQAL language, along with experiments over two datasets to ascertain its performance and limitations. The goals of this prototype are to demonstrate that the language can be used, in practice, to express in-database analytics, and to ascertain the performance achievable when operating over an off-the-shelf SPARQL query engine. The target use-cases for our prototype is – per the scenarios outlined in Examples 4.1.1 and 4.3.3 – to run queralytics (near-)interactively on small-to-medium graphs projected from a larger graph using a query. Along these lines, the prototype was developed on top of the Apache Jena Framework, version 3.10. The core implementation provides the following core functionalities: (1) it parses a SPARQAL procedure into a sequence of statements, which are evaluated according to their semantics by: (2a) maintaining a map of solution variables to solution sequences; (2b) replacing variables used within a `QVALUES` clause with a VALUES string with the respective solution sequence; (2c) evaluating SPARQL queries, and (2d) in order to handle `FIXPOINT` conditions, keeping the previous solution sequence of the respective variable in-memory to track changes. We also provide a rough prototype for the Graph Updates strategy defined in Section 4.3.4.

We first report results for our two motivating scenarios. We then design a benchmark based on Wikidata for running analytical tasks on sub-graphs that are similarly extracted through queries. Finally, we stress-test our prototype for a graph analytics benchmark at a larger scale to identify choke points. In particular, we show that the Graph Updates approach may be better suited at handling large datasets. Experiments were tested on a MacBook Pro with a 3.1 GHz Intel I5 processor and 16 GB of RAM.

TABLE 4.1. Top-3 authors according to their Zika p-index.

?author	?p_index	?name
wd:Q18876341	0.124	George Dick
wd:Q24696365	0.084	Ademola H. Fagbami
wd:Q21165078	0.083	Alexander John Haddow

The source code, procedures and datasets used are available as supplementary material (see Appendix A).

4.5.1. Wikidata: Motivating Examples

Our first experiment is to anecdotally evaluate the procedures described in Examples 4.1.1 and 4.3.3, evaluating the Buenos Aires metro and Zika p-index queralytics. Example 4.1.1 took just 1.3 seconds to return 16 stations from which Palermo can be reached without using Line C. Example 4.3.3 – running 10 iterations of PageRank on a graph of 38,738 edges (citations) and 3,057 nodes (articles) – took 53.1 seconds to find the top author (from 2,214 authors) according to their p-index in the citation network. For reference, Table 4.1 shows the results for the top 3 such authors, ordered by their p-index.

4.5.2. Wikidata: Queralytics Benchmark

To the best of our knowledge, there is no existing benchmark for queralytics along the lines discussed in this paper. This led us to design a novel benchmark for queralytics over the Wikidata knowledge graph. We took the "truthy" RDF dump of Wikidata as our benchmark graph (Malyshev et al., 2018). Designing the queralytic tasks required collecting and combining two elements: queries that return results corresponding to graphs, and graph algorithms to apply analytics on these graphs. In terms of the queries returning graphs, we revised the list of use-case queries for the Wikidata Query Service[5]. From this list, we identified the following six queries returning graphs:

Q1: A graph of adjacent metro stations in Buenos Aires

Q2: A graph of citations for articles about the Zika virus

[5] https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples

TABLE 4.2. Number of nodes and edges in graphs considered .

	Q1	Q2	Q3	Q4	Q5	Q6
Nodes	93	3,057	480	266	7,194	627
Edges	172	38,738	766	211	8,719	996

Q3: A graph of characters in the Marvel universe and the groups they belong to

Q4: A graph of firearm cartridges and the cartridges they are based on

Q5: A graph of horses and their lineage

Q6: A graph of drug–disease interactions on infectious diseases

These queries provide a mix of connected graphs, disconnected graphs, bipartite graphs, trees, DAGs, near-DAGs, and so forth. We provide the sizes of these graphs in Table 4.2, where we see that the smallest graph is indeed the Buenos Aires metro graph, while the largest is the citation graph for Zika articles.

Next we must define the analytics that we would like to apply on these graphs. For this, we adopted five of the six algorithm sproposed for the Graphalytics Benchmark (LDBC, 2019) defined by the Linked Data Benchmark Council (LDBC); namely:

BFS Breadth-First Search
LCC Local Clustering Coefficient
SSSP Single-Source Shortest Path
PR PageRank
WCC Weakly Connected Components

We do not include the *Community Detection through Label Propagation* **CDLP** as it assumes data with initial labels. We implement these five algorithms as procedures in the SPARQAL language, prefixing each with the six different Wikidata graph queries, stored as solution variables. The result is a benchmark of $6 \times 5 = 30$ queralytic tasks.

In Figure 4.3, we show the results for these 30 tasks using our in-memory implementation. First we remark that the Weakly Connected Components (**WCC**) algorithm timed-out in the case of the Zika graph after 10 minutes. While the cheapest algorithm in general was **BFS**, the most expensive was **WCC**. Although some of these tasks took over a minute in the case of graphs with thousands or tens of thousands of nodes

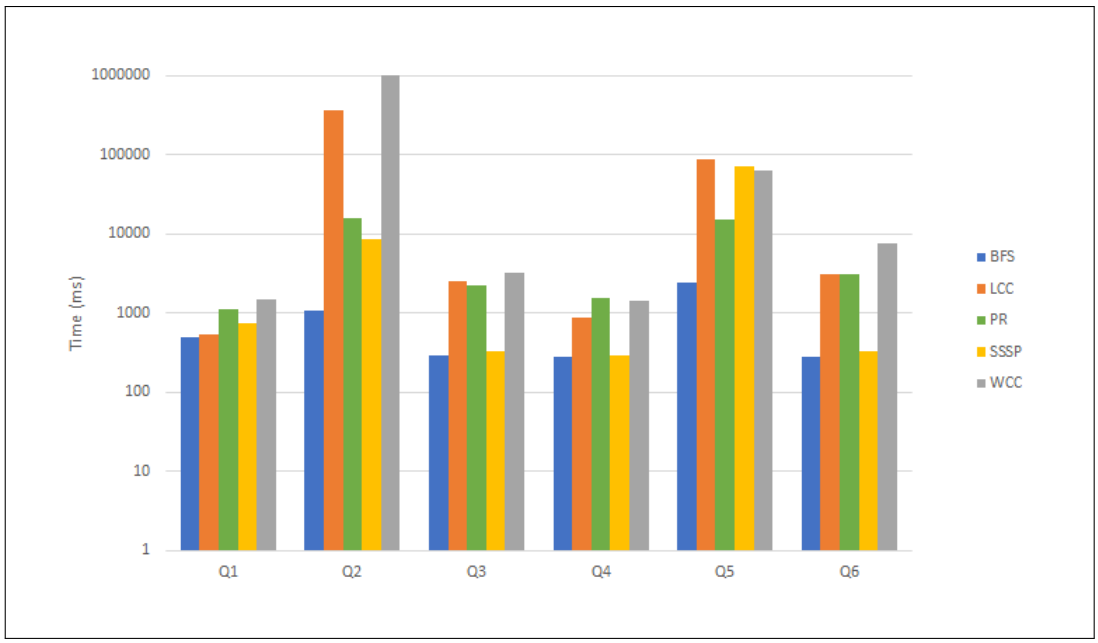


FIGURE 4.3. Result for Wikidata queralytic benchmark.

(Zika/**Q1** and Horses/**Q5**), those with fewer than a thousand nodes/edges ran in under a second, and thus would be compatible with interactive use.

4.5.3. Graphalytics: Stress Test

The scale of the previous graphs is quite low and uses (mostly) the in-memory algorithm. Hence we use the Graphalytics Benchmark (LDBC, 2019) to perform stress tests for our prototype at larger scale with the goal of identifying the choke points of the current implementation. We adopt the `cit-Patents` dataset: a directed graph with 3,774,768 vertices and 16,518,947 edges. We implement both alternatives for evaluating SPARQAL procedures: using VALUES and using Graph Updates.

The results of the Graphalytics benchmark are shown in Table 4.3. For the VALUES implementation, we identify two key choke points. An obvious choke-point is presented by the fact that solution sequences are stored in memory: this puts an upper-bound on scalability, leading to OOM errors for complex queralytics on larger graphs (more specifically, in this case, running **LCC** and **WCC** on a graph of millions of nodes and tens of millions of edges). The other choke-point is the handling of **QVALUES**

clauses using a VALUES clause with large solution sequences, yielding queries that are inefficient for Apache Jena. We view a number of possibilities for addressing these choke points in future work. Keeping with the in-database analytics scenario, the first choke point could be alleviated with compression and indexing techniques, while both choke points could be addressed by batch-at-a-time processing of **QVALUES** clauses.

The performance issues of the VALUES implementation are alleviated, up to some extent, when we switch to the implementation based on graph updates. Intermediate graphs are stored in memory, but their sizes tend to be smaller than the size of solution sequences, as one avoids replication. Here, the main choke-points are (1) the complexity of the join queries that we need to solve (2) the fact that constructed graphs are not indexed, and thus queries over them run slower. We speculate that lightweight indexes in constructed graphs would provide even faster times.

Another in-database alternative would be using GPU-acceleration for parallelizing batches. In general, however, in order to process larger graphs, an in-database solution may not be feasible, but rather SPARQAL procedures would need to be translated to tasks that can run on distributed graph processing frameworks, as discussed in Section 4.2.

Here we studied how a recursive extension of SPARQL can be useful to compute Graph Analytics procedures. We started with the main idea of Recursive SPARQL, but we also added to the language other key features needed for this approach, such as loops and the definition of variables. Although we can cover several use cases with our techniques, it is imperative to study ways to improve the computation of SPARQL techniques, as it may allow SPARQAL procedures to run faster. Thus, in the next chapter we study a way to accelerate the computation of a core SPARQL fragment, which is the computation of Basic Graph Patterns, using the idea of worst-case optimal join algorithms.

TABLE 4.3. Execution time (min) for Graphalytics benchmark. Here OOM is for out-of-memory error.

Algorithm	**BFS**	**LCC**	**PR**	**SSSP**	**WCC**
SPARQAL/Jena	11	OOM	250	300	OOM
SPARQAL/Updates	2	26	112	127	13

Chapter 5. A WORST-CASE OPTIMAL JOIN ALGORITHM FOR SPARQL

As we hinted at the introduction, and according to what we presented with the experiments for Recursive SPARQL and SPARQAL, we need techniques that can handle large datasets in an efficient way. In particular, one of the possible optimizations is to improve the evaluation of joins in SPARQL, since in general, recursive queries have to perform several joins before getting an answer. Moreover, we show in later experiments that SPARQL engines still struggle when evaluating queries with more complex joins; we argue that this is due, in part, to the fact that prominent SPARQL engines rely on traditional join algorithms that have not changed for over a decade.

On the other hand, a new family of join algorithms has received much attention in the recent database literature: the state-of-the-art for join evaluation has moved away from pairwise join evaluation (Ramakrishnan & Gehrke, 2000), towards multiway join evaluation where an arbitrary number of joins can be evaluated "at once". One of the main benefits of the multiway approach is to minimize the number of intermediate results generated. In fact, a variety of modern multiway join algorithms – including, for example, Leapfrog Triejoin (Veldhuizen, 2014), Minesweeper (Ngo et al., 2014), Tetris (Khamis et al., 2016), CacheTrieJoin (Kalinsky et al., 2017), etc. – have been proven to be *worst-case optimal* (Ngo, Porat, Ré, & Rudra, 2012; Ngo et al., 2013), meaning that the runtime of the algorithm is bounded by the worst-case cardinality of the query result (i.e. the AGM bound (Atserias et al., 2008)); this theoretical guarantee implies that no other join algorithm can exist that is asymptotically faster for all database instances. Several systems (e.g. Logicblox (Aref et al., 2015) and Emptyheaded (Aberger et al., 2017)) have further implemented these worst-case optimal strategies and demonstrated their superior performance in practice for evaluating queries with complex joins.

In this chapter, we explore the idea of incorporating a worst-case optimal join algorithm for evaluating basic graph patterns, which form the core of SPARQL queries. Although there are works studying the interaction of such algorithms for graph queries and analytics (Nguyen et al., 2015; Aberger et al., 2017; Kalinsky, Mishali, Hogan,

Etsion, & Kimelfeld, 2018), to the best of our knowledge, no such work has addressed the evaluation of SPARQL basic graph patterns. Thus, we aim to fill this gap by investigating the benefits of worst-case optimal join algorithms for evaluating basic graph patterns.

Given our goal that worst-case optimal join algorithms be widely adopted on the Semantic Web in the near future, we select Leapfrog Triejoin (LFTJ) (Veldhuizen, 2014) as our base algorithm since it is relatively straightforward to adapt to the case of SPARQL while still providing worst-case optimal guarantees. We propose some adaptations of the LTFJ algorithm for the SPARQL setting, proving that these adaptations do not affect the theoretical guarantees of the algorithm. We discuss how the resulting algorithm can be integrated and optimized within a native RDF store that supports multiple index orders and cardinality-based join ordering, reducing the cost of adoption. Analogously, we create a fork of Apache Jena (TDB) (Jena, 2015) that supports worst-case join evaluation, and proceed to evaluate its performance against the unmodified version of the engine, as well as two other prominent SPARQL engines: Virtuoso (Virtuoso, 2015) and Blazegraph (Thompson et al., 2014). We run experiments on the Berlin (Bizer & Schultz, 2009) and WatDiv (Aluç et al., 2014) SPARQL benchmarks, and thereafter on a novel benchmark based on Wikidata (Vrandečić & Krötzsch, 2014) from which we generate a large set of SPARQL basic graph patterns exhibiting a variety of increasingly complex join patterns. Our results show that our fork of Apache Jena can reduce the runtimes of queries with non-trivial joins in comparison with the baseline systems.

Before explaining our approach, we need to introduce some additional concepts for RDF and SPARQL that we use throughout this chapter.

5.1. Basic graph patterns

We keep the definition of RDF graphs and we have used so far, and also we keep the syntax and semantics of SPARQL. However, we need to keep in mind some essential concepts for this chapter.

We recall the definition of $[\![t]\!]_G$, that represents the evaluation of a triple pattern over an RDF graph G:

$$[\![t]\!]_G \quad = \quad \{\mu \mid \text{var}(t) = \text{dom}(\mu) \text{ and } \mu(t) \in G\}$$

In SPARQL queries it is common to query sets of triple patterns. A set of triple patterns is called a **Basic Graph Pattern** (abbreviated as **BGP**). Now we extend the definition of $[\![t]\!]_G$ for a **BGP** P of the form $P = (t_1 \ \mathsf{AND} \ t_2 \ \mathsf{AND} \ \ldots \ t_n)$ as:

$$[\![P]\!]_G = [\![\{t_1, \ldots, t_n\}]\!]_G \quad = \quad [\![t_1]\!]_G \bowtie ... \bowtie [\![t_n]\!]_G$$

Letting $\mu(P)$ denote the image of P under μ, with respect to the latter definition, we can equivalently say that $[\![P]\!]_G = \{\mu \mid \text{dom}(\mu) = \text{var}(P) \text{ and } \mu(P) \subseteq G\}$.

SPARQL further offers a wide range of query operators that can be used to combine or modify the results of basic graph patterns, such as union, optional, filters, aggregates, property paths, etc. In this chapter, we focus on optimizing the evaluation of basic graph patterns, which form the core of SPARQL queries; other SPARQL operators can be supported by applying standard techniques over the mappings generated from the query's basic graph patterns.[1] However, there is the possibility for bespoke methods that merge the evaluation of some of these operators – in particular optional, property paths, named graphs, etc. – with the evaluation of basic graph patterns by the proposed worst-case join algorithm. We leave the exploration of such embedded optimizations for future work. Furthermore, SPARQL assumes a default bag semantics, which preserves duplicates (Angles & Gutierrez, 2016); though we evaluate sets of solutions for basic graph patterns, such patterns alone never generate duplicate mappings, and thus our proposal is compatible with bag semantics being applied in higher-level query operators.

We defined the fragment of SPARQL that we optimize, Thus, we have to review two relevant concepts for our approach: the AGM bound and Worst-case optimal join algorithms.

[1] Other features like `BIND`, `VALUES`, `SERVICE`, etc., that generate or extend mappings can be evaluated in the standard way.

5.2. The AGM bound and worst-case optimal join algorithms

As hinted at the semantics, evaluating BGP is closely related to answering join queries in standard databases. Thus, we looked for new techniques for evaluating join queries that were inspired by the AGM bound (Grohe, 2013). These techniques are worst-case optimal join algorithms. Now we explain the idea behind them.

5.2.1. The AGM Bound

Suppose that you have a join query of the form $Q = R_1 \bowtie \ldots \bowtie R_m$ and a database instance with schema $R_1, ..., R_m$. One would like to bound the size of $Q(D)$ in terms of the sizes $N_i = |R_i(D)|$ of the input relations. If we know that one of the relations, namely R_i, contains all the attributes appearing in the query Q, we know that that we can bound the size of the query by the size of the relation $Q(D) \leq N_i$. Instead, if some relations $R_{i_1}, \ldots, R_{i_k}$ contain all the attributes in Q, then we know that $|Q(D)| \leq \prod_{j=1}^{k} N_{i_j}$. We call $R_{i_1}, \ldots, R_{i_k}$ an **edge cover** of the query Q. If we want to discover the best bound, we can solve an integer linear program with variables $x_1, \ldots, x_m$, where $x_i = 1$ means that R_i is in the edge cover. Now let $A_i, \ldots, A_n$ be the attributes appearing in Q, then our integer linear program is:

$$\begin{aligned} \text{minimize} \quad & \sum_i x_i \log N_i \\ \text{subject to} \quad & \sum_{i \text{ such that } A_j \in R_i} x_i \geq 1 \quad \text{for } j = 1, \ldots, n \\ & x_i \in \{0, 1\} \qquad\qquad\quad \text{for } i = 1, \ldots, m \end{aligned}$$

and it is clear that for every solution of the linear program the following inequality holds:

$$|Q(D)| \leq \prod_{i=1}^{m} N_i^{x_i} = 2^{\sum_i x_i \log N_i}$$

where we call the value $\rho(Q, D) = \sum_i x_i \log N_i$ the **edge cover number** of the query Q in D. Note that $\rho(Q, D)$ is the optimal solution of the program.

However, this bound can be improved. **A**tserias, **G**rohe and **M**arx proposed in (Atserias et al., 2008) the **AGM Bound**. What they found is that the optimal value for the linear programming relaxation of the integer linear program also gives us a bound

for the query. Thus if we replace the constraint $x_i \in \{0, 1\}$ for:

$$0 \leq x_i, \text{ for } i = 1, \ldots, m$$

we will get a solution $(x_1, \ldots, x_m) \in \mathbb{Q}^m$ that we call the **fractional edge cover** of Q. We also define the **fractional edge cover number** of Q in D as $\rho^*(Q, D) = \sum_i x_i \log N_i$. Finally, we have that given join query Q, for every database D it holds:

$$|Q(D)| \leq 2^{\rho^*(Q,D)}$$

and we call the value $2^{\rho^*(Q,D)}$ the AGM bound of the query. To clarify this we show the canonical example.

EXAMPLE 5.2.1. Consider the query $Q = R(a, b) \bowtie S(b, c) \bowtie T(c, a)$ and an instance D such that $|R| = |S| = |T| = |N|$. It is possible to bound the size of the query when it is evaluated over the instance D by N^2, since it is clear that the size of the subquery $R \bowtie S$ is N^2, and T can be considered as a filter. However, we can obtain a better bound if we solve the corresponding instance of the linear problem presented above:

$$\begin{array}{ll} \text{minimize} & \log N(x_R + x_S + x_T) \\ \text{subject to} & x_R + x_T \geq 1 \\ & x_R + x_S \geq 1 \\ & x_S + x_T \geq 1 \\ & x_R,\ x_S,\ x_T \geq 0 \end{array}$$

we will obtain the values $x_R = x_S = x_T = \frac{1}{2}$. Thus, we know that $\rho^*(Q, D) = \frac{3}{2}log(N)$, and then $|Q| \leq 2^{\frac{3}{2}log(N)}$. Finally we get $|Q| \leq N^{\frac{3}{2}}$.

The previous example gives us an important insight. Since join algorithms, such as the Selinger Algorithm, resolve queries by pairs, if we try to answer the query $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$, in the worst we will execute at least N^2 operations. However, we know that the size of the result is bounded by $N^{\frac{3}{2}}$, and this gives us an interesting question. Is there some algorithm where the number of steps of itself is

bounded by the size of the query? We show now how this question inspired a bunch of interesting new algorithms.

5.2.2. Worst-case optimal join algorithms

A join algorithm is called worst-case optimal if it satisfies the AGM bound, namely, if its running time is at most $2^{\rho^*(Q,D)}$ up to a logarithmic factor for a query Q and a database D. Various works in the database literature have focused on worst-case optimal join algorithms (Veldhuizen, 2014; Ngo et al., 2014; Nguyen et al., 2015; Khamis et al., 2016; Kalinsky et al., 2017), which have also been implemented as part of commercial databases (Aref et al., 2015; Aberger et al., 2017).

In (Ngo et al., 2013), the authors showed a constructive proof for the AGM bound. From this proof, one can get a Generic WCO Join Algorithm. It is possible to derive several WCO Join algorithms from this Generic Join, including the one that inspired our join algorithm for SPARQL: the Leapfrog Triejoin algorithm. The Generic Join algorithm is presented in Algorithm 3.

Algorithm 3 Generic-Join($\bowtie_{F\in\mathcal{E}} R_F$)

Input: A natural join query represented by an Hypergraph $\mathcal{H}(\mathcal{V},\mathcal{E})$, where $\mathcal{V}$ contains the attributes and each hyperedge represents a relation.
Output: The answer $\mathcal{A}$ for the query.

$\mathcal{A} \leftarrow \emptyset$
if $|\mathcal{V}| = 1$ **then**
    return $\cap_{F\in\mathcal{E}} R_F$
end if
Pick arbitrary I, J such that $\mathcal{V} = I \uplus J$
$L \leftarrow$ Generic-Join($\bowtie_{F\in\mathcal{E}_I} \pi_I(R_F)$)
for $t_I \in L$ **do**
    $Q[t_I] \leftarrow$ Generic-Join($\bowtie_{F\in\mathcal{E}_J} \pi_J(R_F \ltimes t_I)$)
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{t_I\} \times Q[t_I]$
end for
return $\mathsf{ans}(\boldsymbol{q}_2, D \cup \{\langle t, G_{\text{Temp}}\rangle\})$

An important thing to note is that this algorithm is equivalent to the Leapfrog Triejoin algorithm when we instantiate the Generic-Join as $\mathcal{V} = \{1,\ldots,n\}$ and $\mathcal{I} = \{1,\ldots,n-1\}$[2]. As the authors say, "*A mild assumption which is not very crucial*

[2]Note that $\mathcal{I}$ is chosen arbitrarily at each call of the algorithm.

is to pre-index all the relations so that the inputs to the subqueries $Q[t_I]$ can readily be available when the time comes to compute it". The Leapfrog Triejoin algorithm do this by fixing a global attribute order and build a tree index for each input relation consistent with this global attribute order. We discuss later in this chapter how the Leapfrog Triejoin algorithm works for a SPARQL setting and how we index the data to handle the requirement discussed above.

5.2.3. Worst-case optimal join algorithms for graph databases

Some of the works related to WCO algorithms have looked at the benefits of such algorithms for answering queries over graphs, incorporating experiments for evaluating queries based on graph patterns including cliques, trees, paths, etc. (Nguyen et al., 2015; Aberger et al., 2017; Kalinsky et al., 2017; Kankanamge, Sahu, Mhedbhi, Chen, & Salihoglu, 2017; Aberger, Tu, Olukotun, & Ré, 2016); moreover, Aberger et al. (Aberger et al., 2017) further provide experiments for analytical queries on graphs, such as PageRank and shortest paths.

While these works have provided evidence as to the value of worst-case optimal join algorithms for graphs, they do not address the SPARQL setting. One of the main differences is the indexation of the graph is based on storing binary relations; this makes the indexation easier but in the SPARQL setting, this approach removes the possibility of asking for queries where the predicate is a variable (and in SPARQL it is common the case where a triple pattern has a variable predicate). In order to support this feature, our indexes must allow to store triples and not binary relations, which increases the complexity of the implementation, mainly because we need to replicate the data several times in order to keep the theoretical guarantees of the WCO join algorithms.

Moreover, another key difference of this work and previous proposals, is that one of our goals is to implement the algorithm within a database system and not as a standalone implementation. This is because we believe that the performance of WCO join algorithms may differ when they have to interact with all the components of a system.

Now, to understand our approach, we first discuss the standard evaluation methods used in popular SPARQL engines. We also discuss the current indexing techniques and some approaches towards doing a multiway join evaluation.

5.3. Joins and SPARQL

Now we introduce the current techniques for computing a BGP. First we see how RDF databases index the data and then how the BGPs are resolved, while we discuss how these techniques are related to algorithms for resolving joins in SQL databases.

5.3.1. Indexing

In order to efficiently evaluate triple patterns, SPARQL engines employ indexes that offer optimized access to the underlying data; such engines will often build a complete index that can efficiently evaluate a triple pattern with any combination of constants and variables (Harth & Decker, 2005). A complete index is comprised of multiple index orders, where a single index order with prefix lookups can be used to evaluate multiple forms of triple pattern. Following (Harth & Decker, 2005), we use the notation $(s|?, p|?, o|?)$ to denote eight forms of triple patterns where, for example, $(?, p, o)$ refers to the set of triple patterns with variable subject, constant predicate and constant object: $\mathbf{V} \times \mathbf{I} \times \mathbf{IL}$. For example, the index order **pos** allows for directly evaluating triples patterns of the form $(?, ?, ?)$, $(?, p, ?)$, $(?, p, o)$ and (s, p, o) without filtering, but not $(s, ?, ?)$, which would require reading all triples from the **pos** index and filtering those whose subject does not match the triple pattern (a better choice would be an index order like **spo** or **sop**).

Some SPARQL engines build complete indexes for triples (Weiss, Karras, & Bernstein, 2008; Neumann & Weikum, 2008; Atre, Chaoji, Zaki, & Hendler, 2010), while others directly support named graphs by indexing quads (Harth & Decker, 2005; Virtuoso, 2015). In terms of indexing implementations, one option is to apply standard data structures known from relational databases, such as B+Tree indexes (Harth &

Decker, 2005; Neumann & Weikum, 2008; Virtuoso, 2015); another option is to develop RDF-specific techniques, such as nested data structures (Weiss et al., 2008), bit matrices (Atre et al., 2010), etc., that take advantage of the fixed arity of triples.

5.3.2. Pairwise joins

While a complete index allows individual triple patterns to be evaluated efficiently, the evaluation of basic graph patterns requires applying **join algorithms** over the mappings generated from triple patterns. The most popular strategy for evaluating basic graph patterns is to use pairwise evaluation joining two sets of mappings at a time. In left-deep plans, the results of a triple pattern are joined with the current results of all joins thus far; for example, taking a basic graph pattern with four triple patterns, an example left-deep evaluation would be $(((t_1 \bowtie t_2) \bowtie t_3) \bowtie t_4)$ (Harth & Decker, 2005). As we see, this is equivalent to join evaluation in SQL databases. In bushy plans, two sets of join results can also be joined, leading to more balanced query plans; for example, $((t_1 \bowtie t_2) \bowtie (t_3 \bowtie t_n))$ is an instance of a bushy plan (Neumann & Weikum, 2008).

To implement such joins, SPARQL engines often use variants of well-known algorithms for join evaluation in relational databases, such as nested-loop joins (Harth & Decker, 2005; Neumann & Weikum, 2008), hash joins (Neumann & Weikum, 2008), and sort-merge joins (Neumann & Weikum, 2008). An important aspect of optimizing SPARQL query plans is then to exploit the commutativity and associativity of joins to find a query plan that minimizes the number of intermediate results generated; a common strategy is to rely on cardinality estimates (Harth & Decker, 2005; Neumann & Weikum, 2008; Virtuoso, 2015).

As we discuss before, Pairwise join techniques are suboptimal in terms of the size of the result. This gave us room for exploring the benefits of worst-case optimal algorithms in RDF databases.

5.3.3. Multiway joins

Multiway join algorithms perform joins over two or more sets of mappings at once; a common strategy is to group, evaluate and join triple patterns sharing a given

variable as a single operation. Multiway join evaluation can thus reduce the number of intermediate results that are generated. To the best of our knowledge, few works have investigated multiway joins in the context of SPARQL. One exception is the recent work of Galkin et al. (Galkin et al., 2017), who propose a join algorithm for SPARQL queries called SMJOIN that groups blocks of star-shaped joins (where a common join variable is present in the subject position) and applies multiway joins over each block. Experimental results show that the multiway join performs well for selective query patterns, but is outperformed by a pairwise-join baseline for other types of queries (due to the latter applying selectivity-based join reordering not available to SMJOIN).

5.4. A Multiway Join Algorithm for Basic Graph Patterns

We want to show the potential benefits of using a worst-case optimal join algorithm on SPARQL query performance. Surveying the state-of-the-art algorithms in the database literature (Veldhuizen, 2014; Ngo et al., 2014; Nguyen et al., 2015; Khamis et al., 2016; Kalinsky et al., 2017; Ngo, 2018), we opted to base our algorithm on Leapfrog Triejoin algorithm (LFTJ) (Veldhuizen, 2014), mainly because it is the most concise among all such algorithms (Ngo, 2018), and thus a good starting point for implementation within a SPARQL engine. We first present here a logical version of LFTJ that we call *Leapfrog Join* (LFJ), which includes only the core evaluation strategy on which LFTJ is based. LFJ can be divided into two main phases: *Leapfrog* and *variable elimination*. We begin by discussing both phases and give a running example of the algorithm. Later we propose a physical version of Leapfrog Join, designed to be easily integrated with existing SPARQL engines, mostly requiring adaptations at the index layer (see the discussion in Section 5.5).

5.4.1. Leapfrog

Unlike traditional join algorithms that evaluate triple pattern by triple pattern, Leapfrog Join rather proceeds by evaluating variable by variable. An important procedure in Leapfrog Join is to compute all *non-trivial outputs of a single variable*; more formally, given an RDF graph G, a basic graph pattern P and a variable $?\mathtt{x}$ in $\text{var}(P)$

we want to compute the following set:

$$\mathrm{LF}_G(P, ?\mathtt{x}) \;=\; \{\mu \mid \mathrm{dom}(\mu) = \{?\mathtt{x}\} \text{ and } [\![\mu(t)]\!]_G \neq \emptyset \text{ for all } t \in P\}\,.$$

In other words, we want to identify all single variable mappings μ such that, for every $t \in P$, the output of $\mu(t)$ over G is non-empty when $?\mathtt{x}$ is replaced by $\mu(?\mathtt{x})$. Intuitively, if $\mu \in \mathrm{LF}_G(P, ?\mathtt{x})$, then μ is a good candidate for a partial mapping that can be extended to form an output mapping in $[\![P]\!]_G$. Note also that if $?\mathtt{x}$ is the only variable used in P (i.e., $\mathrm{var}(P) = \{?\mathtt{x}\}$), then the set $\mathrm{LF}_G(P, ?\mathtt{x})$ is the same as computing the intersection of all sets $[\![t]\!]_G$. In Section 5.5, we will show how to implement this function for one or more variables by exploiting standard B+tree indexes while maintaining worst-case optimality.

Algorithm 4 Variable elimination for basic graph patterns.

Input: RDF graph G, BGP P, variable order $O_{var} = ?x_1 \ldots ?x_n$
Output: All mappings $[\![P]\!]_G$.

```
1:  function LFTJ-EVAL(G, P, O_var)
2:      μ_0 ← ∅
3:      for μ ∈ LF_G(μ_0(P), ?x_1) do
4:          μ_1 ← μ_0 ∪ μ
5:          for μ ∈ LF_G(μ_1(P), ?x_2) do
6:              μ_2 ← μ_1 ∪ μ
7:              ⋱
8:                  for μ ∈ LF_G(μ_{n-1}(P), ?x_n) do
9:                      Output μ_{n-1} ∪ μ          ▷ write to output and continue
10:                 end for
11:         end for
12:     end for
13: end function
```

Variable elimination: While the Leapfrog phase evaluates a single variable, the variable elimination phase evaluates multiple variables. Given a basic graph pattern P with $\mathrm{var}(P) = \{x_1, \ldots, x_n\}$, an RDF graph G, and a variable order $O_{\mathrm{var}} = ?\mathtt{x}_1, \ldots, ?\mathtt{x}_m$, Algorithm 4 shows the nested structure of the variable elimination procedure, which constitutes the overall Leapfrog Join process. The procedure iterates over each variable $?\mathtt{x}_i$ in order, extending the mapping μ_i with a mapping $\mu \in \mathrm{LF}_G(\mu_i(P), ?\mathtt{x}_{i+1})$. Variable $?\mathtt{x}_i$ is fixed by extending μ with μ_i (i.e. $\mu_{i+1} = \mu_i \cup \mu$; note that $\mu_i \sim \mu$,

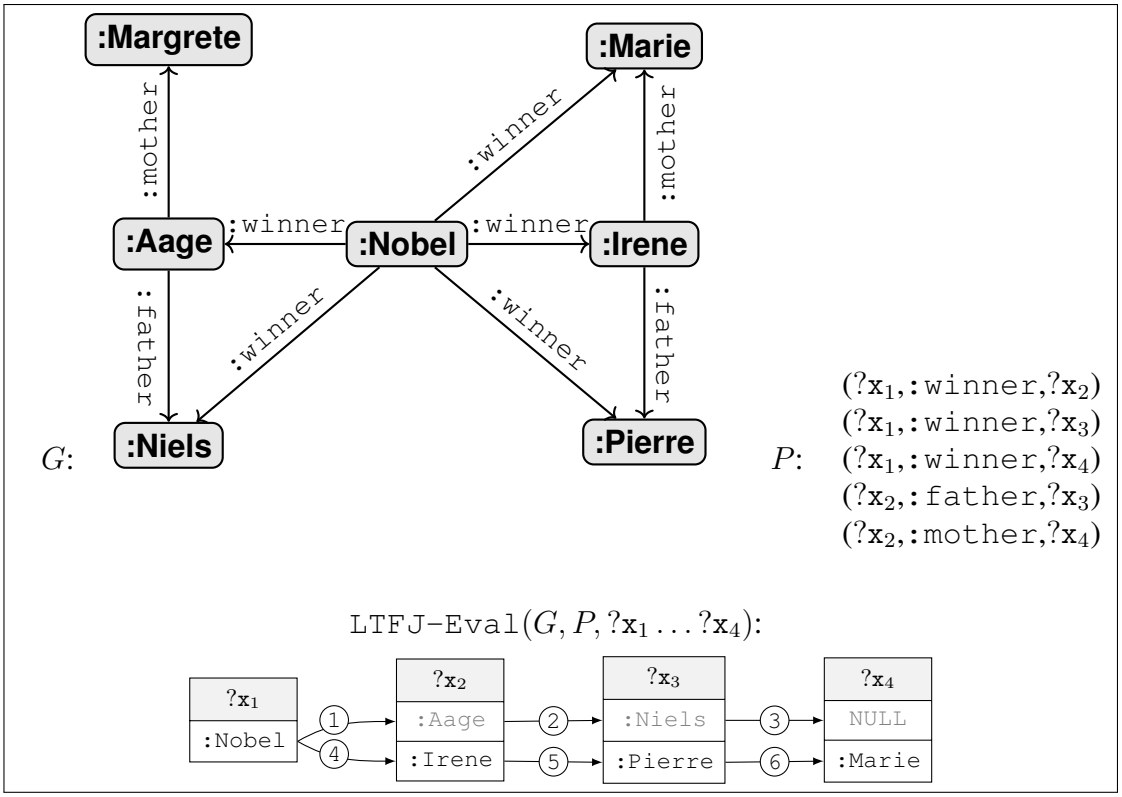


FIGURE 5.1. Example of Leapfrog join for evaluating a SPARQL basic graph pattern P.

so μ_{i+1} is also a mapping); in this way, variable $?\mathtt{x}_i$ is "eliminated" from P. The procedure moves on to eliminate the next variable $?\mathtt{x}_{i+1}$ analogously. After all variables $?\mathtt{x}_1, \ldots, ?\mathtt{x}_m$ are eliminated, the mapping $\mu_{m-1} \cup \mu$ is output, and the search for the next output is continued.

Figure 5.1 provides an example of variable elimination for a basic graph pattern over an RDF graph. We assume the order $?\mathtt{x}_1 \ldots ?\mathtt{x}_4$; how such an order is decided will be discussed later in Section 5.5.[3] Pairwise evaluation with this triple-pattern order would naively produce $5^3 = 125$ intermediary results containing the Cartesian product of all five winners of the Nobel prize (as would the multiway star-shaped join algorithm of Galkin et al. (Galkin et al., 2017)). On the other hand, under Leapfrog Join, variable elimination ensures that when, e.g., $?\mathtt{x}_2$ is evaluated, only those winners

[3]Such an order would be produced by SPARQL engines in practice if we had a graph with many `:father` and `:mother` relations, outnumbering `:winner` relations.

that have some father and some mother are considered. The lower graph then shows the recursion order producing the final result(s).

5.5. A Physical Operator for Leapfrog Join

We implement Leapfrog Join (LFJ) in Apache Jena TDB version 3.9.0, which implements nested-loop joins on top of B+tree indexes. We choose Jena as it is one of the most widely-deployed (fully) open source SPARQL engines; however the methods described can be generalized to other SPARQL engines. We now explain the main modifications required to support LFJ in Jena.

5.5.1. Indexes for LFJ

The first modification needed to run LFJ was to extend the index layer in Jena. Recall that a major phase in LFJ is to compute the set $\mathrm{LF}_G(P, ?\mathtt{x})$ given an RDF graph G, a basic graph pattern P and a variable $?\mathtt{x}$. The next result shows that Leapfrog Join is a worst-case optimal join algorithm whenever the computation of $\mathrm{LF}_G(P, ?\mathtt{x})$ is done in a reasonable time.

THEOREM 5.5.1. The Leapfrog Join is a worst-case optimal join algorithm if, for every RDF graph G, basic graph pattern P, and variable $?\mathtt{x}$, the computation of $\mathrm{LF}_G(P, ?\mathtt{x})$ is done in time at most:

$$O\big(\max(\min_{t\in P:?\mathtt{x}\in\mathrm{var}(t)} |\pi_{?\mathtt{x}}([\![t]\!]_G)|, 1) \cdot \log(|G|)\big)$$

where $\pi_{?\mathtt{x}}([\![t]\!]_G)$ is the projection of $[\![t]\!]_G$ over $?\mathtt{x}$.

PROOF. Fix a basic graph pattern P, an RDF graph G, and $?\mathtt{x}_1, \ldots, ?\mathtt{x}_n$ the chosen variable order. Further assume that the computation of $\mathrm{LF}_G(P', ?\mathtt{x})$ takes time at most $\min_{t\in P':?\mathtt{x}\in\mathrm{var}(t)} |\pi_{?\mathtt{x}}([\![t]\!]_G)| \cdot \log(|G|)$ for every basic graph pattern P' (for simplicity, we will omit the trivial empty case where there exists $t \in P'$ such that $?\mathtt{x} \in \mathrm{var}(t)$ and $|\pi_{?\mathtt{x}}([\![t]\!]_G)| = 0$ since the time taken will be simply $\log(|G|)$). Finally, for every RDF graph G' we will say that G' is of size less than G whenever $|[\![t]\!]_{G_i}| \leq |[\![t]\!]_G|$ for all $t \in P$ (recall that P is fixed).

The proof of Theorem 5.5.1 goes in two steps. First, we will bound the time of Leapfrog Join by bounding the time T_i of each for-loop $?\mathtt{x}_i$ of Algorithm 4. Then for each level $?\mathtt{x}_i$ we define a new RDF graph G_i of size less than G such that $T_i = |[\![P]\!]_{G_i}| \leq 2^{\text{MIN}(P,G)}$. The second step is to sum all the values T_i to obtain the total time of the algorithm. We show that the sum of all T_i is bounded by $n \cdot 2^{\text{MIN}(P,G)} \cdot \log(G)$, which in fact means that the algorithm is worst-case optimal in terms of the AGM bound.

Fix a variable $?\mathtt{x}_i$ and denote by $\bar{x}_{i-1} = ?\mathtt{x}_1, \ldots, ?\mathtt{x}_{i-1}$ the order of variables before $?\mathtt{x}_i$ (for the sake of simplification, in the sequel we consider $\bar{x}_i$ also as a set). We start by bounding the time of the for-loop in Algorithm 4 corresponding to $?\mathtt{x}_i$. For this, consider the following extension of LF_G over $\bar{x}_{i-1}$:

$$\text{LF}_G(P, \bar{x}_{i-1}) \;=\; \{\mu \mid \text{dom}(\mu) = \bar{x}_{i-1} \text{ and } [\![\mu(t)]\!]_G \neq \emptyset \text{ for all } t \in P\}$$

Clearly, the number of times that the for-loop of $?\mathtt{x}_i$ will be called is given by:

$$|\,\text{LF}_G(P, \bar{x}_{i-1})|$$

and then for each $\mu \in \text{LF}_G(P, \bar{x}_{i-1})$ the Leapfrog procedure $\text{LF}_G(\mu(P), ?\mathtt{x}_i)$ is called taking time at most $\min_{t \in \mu(P):?\mathtt{x}_i \in \text{var}(t)} |\pi_{?\mathtt{x}_i}([\![t]\!]_G)|$ (omitting the $\log(|G|)$ factor for the moment). If we call T_i the number of steps that Algorithm 4 spends in the for-loop of $?\mathtt{x}_i$, we have that:

$$T_i \;=\; \sum_{\mu \in \text{LF}_G(P, \bar{x}_{i-1})} \; \min_{t \in \mu(P):?\mathtt{x}_i \in \text{var}(t)} |\pi_{?\mathtt{x}_i}([\![t]\!]_G)|$$

One can easily check that the total time of Algorithm 4 is given by $(\sum_{i=1}^{n} T_i) \cdot \log(G)$. Therefore, if we bound T_i by the AGM bound of P and G, then the worst case optimality of Leapfrog Join will be proven (recall that our analysis is in data complexity, omitting factors that depend on the size of P).

To bound the size of T_i, we build an RDF graph G_i such that $T_i = |[\![P]\!]_{G_i}|$ and the size of G_i is less that the size of G. Let $\perp$ be a dummy value. To build G_i define the set of mappings U_i such that $\mu \in U_i$ if and only if there exists $\mu' \in \text{LF}_G(P, \bar{x}_{i-1})$ such that:

(i) $\mu(?\mathtt{x}) = \mu'(?\mathtt{x})$ for every $?\mathtt{x} \in \bar{x}_{i-1}$,

(ii) $1 \leq \mu(?\mathtt{x}_i) \leq \min_{t \in \mu'(P):?\mathtt{x}_i \in \text{var}(t)} |\pi_{?\mathtt{x}_i}([\![t]\!]_G)|$, and

(iii) $\mu(?\mathtt{x}) = \perp$ for every $?\mathtt{x} \in \{?\mathtt{x}_{i+1}, \ldots, ?\mathtt{x}_n\}$.

In other words, U_i contains all mappings built from mappings of $\text{LF}_G(P, \bar{x}_{i-1})$ and extended by assigning to $?\mathtt{x}_i$ any value less than the time for computing $\text{LF}_G(\mu'(P), ?\mathtt{x}_i)$. From U_i we can build the RDF graph G_i as follows:

$$G_i \ = \ \bigcup_{\mu \in U_i} \mu(P).$$

By construction, note that the size of G_i is less than the size of G. Furthermore, we have that $T_i = |[\![P]\!]_{G_i}|$. Indeed, for each $\mu' \in \text{LF}_G(P, \bar{x}_{i-1})$ we will have

$$\big(\min_{t \in \mu'(P):?\mathtt{x}_i \in \text{var}(t)} |\pi_{?\mathtt{x}_i}([\![t]\!]_G)| \big)$$

different mappings in $[\![P]\!]_{G_i}$ and vice versa.

To finish the proof, recall the linear program associated to P and G, and its minimum value $\text{MIN}(P, G)$. Consider also the same linear program but now for P and G_i. Given that G_i is of size less than G, then the minimization function associated to the linear program of P and G_i always satisfies:

$$\sum_{t \in P} x_t \cdot \log(|[\![t]\!]_{G_i}|) \leq \sum_{t \in P} x_t \cdot \log(|[\![t]\!]_G|).$$

Therefore, we can conclude that $\text{MIN}(P, G_i) \leq \text{MIN}(P, G)$ and thus:

$$T_i \ = \ |[\![P]\!]_{G_i}| \ \leq \ 2^{\text{MIN}(P,G_i)} \ \leq \ 2^{\text{MIN}(P,G)}$$

because $[\![P]\!]_{G_i}$ is a BGP query, and then the cardinality of $|[\![P]\!]_{G_i}|$ is bounded by its AGM bound $2^{\text{MIN}(P,G_i)}$ (with respect to the instance G_i). Since $G_i \leq G$, it follows that $2^{\text{MIN}(P,G_i)} \ \leq \ 2^{\text{MIN}(P,G)}$.

Given that each T_i is bounded by $2^{\text{MIN}(P,G)}$ we conclude that the overall time is bounded by $n \cdot 2^{\text{MIN}(P,G)} \cdot \log(G)$ and that Leapfrog Join is worst-case optimal. $\square$

Calculating $\mathrm{LF}_G(P, ?\mathtt{x})$ is the same as computing the intersection of all sets $[\![t]\!]_G$; hence, one can use any adaptive intersection algorithm over n sets (Demaine, López-Ortiz, & Munro, 2000; Barbay & Kenyon, 2002), which satisfies the time restriction of Theorem 5.5.1. In particular, our implementation of LFJ uses the intersection algorithm proposed by Veldhuizen (Veldhuizen, 2014).

The algorithm of adaptive intersection assumes that each set $\pi_{?\mathtt{x}}([\![t]\!]_G)$ can be navigated in increasing order. For this, we need an index I_G such that for every triple pattern t and every variable $?\mathtt{x}$ in t, it provides a seek method $\mathrm{I}_G[t, ?\mathtt{x}]\,.\,\mathtt{seek}(:\mathtt{a})$ that outputs the least $:\mathtt{b}$ such that $:\mathtt{b} \geq :\mathtt{a}$ and $[\![\mu(t)]\!]_G \neq \emptyset$ for $\mu = \{?\mathtt{x} \to :\mathtt{b}\}$, or $\mathtt{NULL}$ if no such $:\mathtt{b}$ exists; in other words, the seek method jumps to the next non-trivial output for $?\mathtt{x}$ in the order. To satisfy the bound of Theorem 5.5.1, the seek method is required to take time logarithmic in the size of G. Although the original LFTJ algorithm proposes to use tries for I_G, such a seek method can be supported using B+Trees adding all six orders over s, p and o. Hence to Jena's three default orders **spo**, **pos**, and **osp**, our implementation adds three more orders: **sop**, **pso**, and **ops**. This roughly doubles the size of the on-disk index and the number of update operations required to add/remove triples, but (as shown later) offers gains in query performance with LFJ.[4]

Each index order is assigned a B+tree, where the seek method could then be implemented by traversing the B+tree top-down from root to leaf in the standard way. However, given that the seek method requests values in sequential order, we use a stack to store the current node in the iteration, its leaf, and its parents; when the next value is requested, we can read the next value in the order from the leaf or, starting from there, search the B+tree upwards and then back down in case that the next value is in another leaf. This bottom-up seek method offers constant amortized time when only one variable is unbound (Veldhuizen, 2014), logarithmic time when two variables are unbound, and is more efficient in practice.

[4]We currently consider querying over a single RDF graph; if we were to consider a complete index on quads in order to support named graphs, the number of required indexes would jump to 24. In such a case, however, practical steps can be taken to reduce the number of indexes where, for example, some such orders will be rarely accessed by real-world queries and can thus be removed.

5.5.2. LFJ operator

We add a new LFJ join operator to Jena that takes a basic graph pattern and evaluates it using our implementation of LFJ (per Algorithm 4). Note that the original LFTJ algorithm applies some restrictions: (1) each relation symbol must appear only once, (2) the order of attributes of the relations (triple patterns in our case) must follow the global attribute order, (3) constants cannot appear within the join query and (4) each attribute can appear at most once in each relation (triple pattern in our case). The first restriction does not apply for our implementation. Restrictions (2) and (3) are not required to maintain worst-case optimality and are addressed by our indexes. The case of variables occurring twice in a triple pattern requires some extra care, but can be addressed with special indexes for triples repeating the same term in the given positions (which are typically uncommon in RDF data), or using a fresh variable and applying a low-level filter/intersection; we omit these details for brevity.

5.5.3. Variable order

The performance of LFJ is dependent on the chosen variable order (Abo Khamis, Ngo, & Rudra, 2016); referring back to Figure 5.1, for example, a more efficient order would be to swap $?\mathtt{x}_2$ and $?\mathtt{x}_4$, which would allow for more quickly rejecting the incomplete mapping involving `:Aage` in G. In principle, the goal of finding a variable ordering is similar to that for ordering triple patterns: in both cases, we wish to evaluate highly-selective triple patterns/variables that help to filter mappings early on. Along these lines, while specialised variable orderings have been proposed for worst-case optimal join algorithms (Abo Khamis et al., 2016), we propose a solution based on Jena's existing triple ordering; this has the additional advantage of making experiments between the baseline version of Jena and Jena with LFJ more comparable.

Given a triple-pattern order O_{trip} returned by Jena, we first choose join variables in order of appearance, and then select lonely variables in order of appearance; for example, if Jena gives $O_{\text{trip}} = (\texttt{?z}, \texttt{:p3}, \texttt{?u}), (\texttt{?x}, \texttt{:p2}, \texttt{?z}), (\texttt{?x}, \texttt{:p1}, \texttt{?y})$, we will choose the variable order $O_{\text{var}} = \texttt{?z}, \texttt{?x}, \texttt{?u}, \texttt{?y}$ since `?z` is the first join variable that appears in O_{trip}, and `?x` is the second join variable that appears in O_{trip}; given that `?u`

and ?y are lonely variables (appearing in one triple pattern), they come after the join variables, again based on order of appearance. In fact, as we now discuss, the order of lonely variables will not affect performance.

5.5.4. Enumerating mappings

Early experiments comparing Jena with and without LFJ found that the performance of the former was sometimes orders of magnitude *worse* than the latter. We identified the issue as relating to how lonely variables are handled. To illustrate this issue, consider a graph pattern P' containing only the first three triple patterns of P in Figure 5.1 such that $?\mathtt{x}_2$, $?\mathtt{x}_3$ and $?\mathtt{x}_4$ are lonely variables. Applying the procedure of Algorithm 4, after assigning $?\mathtt{x}_1 \rightarrow$:Nobel, we still require $5 \times 5 \times 5$ steps through the recursion, repetitively evaluating the same partially-bound triple patterns. This final recursion is unnecessary: since lonely variables are evaluated last, we know that the final mappings must be extended by the Cartesian product of the non-trivial outputs of the remaining lonely variables. To address this, assume a variable order $O_{\text{var}} = ?\mathtt{x}_1, \ldots, ?\mathtt{x}_m, ?\mathtt{x}_{m+1}, \ldots ?\mathtt{x}_n$ where $?\mathtt{x}_1, \ldots, ?\mathtt{x}_m$ are join variables and $?\mathtt{x}_{m+1}, \ldots, ?\mathtt{x}_n$ are lonely variables. Assume also that $t_1, \ldots, t_k$ are the triple patterns where $?\mathtt{x}_{m+1}, \ldots, ?\mathtt{x}_n$ are mentioned (each such triple pattern may mention one or more lonely variables). We eliminate $?\mathtt{x}_1, \ldots, ?\mathtt{x}_m$ per Algorithm 4, and for each partial solution μ_m generated, we compute the Cartesian product $\mu_m \times [\![\mu_m(t_1)]\!]_G \times \ldots \times [\![\mu_m(t_k)]\!]_G$, requiring k (note that $k < n - m$) additional calls to $[\![\mu_m(\cdot)]\!]_G$ for each μ_m (rather than having to call LF a total of $1 + \sum_{i=m}^{n-2} \prod_{j=m}^{i} |\,\mathrm{LF}_G(\mu_m(P), ?\mathtt{x}_{j+1})|$ times for each μ_m).

5.6. Experiments and Results

To understand how our implementation works we compared the performance of query evaluation for Apache Jena (TDB) v.3.9.0 with LFJ, Apache Jena v.3.9.0 without LFJ, Virtuoso v.OS-7.2.7 (Virtuoso, 2015) (one of the most deployed engines in practice (Aranda, Hogan, Umbrich, & Vandenbussche, 2013)), and Blazegraph v.2.1.4 (Thompson et al., 2014) (used by the Wikidata Query Service (Malyshev et al., 2018)). We run three sets of experiments using the Berlin SPARQL Benchmark (Bizer

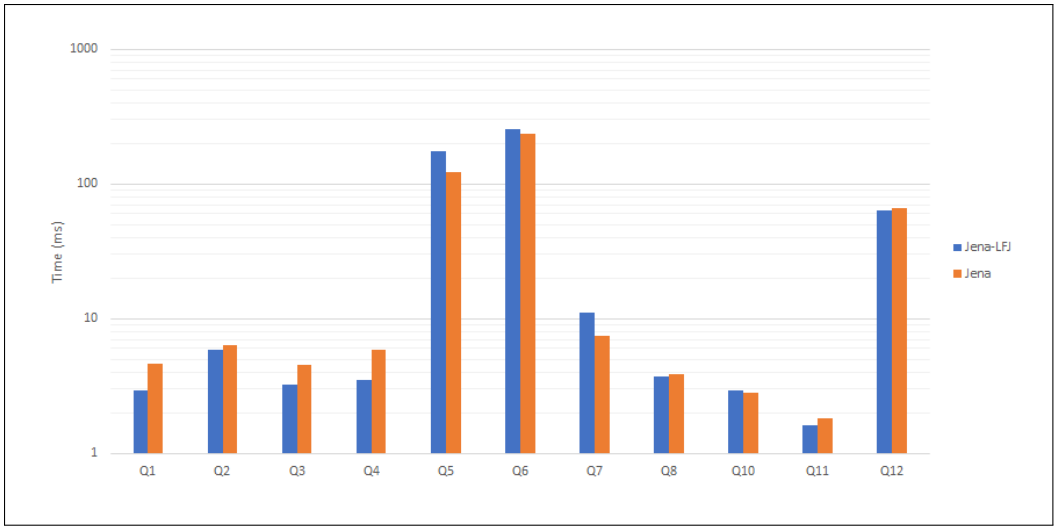


FIGURE 5.2. Plot of runtimes for queries of the Berlin Benchmark with log y-axis.

& Schultz, 2009), the WatDiv Benchmark (Aluç et al., 2014), and a novel Wikidata Benchmark with complex graph patterns that we propose. We run all experiments on a single machine with Ubuntu 16.04.5, Intel Xeon CPU E5-2609 v4@1.70GHz, Seagate 1TB Enterprise Capacity 2.5-Inch HDD, and 32GB RAM. Code and configurations can be found online for reproducibility purposes (*Additional material*, n.d.).

5.6.1. Experiments on existing datasets

Berlin Benchmark. We first ran experiments over the Berlin SPARQL Benchmark (BSBM) (Bizer & Schultz, 2009), comparing query runtimes for Jena with (denoted Jena-LFJ) and without (denoted Jena) the LFJ modifications. We run the Explore Use-Case of BSBM, consisting of 12 queries using a mix of SPARQL 1.0 features, including optional, union, filter, graph, etc. In Figure 5.2 we show the average time of each query in logarithmic scale. These experiments were done by running 10,500 queries; we found that on average each query took 49.3 ms for Jena-LFJ and 41.6 ms for Jena. We conclude that the BSBM results show no clear trend to suggest that one implementation outperforms the other. BSBM queries do not contain large intermediary results and, thus, Jena-LFJ offers no improvement. Furthermore, given that BSBM queries contain other features of SPARQL, the baseline of Jena can use optimizations for other operators not currently available for Jena-LFJ (in particular, pushing range filters, which appear in many BSBM queries).

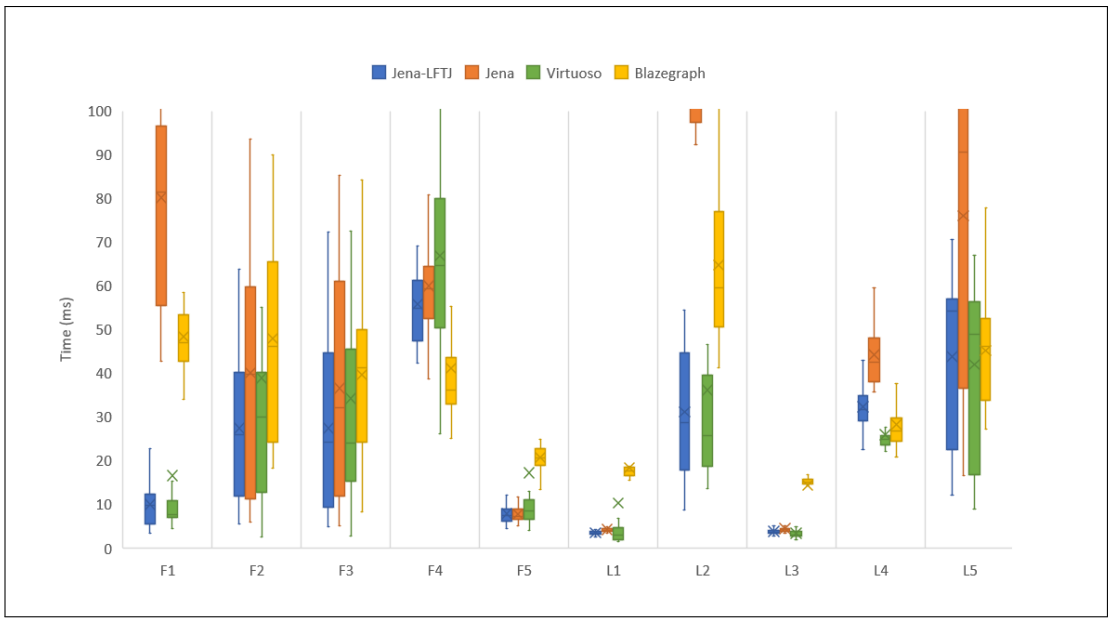


FIGURE 5.3. Box plots of runtimes for queries L and F of the WatDiv Benchmark.

WatDiv Benchmark. After reviewing the BSBM results, we still foresaw the need to run experiments on queries with more complex and diverse basic graph patterns. We chose the WatDiv benchmark (Aluç et al., 2014) which is designed for this purpose. We generate 50 queries for each of the 20 abstract patterns proposed in the benchmark. Executing the $50 \times 20 = 1000$ query instances and taking the average over all of them, Virtuoso takes 64 s, Jena-LFJ takes 77 s, Blazegraph takes 99 s, and Jena takes 198 s. Box-plots of runtimes for each specific query pattern are shown in Figures 5.3 and 5.4. Unlike in the BSBM experiments, here Jena-LFJ is at least twice as fast as Jena in terms of the overall query runtime and it also outperforms Blazegraph. Indeed, these plots suggest that the running time of Jena-LFJ is much more stable than other implementations; the interquartile difference is at most 40 ms. Since this benchmark is oriented towards testing basic graph patterns, we can see here that our implementation is competitive with respect to the other engines, being slightly outperformed by Virtuoso. Despite this analysis, the runtimes of these queries are still in the order of less than 100 ms, making it difficult to claim that Jena-LFJ or Virtuoso is the best approach.

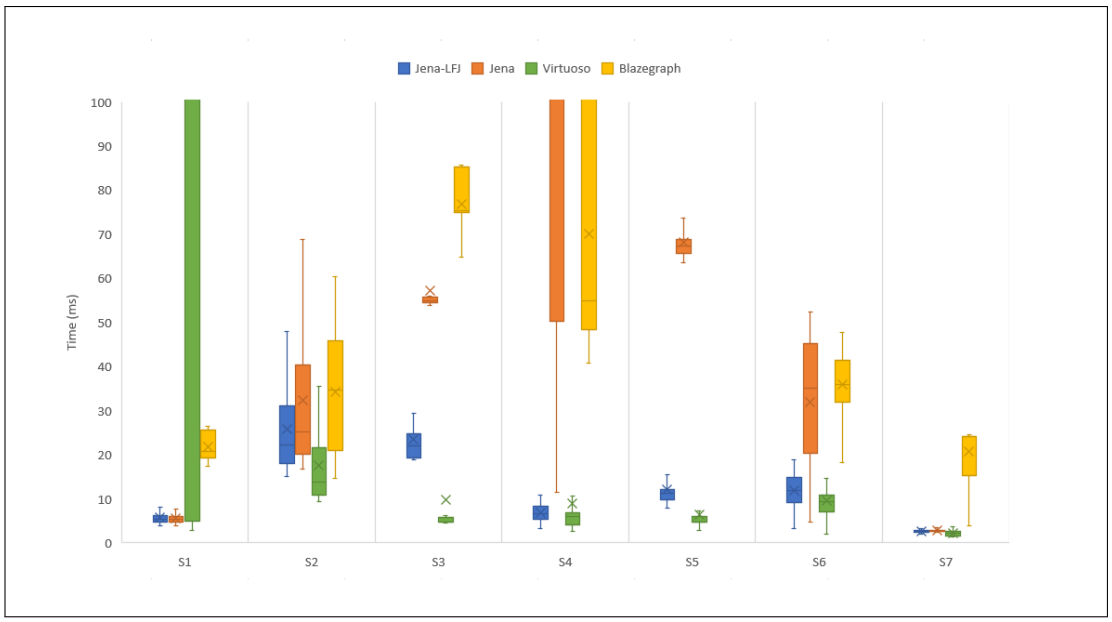


FIGURE 5.4. Box plots of runtimes for queries S of the WatDiv Benchmark.

5.6.2. The Wikidata Benchmark

SPARQL benchmarks usually focus on cover all features of SPARQL query language (Harris & Seaborne, 2013) or the Web Ontology Language (OWL) (Bizer & Schultz, 2009; Schmidt, Hornung, Lausen, & Pinkel, 2009; Guo, Pan, & Heflin, 2005). Indeed, although queries in benchmarks do contain complex BGPs, they are not designed to test the evaluation performance of them. Also, SPARQL benchmarks usually contain a fix set of queries designed to cover practical scenarios of SPARQL engines. These approaches fulfill to be relevant, portable, scalable, and understandable as proposed in (Gray, 1992), however, they miss to provide a diverse and complete set of BGPs to prove statistically which evaluation approach is better.

Though WatDiv is oriented to evaluate BGPs and it contains more complex graph patterns than Berlin, it does not contain (for example) graph patterns with cycles; furthermore, both benchmarks are based on synthetic data with relatively simple schemata (e.g., BSBM and WatDiv have 30 and 85 distinct predicates, respectively). Furthermore, a query engine can easily be tuned in such a way that it has an excellent performance over a fix set of queries, but terrible performance in general.

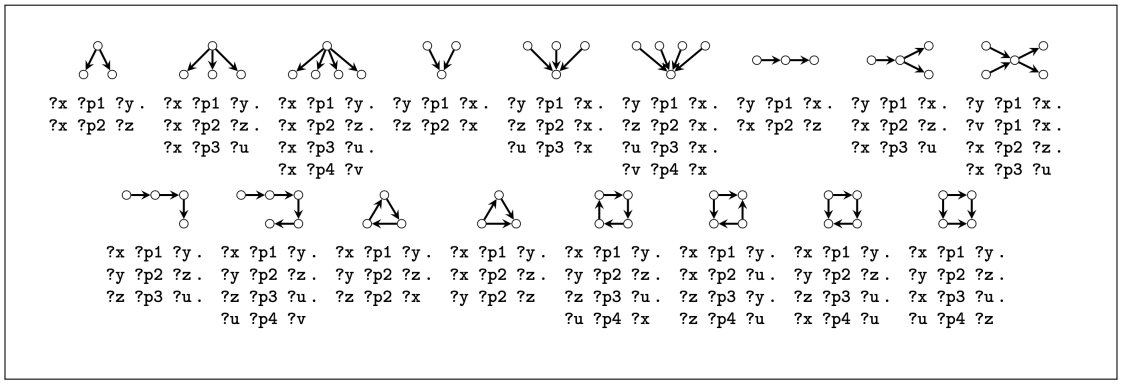


FIGURE 5.5. Basic graph patters and their associated diagram.

Given the above drawbacks of SPARQL benchmarks, we proposed to compare the performance of Leapfrog Triejoin against standard join evaluation by generating queries from Wikidata. Although the knowledge-base can be encoded in several different ways in RDF (Hernández, Hogan, & Krötzsch, 2015; Hernández, Hogan, Riveros, Rojas, & Zerega, 2016), here we only consider statements in Wikidata from the primary relation, where each fact is modeled as a RDF triple with subject, predicate, and object. For example, to state in Wikidata that "Abraham Lincoln was the President of United State of America" we will have a triple (`:q91 :p39 :q11696`) where `:q91` is the identifier (i.e. id) of the subject "Abraham Lincoln", `:p39` is the id of the predicate "position held", and `:q11696` is the id of the object "President of the United States of America". Note that in Wikidata a subject in some fact (i.e. triple) can be the object in another fact and vice versa.

One of the main attributes of Wikidata is that it can be seen as a source of several binary relations with real and heterogeneous data. Currently, Wikidata contains more than 5 thousands predicates (e.g. `:p39` for "position held") and each predicate `:px` defines a fresh binary relation of the form $(s, \texttt{:px}, o)$. From these 5 thousands different predicates, there are predicates than have more than 166 millions triples (e.g. `:p2860`) and predicates than have only a few triples. Moreover, each predicate follows a different distribution and connect with other predicates in an unpredictable way. Therefore, we propose to use Wikidata to randomly generate different BGPs over the binary relations in it, and use these BGPs to test the performance of join evaluation

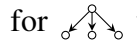
strategies. The advantage of this approach is that (1) we base our test over real and heterogeneous data, (2) each query engine cannot predict the set of predicates to be used, and (3) we can randomly generate a reasonable number of queries without having any bias on the decision.

Query structure. Although our benchmark is based over randomly generated BGPs over Wikidata, the structure of these BGPs is taken from a fix set of what we called *abstract BGPs*. Each abstract BGP is a BGP with variables $?\texttt{x}, ?\texttt{y}, \ldots$ and predicate variables $?\texttt{p1}, ?\texttt{p2}, \ldots$. Then for each abstract BGP A with predicate variables $?\texttt{p1}, \ldots, ?\texttt{pk}$, we will generate predicates $\texttt{:p1}, \ldots, \texttt{:pk}$ from Wikidata and create a BGP $\mu(A)$ where $\mu = \{?\texttt{p1} \to \texttt{:p1}, \ldots, ?\texttt{pk} \to \texttt{:pk}\}$.

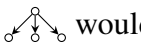
In Figure 5.5, we present 17 different abstract BGPs with the corresponding diagram representing its structure. The first three abstract patterns  ,  , and  represents trees joint over a single join variable rooted on the subject. The next three abstract patterns  ,  ,  are again join trees over a single join variable but now rooted on the object. Then the next BGPs  , , and  are the last patterns that we consider over a single variable but now the join variable combines subjects with objects. In the second line of Figure 5.5, we show BGPs structures with joins over two or more variables. The BGPs  and (and also ) represents path queries of length three and four, respectively. Then we consider join queries like triangles and squares, like the triangles  and  , and the squares  , , and . Note that any triangle or square BGP is isomorphic to one of these two or four cases, respectively.

Although it is possible to consider more different abstract BGPs than the ones in Figure 5.5, we claim that the abstract patterns above are enough to separate the main features of a join algorithm in order to give empirical evidence of its performance. Furthermore, we could have considered abstract BGPs with more triples; however, we will see that the case with three and four triples its enough to see the difference of performance between Leapfrog Triejoin and standard join evaluation.

Random generation of BGPs. Now that the pattern of abstract BGPs is fixed, we focus on the random generation of real BGPs. For each abstract BGP we want to randomly generate with uniform distribution a set of BGPs that uses the same abstract structure, but where predicate variables are instantiated with different and diverse predicates from Wikidata. Given that Wikidata contains more than 5 thousands properties, we have a big pool of predicates to instantiate each abstract BGP in Figure 5.5. From now, let G be the Wikidata graph.

Given a particular abstract BGP, a first naive approach is to randomly pick the predicates $?\texttt{p1}, ?\texttt{p2}, \ldots$ from G. For example, for  we can randomly choose `?p1 = :p234`, `?p2 = :p556`, and `?p3 = :p1134` with uniform distribution, and produce the BGP:

$$P := \texttt{?x :p234 ?y . ?x :p556 ?z . ?x :p1134 ?z}$$

We can repeat this random process until we collect enough different queries. The problem with this approach is that it is highly probable that each generated query will have empty output. Although it could sound reasonable to compare join evaluation strategies against BGPs with empty outputs, it is highly probable that a complex query, can be answered by just checking that the join of a pair of triples is empty. For example, it can happen with high probability that P is empty just because $[\![\texttt{?x :p234 ?y . ?x :p556 ?z}]\!]_G = \emptyset$. Therefore, the evaluation of P representing  would be misleading: we were checking how fast is the query in a binary join rather than in a ternary join.

We want to compare join evaluation strategies with a diverse and meaningful set of queries over Wikidata. Thus another strategy to generate instances of an abstract BGP P is to randomly pick `:p1`, ..., `:pk`, replace these predicates in P to create a BGP P^*, and then check whether $[\![P^*]\!]_G \neq \emptyset$. If the output is non-empty, we found our random instance of P and, if not, we repeat the process again. This strategy will generate random instances of P where each predicate has the same possibilities to appear in P^*, as long as it produces a non-empty BGP. Therefore, we will get a diverse and meaningful set of instances of P over Wikidata. The problem with this approach

is that it is highly probable that $[\![P^*]\!]_G = \emptyset$ and, therefore, it will take ages to generate an instance such that $[\![P^*]\!]_G \neq \emptyset$.

For this reason, we use the following equivalent procedure that converges much faster and generates a diverse and meaningful set of BGPs in Wikidata. Given an abstract BGP P, we generate at random a predicate $:\texttt{p1}$ from the predicates in Wikidata. Then we create the BGP $P_1 = \mu_1(P)$ with $\mu_1 = \{?\texttt{p1} \rightarrow :\texttt{p1}\}$ and check whether $[\![P_1]\!]_G \neq \emptyset$. If this is the case, we randomly generate another predicate $:\texttt{p2} \in \{\mu(?\texttt{p2}) \mid \mu \in [\![P_1]\!]_G\}$ to instantiate $P_2 = \mu_2(P_1)$ with $\mu_2 = \{?\texttt{p2} \rightarrow :\texttt{p2}\}$. We continue this way until all variables predicates are assigned, in which case we have found a randomly generated BGP that has non-empty output on Wikidata and, furthermore, each predicate was selected uniformly at random. Otherwise, if at any point $[\![P_i]\!]_G = \emptyset$, we stop the process and start again. Note that each instance generated with this process will use heavy predicates (predicates with many triples) but also light predicates (predicates with few triples) with the same uniform probability.

With the random process explained above, we were able to generate 50 diverse instances for each abstract BGP, and test Leapfrog Triejoin with the standard join evaluation. The randomly selected BGPs and the implementation of the random process can be found in (*Additional material*, n.d.). It is important to remark that an advantage of the "Wikidata Benchmark" compared to previous benchmarks relies on the random generation of meaningful BGPs from real and heterogeneous data.

Results with single join variable: We first present results for queries with a single join variable (the top row of Figure 5.5), analyzing the performance of the Leapfrog procedure of LFJ. Executing the $50 \times 9 = 450$ query instances, in terms of overall query runtime across all patterns, Jena-LFJ takes 4.0 s, Jena takes 14.0 s, Blazegraph takes 27.9 s, and Virtuoso takes 64.8 s. Figure 5.6 then shows the detailed results per query pattern, where we focus the y-axis in on the range of 0–300 ms for clarity. Here we see that Jena-LFJ is at least twice as fast as Jena in terms of median or mean times, and can be 10–20 times faster than the slowest engine for some queries. The most notable speedup occurs when join variables appear in the object position, which may lead to many intermediate results when a node with high in-degree (e.g., a country) is

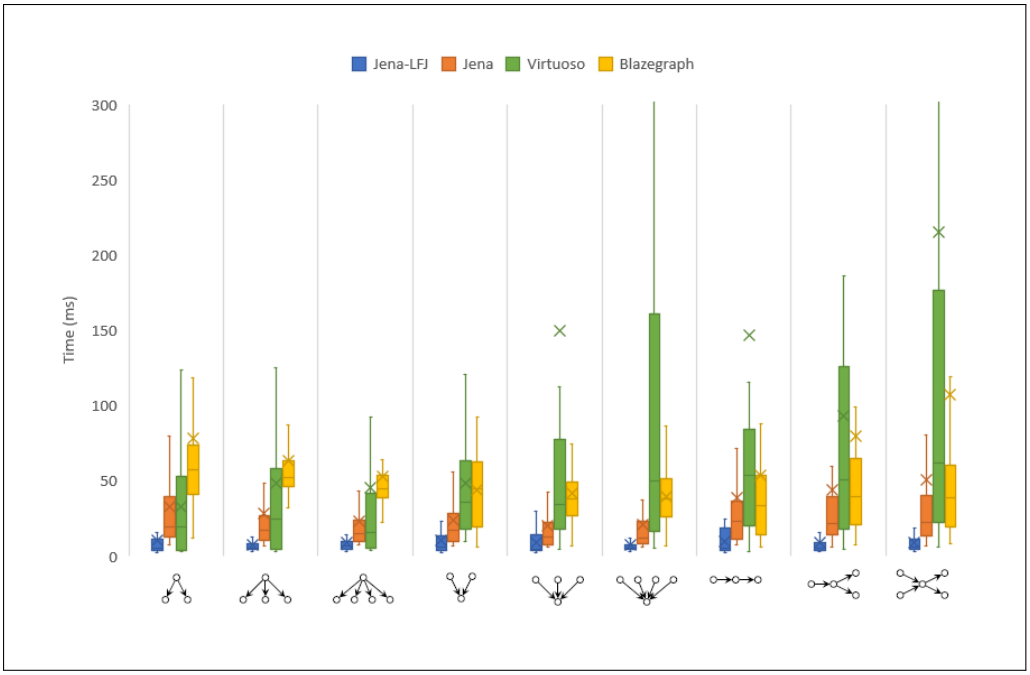


FIGURE 5.6. Box plots of runtimes for queries with a single join variable.

involved; in such cases, LFJ performs better than other engines. One might consider that this speedup may be attributable to the lack of the three additional orders of **s**, **p** and **o** in the other engines. However, in the case of the best gains – i.e., joins in the object position – Jena-LFJ is using the **pos** index, which is already included in Jena; more generally, Jena uses index nested loop joins, which cannot benefit from further index orders when evaluating BGPs/equijoins.

We further observe that the runtimes for Jena-LFJ are more stable, with the maximum runtime never exceeding 55 ms; furthermore, within the 50 queries of each abstract pattern, the standard deviation in runtimes for Jena-LFJ is consistently around 9 ms, while Jena's standard deviation is always over 20 ms, and that of Virtuoso and Blazegraph is even higher, sometimes over 100 ms.

Results with multiple join variables: We now present results for queries with multiple join variables (the bottom row of Figure 5.5). Given that the previous experiments test the performance of Leapfrog for intersecting results for join variables in up to four patterns, our focus now is on the performance of variable elimination. We thus select abstract graph patterns where each variable appears in at most two triple patterns; such

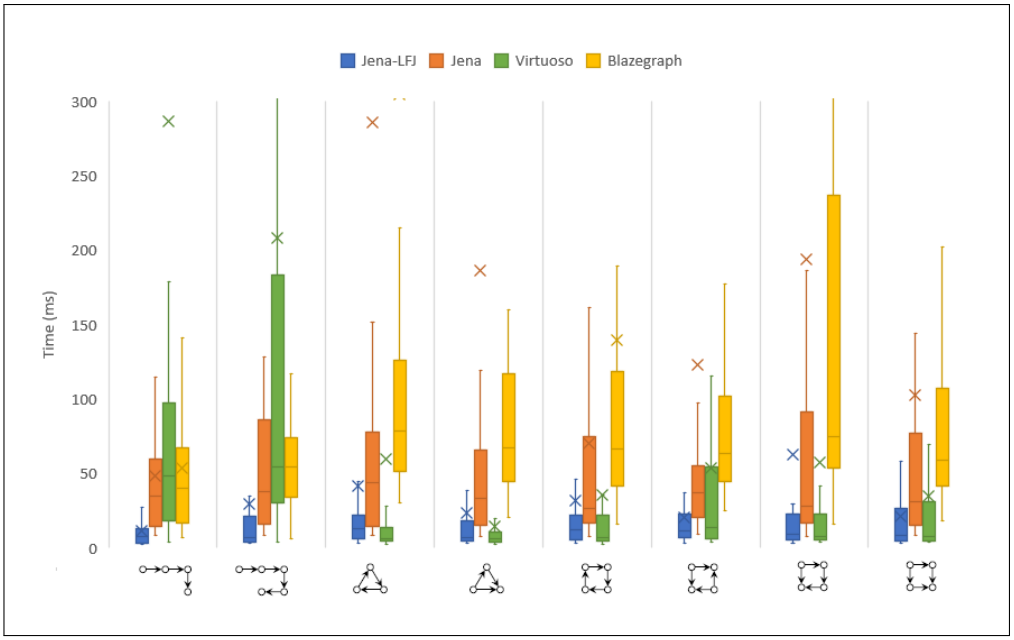


FIGURE 5.7. Box plots of runtimes for queries with multiple join variables.

queries put as much emphasis as possible on the performance of the variable elimination phase versus the Leapfrog phase tested previously. Executing the $50 \times 8 = 400$ query instances, in terms of the overall query runtime across all patterns, Jena-LFJ takes 12 s, Virtuoso takes 37 s, Jena takes 112 s, and Blazegraph takes 35 s. Figure 5.7 again shows the detailed results focusing on the same y-axis range for clarity. We again see that Jena-LFJ generally exhibits the most stable runtimes, clearly outperforming Jena and Blazegraph for all patterns and Virtuoso for the first two patterns. Comparing Jena-LFJ and Virtuoso for the latter six patterns (those with cycles), Virtuoso is competitive with and sometimes even outperforms Jena-LFJ; analysing further, we found that Virtuoso often chooses a better execution order than Jena(-LFJ), where manually optimizing the variable order in Jena-LFJ for such cases results in much better performance than Virtuoso; this suggests that the variable ordering of Jena-LFJ could be improved. Even with the current variable ordering of Jena-LFJ, however, the clear gains in the first two patterns vs. Virtuoso outweigh slight gains by Virtuoso in some of the latter six patterns, as evidenced by the total runtimes mentioned previously (12 s vs. 37 s).

As we see, to implement worst-case optimal join algorithms within RDF databases is worth, since the approach can improve the time of evaluation of basic graph patterns. To the best of our knowledge, here we present the first work that looks at the benefits of this kind of algorithms for RDF databases. We believe that this techniques can be an alternative to be adopted by SPARQL engines in the near future. However, SPARQL is a query language that works not only in local environments but also works well within the web context. Then, a natural question is whether this approach can be used to integrate web data. Hence, in the next chapter, we discuss the idea of integrating web data that currently is not available for the Semantic Web Services using algorithms based on worst-case optimal joins.

Chapter 6. QUERYING APIS WITH SPARQL

The Semantic Web provides a platform for publishing data on the Web via the Resource Description Framework (RDF). Having a common format for data dissemination allows for applications of increasing complexity since it enables them to access data obtained from different sources, or describing different entities. The most common way of accessing this information is through SPARQL endpoints; SPARQL is the standard language for accessing data on the Semantic Web (Harris & Seaborne, 2013), and a SPARQL endpoint is a simple interface where users can obtain the RDF data available on the server by executing a SPARQL query.

As we mention at the introduction, in the Web context it is rarely the case that one can obtain all the needed information from a single data source, and therefore it is necessary to integrate data from multiple servers or endpoints. In order to address this, a specific operator that allows parts of the query to access different SPARQL endpoints, called SERVICE, was included into the latest version of the language (Prud'hommeaux & Buil-Aranda, 2013).

However, the majority of the data available on the Web today is still not published as RDF, which makes it difficult to connect it to Semantic Web services. A huge amount of this data is made available through Web APIs which use a variety of different formats to provide data to the users.

It is therefore important to make all of this data available to Semantic Web technologies, in order to create a truly connected Web. One way of achieving this is to extend the SERVICE operator of SPARQL with the ability to connect to Web APIs in the same way as it connects to other SPARQL endpoints. In this chapter we make a first step in this direction by extending SERVICE with the option to connect to JSON APIs and incorporate their data into SPARQL query answers. We picked JSON because it is currently one of the most popular data formats used in Web APIs, but the results presented in the chapter can easily be extended to any API format. By allowing SPARQL to connect to an API we can extend the query answer with data obtained from a Web service, in real time and without any setup.

6.1. An use-case of the extended SERVICE

Use cases for such an extension are numerous and can be particularly practical when the data obtained from the API changes very often (such as weather conditions, state of the traffic, etc.). To illustrate this let us consider the following example.

EXAMPLE 6.1.1. We find ourselves in Scotland in order to do some hiking. We obtain a list of all Scottish mountains using the Wikidata SPARQL endpoint, but we would prefer to hike in a place that is sunny. This information is not in Wikidata, but is available through a weather service API called `weather.api`. This API implements HTTP requests, so for example to retrieve the weather on Ben Nevis, the highest mountain in the UK, we can issue a GET request with the IRI:

```
http://weather.api/request?q=Ben_Nevis
```

The API responds with a JSON document containing weather information, say of the form

```
{"timestamp": "24/10/2017 11:59:07",
 "temperature": 3, "description": "clear sky",
 "coord": {"lat": 56.79, "long": -5.02}}
```

Therefore, to obtain all Scottish mountains with a favourable weather all we need to do is call the API for each mountain on our list, keeping only those records where the weather condition is `"clear sky"`. One can do this manually, but this quickly becomes cumbersome, particularly when the number of API calls is large. Instead, we propose to extend the functionality of SPARQL SERVICE, allowing it to communicate with JSON APIs such as the weather service above. For our example we can use the following (extended) query:

```
1 SELECT ?x ?l WHERE {
2     ?x wdt:instanceOf wd:mountain .
3     ?x wdt:locatedIn wd:Scotland .
4     ?x rdfs:label ?l .
5     SERVICE <http://weather.api/request?q={?l}>{
```

	Yelp!	Twitter	Open Weather	Wikipedia	StackOverflow	All
min	0.4	0.4	0.4	0.8	0.3	0.3
max	1.3	0.8	1.4	1.3	1.5	1.5
avg	1.1	0.5	0.6	1.0	0.6	0.76

TABLE 6.1. Min, max, and average response time (in seconds) of popular Web APIs based on ten typical calls they support.

```
6         (["description"]) AS (?d)
7       }
8       FILTER (?d = "clear sky")
9   }
```

The first part of our query is meant to retrieve the IRI and label of the mountain in Wikidata. The extended SERVICE operator then takes the (instantiated) URI template where the variable `?l` is replaced with the label of the mountain, and upon executing the API call processes the received JSON document using an expression `["description"]`, which extracts from this document the value under the key `description`, and binds it to the variable $?d$. Finally, we filter out those locations with undesirable weather conditions. □

With the ability of querying endpoints and APIs in real time we face an even more challenging task: How do we evaluate such queries? Connecting to APIs poses an interesting new problem from a database perspective, as the bottleneck shifts from disk access to the amount of API calls. For example, when evaluating the query in Example 6.1.1, about 80% of the time is spent in API calls. This is mostly because HTTP requests are slower that disk access, something we cannot control. To gauge the time taken for APIs to respond to a `GET` request we did a quick study of five popular Web APIs. Based on the API documentation we created ten different calls for each API, and ran each call five times, recording the average value. The results presented in Table 6.1 show us the minimum, the maximum, and the average time over our calls for each API. The least amount of time needed was 0.3 seconds, which is already quite substantial when processing a query that makes a huge amount of API calls, and it can range up to more than a second.

Hence, to evaluate these queries efficiently we need to understand how to produce a query plan for them that minimizes the number of calls to the API. Apart from formally defining the syntax and the semantics of the extended SERVICE operator, finding algorithms that minimize the number of API calls is the main question studied in this chapter.

6.2. SERVICE and JSON

As expected, we keep the definition of RDF graphs and we have used so far, but we have to add a the SERVICE operator to our definition of SPARQL. This operator is used to integrate RDF graphs from different endpoints.

If $a \in \mathbf{I}$, then by $ep(a)$ we denote the graph G that is served by the SPARQL endpoint reached by a. In case that a is not a valid endpoint (i.e. an RDF dataset), we define $ep(a) = \emptyset$. The set $[\![P]\!]_G$ for a pattern $P = P_1$. SERVICE a $\{P_2\}$ is defined as:

$$[\![P]\!]_G = \{ \mu \mid \mu = \mu_1 \cup \mu_2, \text{where } \mu_1 \in [\![P_1]\!]_G, \mu_2 \in [\![P_2]\!]_{ep(a)}, \text{ and } \mu_1 \sim \mu_2\}$$

Note that we are not considering the case when $a \in \mathbf{V}$. However, a complete definition for the SERVICE operator has been done by Aranda et al. (Aranda, Arenas, et al., 2013; Aranda et al., 2011).

JSON. The JSON format (Bray, 2014) defines the following types of values. First, `true`, `false` and `null` are JSON values. Any decimal number (e.g. 3.14, 23) is also a JSON value, called a *number*. Furthermore, if `s` is a string of unicode characters then `"s"` is a JSON value, called a *string value*. Next, if $v_1, \ldots, v_n$ are JSON values and $s_1, \ldots, s_n$ are pairwise distinct string values, then $o = \{s_1 : v_1, \ldots, s_n : v_n\}$ is a JSON value, called an *object*. In this case, each $s_i : v_i$ is called a key-value pair of o. Finally, if $v_1, \ldots, v_n$ are JSON values then $a = [v_1, \ldots, v_n]$ is a JSON value called an *array*. In this case $v_1, \ldots, v_n$ are called the *elements of* a. Numeric values, strings and the boolean values `true`, and `false` are called *basic JSON values*.

To navigate through JSON documents we use *JSON navigation instructions*. For an object J, the navigation instruction $J[\text{“key”}]$ returns the value of a pair in J whose key is the string “key”, and for an array J, the navigation instruction $J[n]$, for a natural number n, returns the n-th element of J. These instructions can be stacked to retrieve values of nested JSON documents; e.g. $J[\text{”key1”}][7]$, will first fetch the value of the key “key1” (if J is an object), and then, assuming that this value is an array, return the seventh element of the array. If the JSON does not have the corresponding key or array element, the navigation expression returns an error.

6.3. Enabling SPARQL to make JSON calls

In this section we define the syntax and the semantics of the overloaded SERVICE operator that allows SPARQL to connect to JSON APIs and incorporate API information into its query answers. We begin by describing how JSON APIs function, followed by the syntax and semantics of the overloaded SERVICE operation in SPARQL. We finish by illustrating the utility of this extension using a set of real world examples.

6.3.1. JSON APIs, requests and navigating JSON documents, URI templates

While theoretically one can use our ideas to connect SPARQL to any Web API, we concentrate on the so-called REST Web APIs, which communicate via HTTP requests, and we only consider requests of type GET. Of course, any implementation needs to take care of many other details when connecting to APIs (e.g. authentication). Our implementation takes this into consideration, but these implementation details vary with APIs and systems, so here we just focus on the problem of evaluating these queries.

We assume that all API responses are JSON documents, and we use JSON *navigation conditions* to navigate and retrieve certain pieces of a JSON document. We always assume that the general structure of the JSON response is known by users; this can be achieved, for example, by including the schema of the response in the documentation of the API (see e.g. (Pezoa, Reutter, Suarez, Ugarte, & Vrgoč, 2016; Galiegue & Zyp, 2013)). This is a common assumption when one works with Web APIs.

The last ingredient we need are URI templates. We use them as a simple way to define placeholders that will be filled at the time an HTTP requests is made.

DEFINITION 6.3.1 (URI templates). A *URI Template* (Utempl, 2012) is an URI in which the query part may contain substrings of the form $\{?x\}$, for $?x$ in $\mathbf{V}$. For example, the following are URI templates:

```
http://weather.api/request?q={?city}
http://other.api/request?q={?city},{?country}
```

The idea behind these templates is that variable elements inside brackets are replaced by concrete values at the time the request is made. In what follows, we will refer to the variables in such substrings of a URI template U as the variables of U, and denote them with $\mathsf{var}(U)$.

6.3.2. Syntax and semantics of the extended SERVICE operator

Our proposal is based on overloading the SERVICE operator to allow for SERVICE-to-API patterns, which we define next.

DEFINITION 6.3.2 (SERVICE-to-API pattern). Let P_1 be a SPARQL pattern, U a URI template using only variables that appear in P_1, $?x_1, \ldots, ?x_m$ a sequence of pairwise distinct variables that do *not* appear in P_1, and $N_1, \ldots, N_m$ a sequence of JSON navigation instructions. Then the following is a SPARQL pattern, that we call a SERVICE-to-API pattern:

$$P_1 \,.\, \mathsf{SERVICE}\ U\ \{(N_1, N_2, \ldots, N_m)\ \mathsf{AS}\ (?x_1, ?x_2, \ldots, ?x_m)\} \tag{6.1}$$

The idea is that we now allow users writing queries to be able to access information given by APIs they know of. The intuition behind the evaluation of this operator over a graph G is the following. For each mapping μ in the evaluation $[\![P_1]\!]_G$ we instantiate every variable $?y$ in the URI template U with the value $\mu(?y)$, thus obtaining an IRI which is a valid API call[1]. We call the API with this instantiated IRI, obtaining a JSON

[1]Note that replacing $?y$ in a URI template with $\mu(?y)$ may result in a IRI, and not a URI, since some of the characters in $\mu(?y)$ need not be ASCII. To stress this, we use the term IRI for any instantiation of the variables in a URI template.

document, say J. We then apply the navigation instruction N_1 to J and, assuming the instruction returns a basic JSON value, store this value into $?x_1$. Similarly, the value of N_2 applied to J is stored into $?x_2$, and so on. The mapping μ is then extended with the new variables $?x_1, \ldots, ?x_m$, which have been assigned values according to J and $N_1, N_2, \ldots, N_m$.

Notice that in (6.1) the pattern P_1 can again be an overloaded SERVICE pattern connecting to another JSON API, thus allowing us to obtain results from one or more APIs inside a single query.

EXAMPLE 6.3.1. Consider the SPARQL basic graph pattern P_1 given by $P_1 = \{$`?x wdt:P131 wd:Q22 . ?x rdfs:label ?y`$\}$ and the URI template U given by $U =$ `<http://weather.api/request?q={?y}>`. Then the following is a SERVICE-to-API pattern.

$$P = P_1 \text{ . SERVICE } U\{\texttt{(["temperature"]) AS (?t)}\}$$

As we explained, the intention of this pattern is to issue a call to U instantiated with all values $?y$ in the evaluation of P_1, and then assign to $?t$ those values found under the key `"temperature"` of the JSON document that is returned by the API.

Semantics. The semantics of a SERVICE-to-API pattern is defined in terms of the *instantiation* of an URI template U with respect to a mapping μ (denoted $\mu(U)$), which is simply the IRI that results by replacing each construct $\{?x\}$ in U with $\mu(?x)$, or an invalid IRI in case some $\mu(?x)$ is not defined. Thus, every mapping produces an IRI, which we then use to execute an HTTP request to the API in the body of the IRI. Formally, we have the following definition.

DEFINITION 6.3.3 (Calling APIs with templates and mappings). Let U be a URI template and μ a mapping. The instantiation of U with respect to μ, denoted as $\mu(U)$, corresponds to

- An arbitrary invalid IRI, if there is some $?x \in \mathsf{var}(U)$ such that $\mu(?x)$ is not defined, or
- The IRI obtained by replacing each construct $\{?x\}$ in U with $\mu(?x)$.

Then, the call to U with respect to μ, denoted as call(U, μ), is the result of the following process:

(i) Instantiate U with respect to μ, obtaining the IRI $\mu(U)$.

(ii) Produce a request to the API signed by $(\mu(U))$, obtaining either a JSON document (in case the call is successful) or an `error`.

Note that we adopt the convention that HTTP requests that do not give back a JSON document result in an error, that is, call$(U, \mu) =$ `error` whenever the request using U does not result in a valid JSON document.

EXAMPLE 6.3.2. Consider the mapping μ, such that $\mu(?y) =$ `Ben_Nevis`, and the template $U =$ `<http://weather.api/request?q={?y}>`. Then we have that $\mu(U) =$ `<http://weather.api/request?q=Ben_Nevis>`. When this request is executed against the weather API in the IRI, the answer result is either a JSON document similar to the one from Example 6.1.1, or it is an error.

To define the evaluation of SERVICE-to-API patterns, we need some more notation. First, if $?x$ is a variable and $t \in \mathbf{T}$ a term, we use $?x \mapsto t$ to denote the mapping that assigns t to $?x$ and does not assign values to any other variable. Next, given a JSON document J, a navigation expression N, and a variable $?x$, we define the set $M_{?x \mapsto J[N]}$ of mappings as follows:

$$M_{?x \mapsto J} = \begin{cases} \{?x \mapsto J\} & \text{, if } J \text{ is a basic JSON value} \\ \bigcup_{1 \leq i \leq k} \{?x \mapsto J_i\} & \text{, if } J = [J_1, \ldots, J_k], \text{ and all } J_i \\ & \qquad \text{are basic JSON values} \\ \emptyset & \text{, otherwise.} \end{cases}$$

The idea is that $M_{?x \mapsto J[N]}$ contains the single mapping $?x \mapsto J[N]$, when $J[N]$ is a basic JSON value (integer, string, or boolean), or, if $J[N]$ is the array $[J_1, \ldots, J_k]$ of basic JSON values, a set of k mappings each of which maps $?x$ to an element J_i. As per the definition above, we also assume that $M_{?x \mapsto J[N]} = \emptyset$ when J is not a valid JSON document, or $J =$ `error`. We can finally state the semantics of SERVICE-to-API patterns.

DEFINITION 6.3.4 (Semantics of SERVICE-to-API). Let P be a SERVICE-to-API pattern as specified in Definition 6.3.2, with the form

$$P = P_1 \,.\, \mathsf{SERVICE}\ U\ \{(N_1, N_2, \ldots, N_m)\ \ \mathsf{AS}\ \ (?x_1, ?x_2, \ldots, ?x_m)\}.$$

The semantics $[\![P]\!]_G$ is then defined as

$$[\![P]\!]_G = \{\mu \bowtie \mu_1 \bowtie \ldots \bowtie \mu_m \mid \mu \in [\![P_1]\!]_G,\ \mu_i \in M_{?x_i \mapsto \text{call}(U,\mu)[N_i]}\}.$$

Therefore, a mapping in $[\![P]\!]_G$ is obtained by extending a mapping $\mu \in [\![P_1]\!]_G$ by binding each $?x_i$ to the value in the JSON value call$(U, \mu)[N_i]$ (or one of the values therein, if said JSON document is an array).

In the case that call$(U, \mu) =$ `error` (e.g. when $\mu(?x)$ is not defined for some $?x \in \mathsf{var}(U)$), or that call$(U, \mu)[N_i]$ is not a basic JSON value, the mapping μ will not be extended to the variables $?x_i$, and will not be part of $[\![P]\!]_G$. Moreover, if $\mu \in [\![P_1]\!]_G$ is not compatible with a mapping in $M_{?x_i \mapsto \text{call}(U,\mu)[N_i]}$ (because, for example, μ assigns a different value to $?x_i$), then this will also not be part of $[\![P]\!]_G$. This behaviour is inline with the default behaviour of SPARQL SERVICE (Prud'hommeaux & Buil-Aranda, 2013) which makes the entire query fail if the SERVICE call results in an error.

In the case that we want to implement the `SILENT` option for SERVICE which makes the latter behave as an OPTIONAL (see (Prud'hommeaux & Buil-Aranda, 2013)), we would need to change the $\emptyset$ in the definition of $M_{?x \mapsto J[N]}$ to the empty mapping $\mu_\emptyset$, since this mapping can be joined with any other mapping.

Let us now illustrate this definition by means of some examples. First, we illustrate the basics of our definitions in a context where the call is given by a single variable and the values returned are basic JSON values.

EXAMPLE 6.3.3 (Example continued). Consider the SPARQL basic graph pattern P_1 given by $P_1 =$ {`?x wdt:P131 wd:Q22 . ?x rdfs:label ?y`} and the URI template U given by $U =$ <`http://weather.api/request?q={?y}`>. Let

$$P = P_1 \,.\, \mathsf{SERVICE}\ U\{\texttt{(["temperature"]) AS (?t)}\}$$

be the pattern we are evaluating over some RDF graph G, and assume that $[\![P_1]\!]_G$ contains the following mappings.

	$?x$	$?y$
μ_1	`wd:London`	`London`
μ_2	`wd:Berlin`	`Berlin`

The evaluation of P over G is then obtained by extending mappings in $[\![P_1]\!]_G$ using U. That is, we iterate over $\mu \in [\![P_1]\!]_G$ one by one, execute the call call(U, μ), and store the value call(U, μ)`["temperature"]` into the variable `?t`, in case that the obtained JSON value is a string, a number, or a boolean value, and discard μ otherwise. For example, if we assume that the calls are as follows,

$$\text{call}(\mu_1, U) = \ \{\texttt{"temperature": 22 }\}, \qquad \text{call}(\mu_2, U) = \ \texttt{error}$$

then $M_{?t \mapsto \text{call}(\mu_1, U)\texttt{["temperature"]}}$ contains the mapping that assigns the number 22 to the variable $?t$ and $M_{?t \mapsto \text{call}(\mu_2, U)\texttt{["temperature"]}}$ is empty. Thus, the evaluation $[\![P]\!]_G$ will contain the following mapping

	$?x$	$?y$	$?t$
μ_1	`wd:London`	`London`	22

As we previously remarked, the call call(U, μ_2) returns an error, the mapping μ_2 can not be extended, so it will not form a part of the output. In the case that the "`SILENT` semantic" is triggered, we would actually output μ_2 where $?t$ would not be bound.

Let us now present a more involved example, in which we have a few variables and the APIs return arrays.

EXAMPLE 6.3.4 (Example 6.3.3 continued). As in Example 6.3.3, we continue using pattern $P_1 = \{$`?x wdt:P131 wd:Q22 . ?x rdfs:label ?y`$\}$. This time, however, we use a different URI template U', given by

$$U' = \texttt{<http://weather.api/request?q=\{?y\}\&type=extended>},$$

which returns more extended weather information, including a 3-day forecast.

Once again, we have a graph G, and we assume that $[\![P_1]\!]_G$ contains mappings.

	$?x$	$?y$
μ_1	`wd:London`	`London`
μ_2	`wd:Berlin`	`Berlin`

This time, however, the documents that the API returns are different. In particular, call(μ_1, U') is now the following JSON document:

```
{
  "temperature": {
      "current": 22,
      "forecast": [18,17,18]
      },
  "description": "possible showers in the evening"
}
```

In order to get both the current temperature and the 3-day forecast, we use the SERVICE-to-API P', given by

$P' = P_1$. SERVICE $U'\{$`(["temperature"/"current"],`
`["temperature"/"forecast"/0], ["temperature"/"forecast"/1],`
`["temperature"/"forecast"/2] ) AS (?c,?f0,?f1,?f2)`$\}$

The idea is to retreive the current temperature into variable $?c$, and the three days of forecast in variables $?f0$, $?f1$ and $?f2$. Thus, in this case, to compute the answer we have the following sets of mappings:

- The single mapping in $M_{?c \mapsto \text{call}(\mu_1, U')\texttt{["temperature"/"current"]}}$ assigns 22 to variable $?c$,
- The single mapping in $M_{?c \mapsto \text{call}(\mu_1, U')\texttt{["temperature"/"forecast"/0]}}$ assigns 18 to variable $?f0$,
- The single mapping in $M_{?c \mapsto \text{call}(\mu_1, U')\texttt{["temperature"/"forecast"/1]}}$ assigns 17 to variable $?f1$, and
- The single mapping in $M_{?c \mapsto \text{call}(\mu_1, U')\texttt{["temperature"/"forecast"/2]}}$ assigns 18 to variable $?f2$,

Once again, if we assume that call(μ_2, U') returns `error`, then the evaluation $[\![P']\!]_G$ of P' over G is the mapping

	$?x$	$?y$	$?c$	$?f0$	$?f1$	$?f2$
μ	`wd:London`	`London`	22	18	17	18

Using JSON arrays directly gives us another option to retrieve this information, which would result in an answer with more mappings, but less variables, which would be specially useful in times we do not know the number of elements in the array we retrieve. For example, if we now use the pattern

$$P'' = P_1 \text{ . SERVICE } U'\{\texttt{(["temperature"/"current"],}$$
$$\texttt{["temperature"/"forecast"]) AS (?c,?f)}\}$$

Now the navigation instruction `["temperature"/"forecast"]` gives the array $[18, 17, 18]$ when evaluated in call(μ_1, U'). Then, the set

$$M_{?c \mapsto \text{call}(\mu_1, U')\texttt{["temperature"/"current"]}}$$

still contains only the mapping that assigns 22 to variable $?c$, but now the set

$$M_{?c \mapsto \text{call}(\mu_1, U')\texttt{["temperature"/"forecast"]}}$$

contains three mappings:

	$?f$
σ_0	18
σ_1	17
σ_2	18

Thus, in this case the semantics of P'' is similar to that of P', but where the values of $f0$, $f1$ and $f2$ are distributed into the value of f in three different mappings. That is, the evaluation $[\![P'']\!]_G$ of P' over G is the set of mapping given by:

	$?x$	$?y$	$?c$	$?f$
μ_1^1	wd:London	London	22	18
μ_1^2	wd:London	London	22	17
μ_1^3	wd:London	London	22	18

6.4. A Basic Implementation

In this section we propose a way to implement the overloaded SERVICE operation on top of any existing SPARQL engine without the need to modify its inner workings. To do so, we partition each query using this operator into smaller pieces, and evaluate these using the original engine whenever possible. The idea here is to obtain all the information needed to execute the API calls, and then do all the calls at once.

Before we describe our algorithms, we have a few important issues to address. First, we remark that all this machinery is designed to work under the assumption that API name parameters are known, as well as the schema of the responses of each API call, and this is why our operators are designed so that users input all of this information at the time of processing queries. Furthermore, we also assume that API results are correct: our goal in this chapter is to produce optimal evaluation of patterns calling APIs, but we do not deal with other important problems such as incomplete information, correctness of web results, or entity-linking.

6.4.1. Basic processing algorithm

The basic implementation delegates the computation of all SPARQL components to the system. When evaluating the answers of a query over a graph G, whenever the system encounters a pattern of the form

$$P \equiv P_1 \ . \ \mathsf{SERVICE}\ U\{(N_1, N_2, \ldots, N_m)\ \mathsf{AS}\ (?x_1, ?x_2, \ldots, ?x_m)\}$$

that needs to be processed, our implementation proceeds as follows. First, we compute the answers $[\![P_1]\!]_G$ by calling again our implementation. Then, we use Algorithm 5 to compute the answers $[\![P]\!]_G$ of the full pattern, which can be invoked once we know the answers of the basic pattern accompanying the SERVICE call. Finally, we serialize the set of mappings M using the VALUES operator, as in (Aranda et al., 2014), to allow

it to be used by the next graph pattern inside the WHERE clause in which it appears, thus enabling us to delegate once again the computation of the answer to a SPARQL system.

Regarding the final step, the obtained mappings need to be serialized as strings in case P is followed by another graph pattern P_2. In particular, if we are processing a query of the form `SELECT * WHERE` $\{P\ .\ P_2\}$, with P as above, then P_2 needs to be able to access the values from the mappings matched to P.

Algorithm 5 Naive evaluation of a SERVICE-to-API pattern.

Input: A graph G, a pattern $P \equiv P_1$. SERVICE $U\{(N_1, N_2, \ldots, N_m)$ AS $(?x_1, ?x_2, \ldots, ?x_m)\}$, set $[\![P_1]\!]_G$ of mappings
Output: Answer $[\![P]\!]_G$

```
Initialize M, M1, ..., Mm as ∅
G_temp ← ∅ named after IRI t
for μ ∈ [[P1]]_G do
    Execute call(U, μ)
    if call(U, μ) returns error then
        continue to the beginning of the loop
    end if
    for i such that 1 ≤ i ≤ m do
        compute M_i = M_{?x_i ↦ call(U,μ)[N_i]};
        if M_i = ∅ then
            break
        end if
    end for
    Let M = M ∪ ({μ} ⋈ M1 ⋈ ... ⋈ Mm)
end for
return M
```

With this implementation, the natural question is whether it can be optimized. As we have mentioned in the introduction, the bottleneck in our case is API calls, so if we want to evaluate queries efficiently, we need to do the least amount of API calls as possible. There are a number of optimisations we can immediately apply to our basic implementation that will reduce the number of calls, and we discuss them next. Afterwards, in Section 6.5, we consider a rather different question, for a broad subclass of patterns: Can we reformulate query plans to make sure we are making as few calls as possible?

6.4.2. Immediate optimisations

Here we describe two simple approaches for reducing the number of API calls: avoiding duplicate calls, and caching. These simple optimisations will also be compared to the more general algorithm for minimising the number of calls that we propose in Section 6.5.

Removing duplicate calls. In many scenarios we might end up with several API calls for the same URL. For example, a simple query of the form

```
1 SELECT ?x ?t WHERE {
2     ?x ex:label1 ?z .
3     ?x ex:label2 ?n .
4     SERVICE <http://api.org/{?n}> {
5       (["time"]) AS (?t)
6     }
```

might contain several mappings where the same value is bound to the variable $?n$, resulting in several calls using the same value, which is suboptimal. To eliminate this behavior we proceed as follows. We first obtain all *distinct* values $?n$ that can be bound in the query, produce one call for each value, and then join with the result of the rest of the query. More generally, we can replace

$$P_1 \text{ . SERVICE } U \text{ } \{ (N_1, N_2, \ldots, N_m) \text{ AS } (?x_1, ?x_2, \ldots, ?x_m)\}, \tag{6.2}$$

where U uses variables $y_1, \ldots, y_n$, with the SPARQL pattern

$$\begin{aligned}
&\{ \texttt{SELECT } \mathsf{var}(P_1) \texttt{ WHERE } \{P_1\} \} \texttt{ AND} \\
&\{ (\texttt{SELECT DISTINCT } ?y_1, \ldots, ?y_n \texttt{ WHERE } P_1) \\
&\quad \texttt{SERVICE } U \texttt{ } \{(N_1, N_2, \ldots, N_m) \texttt{ AS } (?x_1, ?x_2, \ldots, ?x_m) \} \}
\end{aligned}$$

It is easy to see that these patterns are equivalent. However, this transformation introduces a significant increase of the workload of our local database, so the usage depends heavily on how slow we expect APIs to respond: the slower the API response time, the better that this optimization performs.

Caching. Clearly the best way of reducing API calls is to not do them at all, because we already have them in the system. This is important even if we are dealing with just a single query, as several mappings may actually require the same API calls.

Our implementation has support for caching API calls while processing the same SERVICE-to-API pattern, which should remove the need for duplicate requests when processing a single SERVICE-to-API query. This caching is brute-force: every time a call to an API is produced, we cache the exact IRI that was used in this call, as well as the resulting JSON document (if the call was successful). Thus, before each call we first retrieve the IRI in the cache (which is a very fast operation since we maintain an index on the cache), and only proceed with the call if we cannot find the answer. Of course, for complex queries there is a memory issue where all cached files may not fit into working memory. If this is the case we just keep the files in disk until we finish processing the API. We remark that this is a usual problem when caching data, and one can tackle this problem using the same techniques as general caching in databases (for example, by leasing the control of the cache to the operating system), and for this reason we do not discuss this issue any further.

It is important to note that we are assuming that making different calls to the same URL will return the same answer. Thus, functions as `Time()` or `RandomNumberGenerator()` will not be updated for the duplicate calls. This is a reasonable assumption since the consecutive calls will be done in the same SERVICE-to-API query.

However, one could also think of a caching strategy that is based on optimizing calls across the evaluation of multiple SERVICE-to-API patterns, and over a wider time span. The problem with this optimization is, precisely, that it is only correct when the result of the same API call does not change over time, which is something we cannot control.

6.5. A Worst-case optimal algorithm

Our goal is to evaluate SERVICE-to-API queries as efficiently as possible, which implies minimizing the number of API calls we issue when evaluating queries. This leads us to the following question: what is the minimal amount of API calls that need to be issued to answer a given query? Ideally, we would like to issue a number of calls that is linear in the size of the output of the query: for each tuple in the output we issue only those calls that are directly relevant for returning that particular tuple. But in general this is not possible. Consider a pattern of the following form:

$$\{\ ?x_0\ p\ ?x_1\ \}\ \ldots\ \{\ ?x_{m-1}\ p\ ?x_m\ \}\ .\ \mathsf{SERVICE}\ U\ \{\ (N)\ \mathsf{AS}\ ?y\ \}$$

where U is a call that uses all variables $?x_1, \ldots, ?x_m$. Then the number of calls we would need to issue could be of order $|G|^m$ (e.g. when all triples in G are of the form (a, p, b)), but depending on the API data the output of this query may even be empty!

What we can do is aim to be optimal in the worst case, making sure that we do not make more calls than the number we would need in the worst case over all graphs and APIs of a given size. We can devise an algorithm that realises this bound if we focus on the smaller class of SPARQL queries made just from concatenating basic graph patterns and SERVICE-to-API operators, which we denote as *conjunctive patterns*. This is the federated analogue of conjunctive queries, which amount to roughly two thirds of the queries issued on the most popular endpoints on the Web, according to (Bonifati, Martens, & Timm, 2017).

We tackle this problem with a novel framework that reduces API calls to the problem of bounding the number of tuples in the output of a relational query, a subject that has received considerable attention in the past few years in the database community (see e.g. (Atserias et al., 2008; Gottlob, Lee, Valiant, & Valiant, 2012; Ngo, Porat, Ré, & Rudra, 2012)), but over an extended relational model that includes *access methods* for each relation. We then devise an algorithm that can evaluate queries over relational databases with access methods, a result which is of independent interest. We start this

section with a high-level explanation of our technique, and then proceed to design the algorithm and establish the promised worst-case optimality.

6.5.1. Overview of our framework

We begin by showing how to cast the problem of processing a SERVICE-to-API pattern as a problem of answering a join over a relational database with access methods. More precisely, given a graph G and a pattern P, we construct a relational instance $I_{P,G}$ and a relational query Q_P such that the evaluation of Q_P over $I_{P,G}$ corresponds precisely to the evaluation $[\![P]\!]_G$. Note that we can do this because P is a conjunctive pattern, and therefore every mapping in $[\![P]\!]_G$ bounds the same variables.

So how does Q_P and $I_{P,G}$ look like? We can easily translate basic graph patterns in P as relational tables, simply by storing all the results of the pattern as a table. For example, given a basic graph pattern $P' = \{?x\ p_1\ ?y\}.\{?x\ p_2\ ?z\}$, we can store it as a relation R' on attributes $?x, ?y, ?z$, such that $(a, b, c) \in R'$ if and only if both (a, p_1, b) and (a, p_2, c) are triples in G.

But APIs require a more careful treatment. We model them as relations with *access methods* (see e.g. (Benedikt, Leblay, & Tsamoura, 2015; Calì & Martinenghi, 2008)). Intuitively, the idea is that these relations have a set of *ouptut attributes* that can only be accessed once we know the *input attributes*. Then, the variables used to construct an API call are represented as input attributes, and the information we extract from the call is modelled as output attributes.

In order to state our bounds, denote by $M_{Q,D}$ the maximum size of the projection of any relation appearing in a relational query Q over a single attribute in the database D. The main result we show in this section is the following

PROPOSITION 6.5.1. Any conjunctive SERVICE-to-API P can be evaluated over a graph G using a number of calls in

$$O(M_{Q_P,I_{P,G}} \times 2^{\rho^*(Q_P,I_{P,G})})$$

where $2^{\rho^*(Q_P,I_{P,G})}$ is the AGM bound of the query. This bound is interesting because we can bound the number of API calls in terms of factors that depends on the size of

the database. Thus, one would like to define an algorithm that performs a number of API calls less or equals to this bound.

The proof of this proposition comes from a novel result establishing tight bounds for answering join queries over relations with access methods. The remainder of this section is devoted to the proof of these bounds, and then relating it to Proposition 6.5.1. We begin by formally defining relations with access methods, and then show how one can minimize the number of calls when evaluating queries over such relations.

6.5.2. Relational setup

In the following we assume familiarity with relational databases and schemas, and relational algebra (Abiteboul et al., 1995). We begin with the notion access methods.

DEFINITION 6.5.1 (relational schemas with access methods). An access method for a relation R partitions its attributes R into input attributes and output attributes. We denote *access methods* with the same symbol as the relations they specify, but making explicit which of the attributes are input attributes, and which are output attributes. For example, an access method for a relation $R(A, B, C)$ with attributes A, B and C and where A and C are input attributes is denoted by $R(A^i, B^o, C^i)$ (letter i is a shorthand for *input* and o for *output*).

Access methods impose a restriction on the way queries are to be evaluated, as there are queries that cannot be evaluated at all. For example, consider a schema with relations $R(A^i, B^o)$, $S(A^o, B^o)$ and $T(B^i, C^o)$. Then $S \bowtie R \bowtie T$ can always be evaluated[2], since the input for R is an output of S and likewise for T. However, $R \bowtie T$ can not be evaluated, as we do not have a source for the input of R. Let us formalize these notions.

DEFINITION 6.5.2 (Feasible join queries). A *join query* over a relational schema $\mathcal{S}$ with access methods is a query of the form $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$, where each R_i is a (not necessarily distinct) access method for a relation in $\mathcal{S}$, and each join is a natural join.

[2]We abuse the notation and denote relational joins using the same symbol that we use for mappings; the two operators are always distinguished by the context.

An access method for relation R_i in Q is *covered* if all of its input attributes appear as an output in any of the relations $R_1, \ldots, R_{i-1}$. If all its access methods are covered, then Q is said to be *feasible*.

Naturally, a join query can only be answered if it is equivalent to a feasible query, so without loss of generality we focus on feasible queries. It turns out that this is also enough for our purposes, as queries we produce out of conjunctive SERVICE-to-API patterns are always feasible.

The task of evaluating a query over access methods closely resembles querying APIs, as in both cases we are using known information to query data sources we don't know. Furthermore, an API itself can be understood as a relation with access methods, in which the information requested by the API is their inputs, and the information returned is the output. However, in order to analyze the cost of answering a relational query with access methods we need to fix the communication cost of accessing a particular set of tuples that is under an access method with an input. As in APIs, we adopt the convention of one call per each different set of inputs.

DEFINITION 6.5.3 (Calls in access methods). For a relation T with input attributes $A_1, \ldots, A_k$ and a set R of tuples having all attributes $A_1, \ldots, A_k$, the number of calls required to answer $R \bowtie T$ corresponds to the size of $\pi_{A_1,\ldots,A_k}(R)$. Intuitively, this means that we answer $R \bowtie T$ by selecting all different inputs coming from the tuples of R, and issue one call for each of these inputs.

We can then analyse the number of calls for the naive left-deep join plan for $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$, which corresponds to setting $\phi_1 = R_1$ and iteratively computing $\phi_{i+1} = \phi_i \bowtie R_{i+1}$ until we obtain ϕ_m, which corresponds to the answers of Q. How many calls do we issue? In the worst case where all except R_1 are relations representing APIs, we would need to issue a number of calls corresponding to the sum of the tuples in R_1, $R_1 \bowtie R_2$, and so on until $R_1 \bowtie \ldots \bowtie R_{n-1}$.

EXAMPLE 6.5.1. Consider a schema $R(A^o, B^o), S(B^i, C^o), T(C^i, A^i)$ and query $Q = R \bowtie S \bowtie T$. The left deep plan for Q first computes $R \bowtie S$, which needs a number of calls equivalent to the number of tuples in $\pi_B(R)$. Then, we use the result

of $R \bowtie S$ to compute $R \bowtie S \bowtie T$, which requires a number of calls equivalent to the number of tuples in $\pi_{A,C}(R \bowtie S)$, which is quadratic on the size of R and S.

In the following section we show a tighter bound for the number of calls required, as well as an algorithm fulfilling this bound.

6.5.3. Optimal algorithm for join queries with access methods

Without loss of generality, in this section we assume that there is exactly one access method per relation in Q (if not one can construct two different relations, the worst case analysis does not change).

Our algorithm is inspired by the optimal plan exhibited in (Atserias et al., 2008; Grohe, 2013) for conjunctive queries without access methods. Starting from a join query Q with attributes $A_1, \ldots, A_n$, we construct a sequence of queries $\Delta_1, \ldots, \Delta_n$, where Δ_n is equivalent to Q, and where the evaluation of each such query Δ_i overestimates the evaluation of the query $\pi_{A_1,\ldots,A_i}(Q)$.

The construction is depicted as pseudocode in Algorithm 6. The idea is to order every attribute participating in the query, and for an (ordered) increasing amount of attributes $\mathcal{A}$, we obtain all potential values of these attributes that may be part of the final answer. The key component is that this set of potential values is the best overestimation of the API calls we need to issue when processing relations whose input values are contained in the set $\mathcal{A}$ of attributes.

Next, as each of these queries is feasible, we can evaluate them one-by-one over our database. As the analysis shows, this is in fact an optimal way of reducing the number of calls issued by the query (as long as we assume that we do not call two times the same access method with the same parameter).

Analysis. Let us first remark that the feasibility of Q^* follows from the fact that Q is feasible, so every relation with inputs $A_1, \ldots, A_i$ appears after all these attributes are outputs of previous relations, and we order attributes in the order of appearance. Furthermore, we assume that we store the results of all relations R with $\mathit{Input}(R) \neq \emptyset$ in memory the first time they are requested, and before we compute any projection

Algorithm 6 Rewrite.

Input: A feasible join query $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$ under access methods
Output: A sequence of queries $\Delta_1, \ldots, \Delta_n$

Initialize $M, M1, \ldots, M_m$ as $\emptyset$
$G_{temp} \leftarrow \emptyset$ named after IRI t
Let $A_1, \ldots, A_n$ be an enumeration of all attributes involved in Q, in order of their appearance
Let inputs(R) denote the set of all input attributes of the access method for R
Let $S^1_1, \ldots, S^1_{k_1}$ be all relations in $\{R_1, \ldots, R_n\}$ whose set inputs(S^1_ℓ) of input attributes is contained in $\{A_1\}$ (including relations with only output attributes)
$\Delta_1 \leftarrow \pi_{A_1}(S^1_1) \bowtie \ldots \bowtie \pi_{A_1}(S^1_{k_1})$
for i such that $2 \leq i \leq n$ **do**
    Let $S^i_1, \ldots, S^i_{k_i}$ be all relations such that inputs(S^i_ℓ) is contained in $\{A_1, \ldots, A_i\}$
    $\Delta_i \leftarrow \Delta_{i-1} \bowtie \pi_{A_1,\ldots,A_i}(S^i_1) \bowtie \ldots \bowtie \pi_{A_1,\ldots,A_i}(S^i_{k_i})$
end for
return $\Delta_1, \ldots, \Delta_n$

Algorithm 7 Evaluating a join query with access methods.

Input: A feasible join query $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$ under access methods, a database D
Output: Evaluation of Q over D with minimal calls

Let $A_1, \ldots, A_n$ be an enumeration of all attributes involved in Q, in order of their appearance
$\Delta_1, \ldots, \Delta_n \leftarrow Rewrite(Q)$
for i such that $1 \leq i \leq n$ **do**
    Let $S^i_1, \ldots, S^i_{k_i}$ be all relations such that inputs(S^i_ℓ) is contained in $\{A_1, \ldots, A_i\}$, and recall that $\Delta_i \leftarrow \Delta_{i-1} \bowtie \pi_{A_1,\ldots,A_i}(S^i_1) \bowtie \ldots \bowtie \pi_{A_1,\ldots,A_i}(S^i_{k_i})$; Then, Δ_i is feasible
    Evaluate Δ_i over database D using the evaluation of Δ_{i-1}
end for
return the evaluation of Δ_n over database D

over them, so that we do not have to issue another call to R using the same inputs. This only imposes a memory requirement that is at most as big as what we would need with the basic implementation described in the previous section.

Recall that for a query Q and instance D, $M_{Q,D}$ is the maximum size of the projection of any relation in Q over a single attribute, and $2^{\rho^*(Q,D)}$ is the AGM bound of the query. We then have the following result.

PROPOSITION 6.5.2. Let Q be a feasible join query over a schema with access methods and D a relational instance of this schema. Let Δ_n be the query constructed

from Q by the algorithm above. Then the number of calls required to evaluate Δ_n over D using a left-deep plan is in

$$O(M_{Q,D} \times 2^{\rho^*(Q,D)}).$$

PROOF. Let us assume that $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$, all over attributes $A_1, \ldots, A_n$, and let $\Delta_1, \ldots, \Delta_n$ as explained above. As we mentioned, observe that each Δ_i is also feasible. Now any API call we do for a relation R_j with input attributes $A_1, \ldots, A_i$ in our left deep plan is of the form

$$\big(\Delta_{i-1} \bowtie \pi_{A_1,\ldots,A_i}(R_1) \bowtie \ldots \bowtie \pi_{A_1,\ldots,A_i}(R_{j-1})\big) \bowtie \pi_{A_1,\ldots,A_i}(R_j).$$

The size of $\Delta_{i-1} \bowtie \pi_{A_1,\ldots,A_i}(R_i)$ is bounded by the size of $\Delta_{i-1} \times \pi_{A_i}(R_i)$, where $\times$ denotes the Cartesian product of relations, as Δ_{i-1} already contains a subexpression joining with $\pi_{A_1,\ldots,A_{i-1}}(R_i)$. Hence, the number of tuples in the output of the expression

$$\big(\Delta_{i-1} \bowtie \pi_{A_1,\ldots,A_i}(R_1) \bowtie \ldots \bowtie \pi_{A_1,\ldots,A_i}(R_{j-1})\big)$$

is bounded by $M_{Q,D} \cdot |\Delta_{i-1}(D)|$, as we only add to Δ_{i-1} the values $\pi_{A_i}(R)$ of the relation R with the greatest projection over A_i, the others act only as semijoins that filter the result. So to evaluate an appearance of a relation R_j in some Δ_i we need to pose at most $M_{Q,D} \cdot |\Delta_{i-1}|$ calls. Since by the AGM bound we have that $|\Delta_{i-1}| \leq 2^{\rho^*(Q,D)}$, we can therefore evaluate the appearance of R_j in Δ_i using at most $M_{Q,D} \cdot 2^{\rho^*(Q,D)}$ calls.

Therefore, the total number of calls is bounded by $m \cdot M_{Q,D} \cdot 2^{\rho^*(Q,D)}$ (as we are caching results), which is in $O(M_{Q,D} \times 2^{\rho^*(Q,D)})$ when we assume the query to be fixed (that is, when we consider the data complexity). □

Let m be the number of relations in Q and n the total number of attributes. If we are considering combined complexity (i.e. Q is part of the input), the bound above raises to $O(m \times M_{Q,D} \times 2^{\rho^*(Q,D)})$ for the algorithm that does caching. Likewise, the number of calls is in $O(n \times m \times M_{Q,D} \times 2^{\rho^*(Q,D)})$ if we rule out the possibility of storing results of the calls in memory during the process of evaluating the queries.

Optimality. Our last result establishes the optimality of our solution, as there are queries that must be evaluated using at least the number of calls demanded by Proposition 6.5.2.

PROPOSITION 6.5.3. There is a schema $\mathcal{S}$, a query Q and a family of instances $(D_n)_{n\geq 1}$ such that: (i) The maximum size of the projection of a relation in D over one attribute is n, (ii) The AGM bound is n^2, and (iii) Any algorithm evaluating Q must make at least n^3 calls to a relation with access methods.

PROOF. Consider the schema with access methods with relations $R(A^o, B^o, C^o)$, $S(B^i, C^i, D^o)$, $U(B^i, E^o)$ and $T(A^i, D^i, E^i)$, and an instance D where R contains tuples $\{(i, 1, 1) \mid 1 \leq i \leq n\}$, S contains tuples $\{(1, 1, i) \mid 1 \leq i \leq n\}$, U contains tuples $\{(1, i) \mid 1 \leq i \leq n\}$ and T contains n arbitrary tuples. Now let $Q = R \bowtie S \bowtie U \bowtie T$. The maximum size of the projection of a relation in D over one attribute is n, and it can be checked that the bound $2^{\rho^*(Q,D)}$ corresponds to n^2, by solving the appropriate linear program. Furthermore, since T has inputs which are only in the union of all R, S and U, any algorithm looking to answer Q must issue one call for each tuple in $\pi_{A,D,E}(R \bowtie S \bowtie U)$ (if not, one can create an interpretation for T where this algorithm does not answer the query correctly). We obtain that n^3 calls are needed, as this is the size of the ouptut of $R \bowtie S \bowtie U$ over D. □

6.5.4. Algorithm for SERVICE-to-API patterns

Let us now come back to SERVICE-to-API patterns. The way we provide an algorithm is by (1) translating SERVICE-to-API patterns into join queries with access methods, and then (2) constructing an optimal plan for the SERVICE pattern when given a plan for the relational query.

For (1), let P be a SERVICE-to-API pattern and Q_P the constructed join query, and consider an RDF graph G. Next we describe the construction of the relational query Q_P and explain how to construct the instance $I_{P,G}$ in which Q_P should be evaluated.

DEFINITION 6.5.4 (Relational counterparts of SERVICE-to-API patterns). Let G be a graph and P be a conjunctive SERVICE-to-API pattern of the form:

$$\{\{\{\{\{\{\ P_1\ .\ S_1\ \}\ .\ P_2\ \}\ .\ S_2\ \}\ .\ P_3\ \}\ .\ S_3\ \}\ \ldots\ P_n\ \}\ .\ S_n$$

where each P_i is a basic graph pattern (not using SERVICE) and each S_i is a different SERVICE-to-API pattern.

Query Q_P is then defined as

$$Q_P = R_1 \bowtie T_1 \bowtie R_2 \bowtie T_2 \bowtie \ldots \bowtie R_n \bowtie T_n, \tag{6.3}$$

where each R_i is a relation whose attributes corresponds to the variables in P_i, and each T_i is a relation with access methods constructed from S_i as follows. Suppose T_i is of the form

$$\mathsf{SERVICE}\ U\ \{(N_1, N_2, \ldots, N_m)\ \mathsf{AS}\ (?x_1, ?x_2, \ldots, ?x_m)\}.$$

Then the output attributes in T_i are $?x_1, \ldots, ?x_m$, and the input attributes of T_i are $\mathsf{var}(U)$, the variables mentioned in the URI template U.

Next, the relational instance $I_{P,G}$ is defined as follows.

- Each relation R_i in $I_{P,G}$ with attributes $?x_1, \ldots, ?x_m$ contains the set of tuples

$$\{(\mu(?x_1), \ldots, \mu(?x_n)) \mid \mu \in [\![P]\!]_G\}.$$

- Each relation T_i in $I_{P,G}$ with input attributes $?z_1, \ldots, ?z_k$ and output attributes $?y_1, \ldots, ?y_p$ contains the set of tuples

$$\{(\mu(?z_1), \ldots, \mu(?z_k), \mu(?y_1), \ldots, \mu(?y_p)) \mid \mu \in [\![P]\!]_G\}.$$

With this in mind, we can now show the soundness of using a relational translation to answer conjunctive patterns:

LEMMA 6.5.1. Let P be a conjunctive SERVICE-to-API pattern using variables $\{?x_1, \ldots, ?x_\ell\}$. A tuple $(a_1, \ldots, a_\ell)$ is in the evaluation of Q_P over $I_{P,G}$ if and only if there is a mapping $\mu \in [\![P]\!]_G$ such that $(a_1, \ldots, a_\ell) = (\mu(?x_1), \ldots, \mu(?x_\ell))$.

PROOF. Let us assume P and G are of the form described in Definition 6.5.4, where in particular the set of variables mentioned in P is $\{?x_1, \ldots, ?x_\ell\}$, $Q_P = R_1 \bowtie T_1 \bowtie R_2 \bowtie T_2 \bowtie \ldots \bowtie R_n$ and $I_{P,G}$ is constructed as explained above.

For the **if** direction, consider a tuple $(a_1, \ldots, a_\ell)$ in the evaluation of Q_P over $I_{P,G}$. By definition, there must be mappings $\mu_1, \ldots, \mu_n$ and $\mu'_1, \ldots, \mu'_{n-1}$ such that (i) all these mappings are compatible and their value over $\{?x_1, \ldots, ?x_\ell\}$ must coincide whenever at least two of them have one of these variables in the domain, (ii) μ_i is in $[\![P_i]\!]_G$ and (iii) Letting

$$\mu = \bigcup_{1 \leq j \leq n} \mu_i \cup \bigcup_{1 \leq k \leq n-1} \mu'_k$$

we have that for each SERVICE call

$$S_j = \mathsf{SERVICE}\ U\ (N_1, N_2, \ldots, N_m)\ \mathsf{AS}\ (?z_1, ?z_2, \ldots, ?z_m),$$

we have that call$(U, \mu)[N_k]$ is either $\mu'_j(?z_k)$ or is an array that contains $\mu'_j(?z_k)$.

By the semantics of SPARQL and (conjunctive) SERVICE-to-API patterns we get that $\mu \in [\![P]\!]_G$, and that $(a_1, \ldots, a_\ell) = (\mu(?x_1), \ldots, \mu(?x_\ell))$.

For the **only if** direction, consider such a mapping μ in $[\![P]\!]_G$ and let $(a_1, \ldots, a_\ell) = (\mu(?x_1), \ldots, \mu(?x_\ell))$. It is then easy to see that each of the relations are populated with the appropriate subsets of $(a_1, \ldots, a_\ell)$, so that this tuple must be in the evaluation of Q_P over I_P. □

Note that for finding the worst-case optimal plan for Q_P we do not need to construct the instance $I_{P,G}$, as this would amount to pre-computing the answer $[\![P]\!]_G$. The lemma above is just to state that the translation is correct.

Next we show how to construct an optimal plan for P from the optimal plan we know how to construct from Q_P. Together with Lemma 6.5.1, this completes the proof of Proposition 6.5.1.

PROPOSITION 6.5.4. Let P be SERVICE-to-API pattern, G an RDF graph and Q_P, $I_{P,G}$ the corresponding relational query with access methods and instance as constructed above. Then any optimal query plan Q^* for Q_P over an instance $I_{P,G}$ can be

transformed (in polynomial time) into a query plan for P that evaluates P over G using the same amount of API calls as the evaluation of Q^*.

PROOF. The plan for P mimics step-by-step the plan for Q_P. That is, assume that Δ_n is the reformulation of Q_P from Section 6.5.3, and that $\Delta_n = E_1 \bowtie \ldots \bowtie E_r$, with each E_r a join-free expression. Starting with $\phi_1 = E_1$, we iteratively compute the the set $[\![\phi_i]\!]_G = [\![\phi_{i-1}]\!]_G \bowtie E_i$ of mappings for each $i = 2 \cdots r$. This is done in the following way. Whenever $E_i = \pi_{?x_1,\ldots,?x_p} R_i$, we evaluate the query SELECT $?x_1, \ldots, ?x_p$ WHERE P_i over G. On the other hand, if E_i is a relation using T_j for the first time, we call the API (because Q_P is feasible we will have all the needed input parameters), cache all the API results and then only retrieve the attributes that are not projected out in E_i. All subsequent appearances of T_j are evaluated directly on the JSON file we store while the evaluation continues. (If we are not using this mild form of caching, then we need to call the API for each E_k, where $k > i$, that uses T_j.) Since the query ϕ_r is equivalent to Q^*, it is also equivalent to Q_P. Thus the output of this query plan correctly computes $[\![P]\!]_G$ by Lemma 6.5.1. The number of calls is worst-case optimal by Proposition 6.5.2 and Proposition 6.5.3.

□

6.6. Experiments

The goal of this section is to give empirical evidence that the worst-case optimal algorithm of Section 6.5 does indeed minimize the number of API calls. For this, we run a set of experiments based on the Berlin SPARQL Benchmark (v3.1) (Bizer & Schultz, 2009). In the corresponding subsections, we specify how we simulate API calls and responses. All the experiments where run on a 64-bit Windows 10 machine, with 8 GB of RAM, and Intel Core i5 7400 3.0 GHz processor.

Implementation. Our implementation of SERVICE-to-API patterns is done on top of Jena TBD 3.4.0 using Java 8 update 144. We differentiate four evaluation algorithms for SERVICE-to-API patterns: (1) **Vanilla**, the base implementation described in Section 6.3; (2) **Cache**, the base algorithm that uses caching to avoid doing the same API

call more than once; (3) **WCO**, the implementation of our WCO algorithm (3) **WCO + Cache**, which is the worst-case optimal algorithm avoiding duplicate calls via caching.

6.6.1. Berlin Benchmark

As we mention in Section 2, the Berlin Benchmark dataset (Bizer & Schultz, 2009) is inspired by an e-commerce use case. It consists of products that are offered by vendors and are reviewed by users. Each one of those entities has properties related with them (such as labels, prices, etc.). The benchmark itself is composed of 12 queries, named Q1-Q12, and the size of the dataset is specified by the user. To test our implementation we generate 3 distinct datasets of different size. The specifications can be found at the Table 6.2.

Dataset	Number of triples	Size
D_1	19625500	5 Gb
D_2	41795444	10 Gb
D_3	79881347	20 Gb

TABLE 6.2. Sizes of the datasets created for the Berlin Benchmark.

Adapting the Berlin benchmark to include API calls. Our adaptation consists of exposing the data of five recurrent patterns we find in the benchmark queries as APIs that return JSON documents. For instance, one such pattern is

```
{?x bsbm:productPropertyNumericZ ?y}
```

where `Z` is a number between 1 and 5. This pattern is used to return the value of some numeric property of a product with the label `?x`, so we created a (local) API route `api/numeric-properties/{label}`, that will return all the values of numeric properties of an object as a JSON file. That is, if a product with the IRI `bsbm:Product1` has a label `"Product1"`, and its numeric properties are

- `PropertyNumeric1` $= 3$,
- `PropertyNumeric2` $= 10$,

and then the request

```
api/numeric-properties/Product1
```

returns the JSON: `{ "p1": 3, "p2": 10 }`.

All together, we simulate an API that has the following five paths:

- `api/numeric-properties/{label}`, which receives a label of a product and returns a JSON response containing all of the product's numeric values as pairs `"propertyNumericX" : n`, with `n` the value of `propertyNumericX`.
- `api/textual-properties/{label}`, which receives a label of a product and returns a JSON response containing all of the product's numeric values as pairs `"propertyTextualX" : s`, with `s` the value of `propertyTextualX`.
- `api/features/{label}`, which receives a label of a product and it returns an array with all the labels of its features.
- `api/offers/{offer-id}`, which receives the `id` of an Offer and returns the properties of this offer.
- `api/review/{review-id}`, which receives the `id` of a Review and returns the properties of this review.

Next, we transform the original benchmark queries by replacing each pattern used when creating the APIs by a SERVICE call to the corresponding API. For instance, in the case of the "numeric properties API" described above, we replace each pattern of the form: `{?product bsbm:productPropertyNumericX ?valueX}`, by the following API call:

```
1 SERVICE <api/numeric-properties/{label}>{
2   (["pX"]) AS (?valueX)
3 }
```

We do a similar transformation for each of the other four patterns that were exposed as APIs. We run all the queries of the Berlin Benchmark except Q6, Q9, and Q11, because they were too short to include API calls in their patterns. To understand the results of the experiment, first we have to discuss the meaning of each query (Bizer & Schultz, 2009).

- **Q1**: Find products for a given set of generic features.

- **Q2**: Retrieve basic information about a specific product for display purposes.
- **Q3**: Find products having some specific features and not having one feature.
- **Q4**: Find products matching two different sets of features.
- **Q5**: Find products that are similar to a given product.
- **Q7**: Retrieve in-depth information about a product including offers and reviews.
- **Q8**: Retrieve recent English language reviews for a specific product.
- **Q10**: Get cheap offers which fulfill the consumer's delivery requirements.
- **Q12**: Export information about an offer into another schema.

We change the `OPTIONAL` operator in each query by `AND`, because the two are the same in terms of worst case optimal analysis.

Results. Since our goal is to show how more involved algorithms reduce API calls, we report the total number of calls issued by the Vanilla implementation when evaluating queries, as well as the calls issued by the other three implementations, in terms of the percentage with respect to the Vanilla evaluation. Here we do not include the results of the WCO algorithm without caching, since it does not present a significant improvement with respect to the Cache implementation and the best performance is obtained with the WCO+Cache algorithm. Results for D_1 are presented in Table 6.3.

As we see, avoiding duplicate calls reduces the number of calls to some extent, but the best results are obtained when we use the worst-case optimal algorithm. As a summary we show the average percentage of API calls done with respect to the original algorithm. On average, the worst-case optimal algorithm with caching does 14% of the calls done by the naive algorithm.

Here we note that Q2, Q6, Q7 and Q8 are related to a single product that is fixed. Thus, when we use the WCO algorithm, all the triple patterns related to such product are resolved before requesting data from the API, hence the number of calls to the API is reduced dramatically except for Q7, where the resolution of the triple patterns related to the product does not filter the possible answers. Also for query Q12, the query involves a single offer that is also fixed but it is not being filtered by the following

	Calls Vanilla	% Vanilla	% Cache	% WCO+Cache
Q1	28	100%	100%	89%
Q2	43	100%	6%	0%
Q3	28	100%	100%	39%
Q4	139	100%	100%	10%
Q5	346	100%	8%	8%
Q7	6	100%	100%	100%
Q8	2	100%	100%	0%
Q10	6	100%	100%	0%
Q12	1	100%	100%	100%
AVG	66	100%	40%	14%

TABLE 6.3. Number of API calls performed by the Vanilla implementation over D_1 as well as the percentage of this total issued by all four implementation. Berlin queries are adapted into SERVICE-to-API patterns. The last row shows the average percentage of calls that the algorithm does compared to Vanilla. The combination of WCO + Cache does 14% of the original calls done by the naive algorithm.

triple patterns, and then the number of API calls remains the same for every algorithm. It is important to note that not every product is related with certain properties (such as `propertyNumeric4` or `propertyNumeric5`), and then, it is possible that our algorithm resolves that a query answer is empty before executing a SERVICE-to-API. This happens for queries Q2, Q8 and Q10, where we reduce the number of API calls to zero. Finally we note that for the other queries, the WCO algorithm is definitely an improvement with respect to the naive algorithm. The trend shown for this dataset holds for D_2 and D_3 as we can see in tables 6.4 and 6.5 respectively.

	Calls Vanilla	% Vanilla	% Cache	% WCO+Cache
Q1	170	100%	81%	77%
Q2	49	100%	6%	0%
Q3	163	100%	100%	34%
Q4	169	100%	100%	63%
Q5	1348	100%	7%	7%
Q7	6	100%	100%	100%
Q8	4	100%	100%	50%
Q10	6	100%	100%	0%
Q12	1	100%	100%	100%
AVG	209	100%	30%	20%

TABLE 6.4. Percentage of total issued by all four implementations over D_3. The combination of WCO + Cache does 21% of the original calls done by the naive algorithm.

	Calls Vanilla	% Vanilla	% Cache	% WCO+Cache
Q1	343	100%	80%	73%
Q2	49	100%	6%	0%
Q3	323	100%	100%	31%
Q4	275	100%	100%	58%
Q5	2642	100%	10%	10%
Q7	5	100%	100%	100%
Q8	6	100%	100%	50%
Q10	5	100%	100%	20%
Q12	1	100%	100%	100%
AVG	405	100%	31%	21%

TABLE 6.5. Percentage of total issued by all four implementations over D_3. The combination of WCO + Cache does 21% of the original calls done by the naive algorithm.

As we see, to integrate data from JSON APIs is feasible in practice thanks to our algorithm based on worst-case optimal join techniques. We believe that this is an important feature in order to increase the number of data sources available available for the Semantic Web Services, and also, we think that this approach can be useful to integrate more data sources, such as GraphQL APIs and tabular data.

Chapter 7. CONCLUSIONS

In this work we explored the Semantic Web from different points of view, starting from the expressivity of SPARQL and the proposal of tools for covering use-cases that the language does not cover now, such as in-database graph analytics. This new features for the language showed us that it is necessary to improve the efficiency of RDF systems and the evaluation of SPARQL queries, and thus, we study techniques based on worst-case optimal join algorithms that can improve dramatically the performance of RDF databases. Finally, this techniques showed us a way to design algorithms that allow us to integrate data from Web APIs that currently are bit available for the Semantic Web.

7.1. Lessons Learned

SPARQL have had considerable growth through its more than 13 years of life, but there are aspects of the language that still need some improvement. The first thing that we discuss is about having functionalities that allows the user to express queries related to paths within an RDF graph. Although we have some features such as property paths, or similar recursive extensions, we show that the current alternatives cannot express all the queries that one would like to. Also, as our experiments show, some easy queries cannot be handled efficiently by RDF systems, and queries that are executed over large datasets usually end with memory errors. However, in this particular case, we faced a issue that has been studied before but for other data models.

Thus, we study linear recursive queries which have been well established for relational databases and cover the majority of interesting use cases. Moreover we show how to implement them as an extension to the Apache Jena framework. Given that recursion can express many requirements outside of the scope of SPARQL 1.1, coupled with the fact that implementing the recursive operator on top of existing SPARQL engines does not require to change their core functionalities, allows us to make a strong case for including recursion in the future iterations of the SPARQL standard.

Of course, such an expressive recursive operator is not expected to beat specific algorithms for smaller fragments such as property paths. But nothing prevents the language having both a syntax for property paths and also for recursive queries, with different algorithms for each operator. On the other hand, systems looking for more lightweight alternatives may prefer to eschew algorithms for property paths and just compile everything into our recursive operator. One key point here is that SPARQL has had to take several design decisions since its proposal, in order to be applicable for large-scale graphs. But in our opinion, in this case it is not necessary to begin from scratch. Some challenges are shared by all data models, and in particular, the topic of recursive queries have interesting results that have to be revisited.

Then we study how to use a recursive language to do something that has not been explored enough, in particular for RDF databases: doing in-database graph analytics. We believe that the combination of graph queries and analytics is a natural one, in the sense that tasks of interest to users often involve interleaving both paradigms. The SPARQAL language provides a way to express such tasks, and makes initial steps towards a system to support them. We see this language as being useful for combining querying and analytical tasks specifically in an RDF/SPARQL setting. However, we think that this is still and open problem, because an on-top implementation for doing queralytics is useful mainly for interactive use, which needs to be restudied for analyzing huge RDF graphs. Also, this part of the work shows us that it is important to improve the efficiency of RDF systems. In particular, to execute recursive queries efficiently, it is imperative to improve the evaluation of join queries for RDF databases.

To fill this gap, we looked to a relatively recent field that has been studied by the database theory community: the idea of worst-case optimal join algorithms. We argue that RDF databases are using the same techniques that have been used for years for relational databases, based on pairwise joins, left deep plans and tree based indexes. We believe that it is necessary to try something new, and in this case we opted for worst-case optimal join algorithms since the approach makes sense for graph databases in general. Thus, to the best of our knowledge, here we present the first work that looks at the benefits of worst-case optimal join algorithms for RDF databases. Based

on our results, we believe that worst-case optimal joins should become widely adopted by SPARQL engines in the near future; we also firmly believe that our results are only a starting point in this line of research, and that there is still much room left for maximizing the potential benefits of such algorithms in a SPARQL setting. Along these lines, we remark that we released an open source fork of Apache Jena implementing LFJ that can serve as a baseline for future experiments, and a novel benchmark based on Wikidata that can be used for testing future developments in a real-world setting.

We believe that the Semantic Web has to improve in order to make it go further, and one key aspect to improve is the reliability of RDF engines. Consider the following case: we have a friend that is not familiar with the Semantic Web technologies and he wants to build a project for exposing some data. Is there any chance that we recommend him to use an RDF database? In our opinion there are occasions for recommending RDF databases, but the decision is not always clear. The main concern we see is that RDF databases are not a top-tier alternative for production environments, in contrast to SQL databases, document oriented storages or even other graph-based models, such as Neo4J. One of the lessons learned by doing this work is that the techniques used to process SPARQL queries rely on techniques created for relational databases, and it is not clear that RDF databases can give us a significant improvement in terms of performance in comparison with SQL systems; at first, one can say RDF is fine for publishing open data, because RDF endpoints have been widely adopted for this purpose. But what if the amount of requests to our endpoint is considerably high? Then, at the moment of recommending a system, it seems reasonable to think about exposing an API that relies on a SQL database or a document-oriented storage, because we would have a wider community open to help us in case of issues or maybe just because they are more popular. In our opinion, it is imperative to build systems that implement techniques specialized in querying RDF graphs in order to have a optimized fragment that can outperform traditional databases. Although it may be risky to build RDF systems that do not rely on techniques from relational databases, we believe that it is possible to find interesting results. In this work we studied an approach based on worst-case optimal join algorithms, but there are many other ideas to explore, for instance, to use specialized indexes for graphs, instead of using the classic tree-index approach.

And maybe a more important issue than reliability is to work on the adoption of Semantic Web technologies. A first part that should be addressed for this is to change the perception about the Semantic Web community. For instance, without being an outstanding graph-based engine, Neo4J can be considered a more probable choice than RDF systems for production environments, and in our opinion, this is mainly because they have constructed a wider community which implies that there exists more support in case of issues and questions. One way to increase the adoption of the Semantic Web is to build tools that facilitates interaction with the current techniques used in production environments. We believe that techniques (such as the one presented in this work) for integrating data sources on the web that currently are not reachable for the Semantic Web are important to improve the adoption of this ecosystem. This is the reason we propose a way to allow SPARQL queries to connect to JSON APIs, which are possibly the most important way for exposing data on the web nowadays. Moreover, we extended the notion of worst-case optimal join algorithms for processing queries that involve data sources available online. We think that it is not realistic to consider that all of the data that is currently available in the Web as an API will be available as an RDF Graph or as a SPARQL endpoint in a reasonable amount of time, so it is important to design tools that allow the Semantic Web to coexist with the current web, and at the same time, help to its adoption.

7.2. Looking Forward

Although we mention some key aspects to improve, we believe that the Semantic Web technologies have many opportunities to be an outstanding alternative to manage graph-based data. For instance, one aspect that we have to highlight is the query language language SPARQL, which has features that are not possible to find in another query languages, such as having a natural way for integrating distinct data sources. Moreover, this is a language that can easily support navigational features. We show that it is worth adding a fixpoint operator to SPARQL, which can bring an immediate impact in several areas. To begin with, it has been shown that a wide fragment of recursive SHACL constraints can be compiled into recursive SPARQL queries (Corman, Florenzano, Reutter, & Savković, 2019), and a similar result should hold for ShEx

constraints (Boneva, Gayo, & Prud'hommeaux, 2017). Another interesting direction is managing ontological knowledge. Indeed, it was shown that even a mild form of recursion is sufficient to capture RDFS entailment regimes (Pérez et al., 2010) or OWL2 QL entailment (Bischof, Krötzsch, Polleres, & Rudolph, 2014), and it stands open to which extent can rec-SPARQL help us capture more complex ontologies, and evaluate them efficiently.

When considering our recursive language as a way to do in-database graph analytics, we hope that our proposal ignites discussion on different ways for enriching SPARQL with graph analytics, and the best architecture to support them. A key research challenge relates to the optimization of SPARQAL procedures. We have investigated batch-at-a-time and also compilation into algebraic-like statements for evaluation within the database, but we still need support for indexing temporary graphs, and looking at whether or not traditional database optimization tasks are likewise suitable for optimizing SPARQAL procedures. In particular, we believe that our algebra of updates has strong relationships with linear algebra languages, and then, there is a way to compile our procedures to languages that can be handled by the GPU. This is one of the main lines to follow after this work, because it is a chance to develop a declarative language that would be able to compute graph patterns, navigational queries and analytics procedures with the GPU, which should be considerably faster than what we obtained with our prototype. We believe that this can be a huge contribution in order to make Semantic Web technologies stand out with respect to other approaches for handling large datasets.

And in terms of query processing, we also identify interesting lines of future research and open questions: first we have the potential benefits of other worst-case optimal join algorithms for SPARQL such as (Veldhuizen, 2014; Ngo et al., 2014; Nguyen et al., 2015; Khamis et al., 2016; Kalinsky et al., 2017; Ngo, 2018) and also, indexing structures specialized to handle graph-like data; we think that it is important to understand how to decide effective ways to optimize the variable order in the context of worst-case optimal join algorithms (Khamis et al., 2016; Abo Khamis et al., 2016); The next research line is, in our opinion, the most interesting one: to understand how

can we decrease the size of the indexes and the memory usage demanded by the worst-case optimal join algorithms. One possibility here is to study how we can benefit from using succinct data structures instead of classic tree-like indexes. And finally, as hinted before, we would like to study how we can put all of this together to integrate these algorithms with the other functionalities of SPARQL engines.

Finally, in terms of integrating data sources other than SPARQL endpoints, we plan to support formats other than JSON, and explore how to support our extension in public endpoints. It would be also interesting to test how issuing API calls in parallel affects the running times of different algorithms. Also, an orthogonal direction is to explore the potential for automatic entity resolution based on an API answer; that is, to transform the API information such as literals representing names of people or places back into IRIs. Thus, it would be possible to achieve a better integration between the API data and the RDF Graph. Also, we would like to explore how our extension can handle real-world issues related to APIs, such as how to manage APIs that paginate the results or that do not accept a high number of requests in short intervals of time.

References

Aberger, C. R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., & Ré, C. (2017). Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, *42*(4), 20.

Aberger, C. R., Tu, S., Olukotun, K., & Ré, C. (2016). Old techniques for new join algorithms: A case study in RDF processing. In *32nd IEEE international conference on data engineering workshops, ICDE workshops 2016, helsinki, finland, may 16-20, 2016* (pp. 97–102). IEEE Computer Society. Retrieved from `https://doi.org/10.1109/ICDEW.2016.7495625` doi: 10.1109/ICDEW.2016.7495625

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of databases*. Addison-Wesley.

Abo Khamis, M., Ngo, H. Q., & Rudra, A. (2016). FAQ: questions asked frequently. In *Principles of database systems (pods)* (pp. 13–28).

Additional material. (n.d.). https://cirojas.github.io/leapfrog-benchmark/. (Accessed on 2018-12-30)

Aluç, G., Hartig, O., Özsu, M. T., & Daudjee, K. (2014). Diversified stress testing of RDF data management systems. In *International semantic web conference (iswc)* (pp. 197–212). Springer.

Angles, R., Arenas, M., Barceló, P., Boncz, P. A., Fletcher, G. H. L., Gutierrez, C., … Voigt, H. (2018). G-CORE: A Core for Future Graph Query Languages. In *SIGMOD* (pp. 1421–1432).

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J. L., & Vrgoc, D. (2017). Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.*, *50*(5), 68:1–68:40.

Angles, R., & Gutierrez, C. (2016). The multiset semantics of sparql patterns. In *International semantic web conference* (pp. 20–36).

Angles, R., & Gutiérrez, C. (2016). Negation in SPARQL. In R. Pichler & A. S. da Silva (Eds.), *Proceedings of the 10th alberto mendelzon international workshop on foundations of data management, panama city, panama, may 8-10, 2016* (Vol. 1644). CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-1644/paper11.pdf

Anyanwu, K., & Sheth, A. (2003). ρ-Queries: enabling querying for semantic associations on the semantic web. In *12th international world wide web conference (www).*

The Apache Jena Manual. (2015). http://jena.apache.org.

Aranda, C. B., Arenas, M., & Corcho, Ó. (2011). Semantics and optimization of the SPARQL 1.1 federation extension. In *ESWC 2011* (pp. 1–15).

Aranda, C. B., Arenas, M., Corcho, Ó., & Polleres, A. (2013). Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Sem.*, *18*(1), 1–17.

Aranda, C. B., Hogan, A., Umbrich, J., & Vandenbussche, P. (2013). SPARQL web-querying infrastructure: Ready for action? In *International semantic web conference (iswc)* (pp. 277–293). Springer.

Aranda, C. B., Polleres, A., & Umbrich, J. (2014). Strategies for executing federated queries in SPARQL1.1. In *ISWC 2014* (pp. 390–405).

Aref, M., ten Cate, B., Green, T. J., Kimelfeld, B., Olteanu, D., Pasalic, E., ... Washburn, G. (2015). Design and implementation of the LogicBlox system. In *Sigmod international conference on management of data* (pp. 1371–1382).

Arenas, M., Gutierrez, C., & Pérez, J. (2010). On the semantics of sparql. In R. de Virgilio, F. Giunchiglia, & L. Tanca (Eds.), *Semantic web information management: A model-based perspective* (pp. 281–307). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/978-3-642-04329-1_13

Atre, M., Chaoji, V., Zaki, M. J., & Hendler, J. A. (2010). Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *World wide web (www)* (pp. 41–50).

Atserias, A., Grohe, M., & Marx, D. (2008). Size bounds and query plans for relational joins. In *Foundations of computer science (focs)* (pp. 739–748).

Atzori, M. (2014). Computing recursive SPARQL queries. In *Icsc* (pp. 258–259).

Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G. H. L., Lemay, A., & Advokaat, N. (2017). gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, *29*(4), 856–869.

Barbay, J., & Kenyon, C. (2002). Adaptive intersection and t-threshold problems. In *Symposium on discrete algorithms (soda)* (pp. 390–399).

Barceló, P., Fontaine, G., & Lin, A. W. (2013). Expressive path queries on graphs with data. In *Logic for programming, artificial intelligence, and reasoning* (pp. 71–85).

Barceló, P., Pérez, J., & Reutter, J. L. (2012). Relative Expressiveness of Nested Regular Expressions. In *Amw* (p. 180-195).

Battle, R., & Benson, E. (2008). Bridging the semantic web and web 2.0 with representational state transfer (REST). *J. Web Sem.*, *6*(1), 61–69.

Beek, W., Rietveld, L., Schlobach, S., & van Harmelen, F. (2016). Lod laundromat: Why the semantic web needs centralization (even if we don't like it). *IEEE Internet Computing*, *20*(2), 78–81.

Benedikt, M., Leblay, J., & Tsamoura, E. (2015). Querying with access patterns and integrity constraints. *PVLDB*, *8*(6), 690–701.

Berners-Lee, T. (2012). *5-stars Open Data.* `https://5stardata.info/en/`.

Bischof, S., Krötzsch, M., Polleres, A., & Rudolph, S. (2014). Schema-agnostic query rewriting in sparql 1.1. In *International semantic web conference* (pp. 584–600).

Bizer, C., & Schultz, A. (2009). The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, *5*(2), 1–24.

Bonatti, P. A., Decker, S., Polleres, A., & Presutti, V. (2018). Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web. *Dagstuhl Reports*, *8*(9), 29–111.

Boneva, I., Gayo, J. E. L., & Prud'hommeaux, E. G. (2017). Semantics and validation of shapes schemas for rdf. In *International semantic web conference* (pp. 104–120).

Bonifati, A., Martens, W., & Timm, T. (2017). An analytical study of large SPARQL query logs. *CoRR*, *abs/1708.00363*. Retrieved from `http://arxiv.org/abs/1708.00363`

Bourhis, P., Krötzsch, M., & Rudolph, S. (2014). How to best nest regular path queries. In *Informal proceedings of the 27th international workshop on description logics.*

Bray, T. (2014). The javascript object notation (json) data interchange format.

Brijder, R., Geerts, F., den Bussche, J. V., & Weerwag, T. (2018). On the Expressive Power of Query Languages for Matrices. In *International conference on database theory (icdt)* (pp. 10:1–10:17). Schloss Dagstuhl.

Calì, A., & Martinenghi, D. (2008). Querying data under access limitations. In *ICDE 2008* (pp. 50–59).

Charalambidis, A., Troumpoukis, A., & Konstantopoulos, S. (2015). Semagrow: Optimizing federated sparql queries. In *Proceedings of the 11th international conference on semantic systems* (pp. 121–128).

Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., & Muthukrishnan, S. (2015). One trillion edges: Graph processing at facebook-scale. *PVLDB*, *8*(12), 1804–1815.

Churchill, A., Biderman, S., & Herrick, A. (2019). Magic: The gathering is turing complete. *arXiv preprint arXiv:1904.09828*.

Consens, M., & Mendelzon, A. (1990). Graphlog: A visual formalism for real life recursion. In *9th acm symposium on principles of database systems (pods)* (pp. 404–416).

Corby, O., Faron-Zucker, C., & Gandon, F. (2017). LDScript: A Linked Data Script Language. In *International semantic web conference (iswc)* (pp. 208–224). Springer.

Corman, J., Florenzano, F., Reutter, J. L., & Savković, O. (2019). Validating shacl constraints over a sparql endpoint. In *International semantic web conference (to appear).*

DeLorimier, M., Kapre, N., Mehta, N., Rizzo, D., Eslick, I., Rubin, R., … DeHon, A. (2006). GraphStep: A System Architecture for Sparse-Graph Algorithms. In *IEEE symposium on field-programmable custom computing machines (fccm)* (pp. 143–151). IEEE Computer Society.

Demaine, E. D., López-Ortiz, A., & Munro, J. I. (2000). Adaptive set intersections, unions, and differences. In *Symposium on discrete algorithms (soda).*

Dimou, A., Sande, M. V., Colpaert, P., Verborgh, R., Mannens, E., & de Walle, R. V. (2014). RML: A generic language for integrated RDF mappings of heterogeneous data. In *LDOW.*

Fafalios, P., & Tzitzikas, Y. (2015). SPARQL-LD: a SPARQL extension for fetching and querying linked data. In *ISWC demos.*

Fafalios, P., Yannakis, T., & Tzitzikas, Y. (2016). Querying the web of data with SPARQL-LD. In *TPDL 2016* (pp. 175–187).

Fionda, V., & Pirrò, G. (2017). Explaining graph navigational queries. In *European semantic web conference* (pp. 19–34).

Fionda, V., Pirrò, G., & Consens, M. P. (2015). Extended property paths: Writing more sparql queries in a succinct way. In *Twenty-ninth aaai conference on artificial intelligence.*

Fionda, V., Pirrò, G., & Gutierrez, C. (2015). N auti lod: A formal language for the web of data graph. *ACM Transactions on the Web (TWEB)*, *9*(1), 5.

Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., ... Taylor, A. (2018). Cypher: An Evolving Query Language for Property Graphs. In *International conference on management of data (sigmod)* (pp. 1433–1445). ACM.

Gaifman, H., Mairson, H., Sagiv, Y., & Vardi, M. Y. (1993). Undecidable optimization problems for database logic programs. *Journal of the ACM (JACM)*, *40*(3), 683–713.

Galiegue, F., & Zyp, K. (2013). Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*.

Galkin, M., Endris, K. M., Acosta, M., Collarana, D., Vidal, M., & Auer, S. (2017). SMJoin: A Multi-way Join Operator for SPARQL Queries. In *International conference on semantic systems (SEMANTICS)* (pp. 104–111).

Geerts, F. (2019). On the Expressive Power of Linear Algebra on Graphs. In *International conference on database theory (icdt)* (pp. 7:1–7:19). Schloss Dagstuhl.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., & Guestrin, C. (2012). PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation, OSDI 2012, hollywood, ca, usa, october 8-10, 2012* (pp. 17–30). USENIX Association.

Gottlob, G., Lee, S. T., Valiant, G., & Valiant, P. (2012). Size and treewidth bounds for conjunctive queries. *J. ACM*, *59*(3), 16:1–16:35.

Gray, J. (1992). *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann Publishers Inc.

Green, T. J., Huang, S. S., Loo, B. T., & Zhou, W. (2013). Datalog and recursive query processing. *Foundations and Trends in Databases*, *5*(2), 105–195.

Grohe, M. (2013). Bounds and algorithms for joins via fractional edge covers. In V. Tannen, L. Wong, L. Libkin, W. Fan, W. Tan, & M. P. Fourman (Eds.), *In search of elegance in the theory and practice of computation - essays dedicated to peter buneman* (Vol. 8000, pp. 321–338). Springer.

Gubichev, A., Bedathur, S. J., & Seufert, S. (2013). Sparqling kleene: fast property paths in RDF-3X. In *GRADES.*

Guo, Y., Pan, Z., & Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, *3*(2-3), 158–182.

Harris, S., & Seaborne, A. (2013). SPARQL 1.1 query language. *W3C Recommendation*, *21*.

Harth, A., & Decker, S. (2005). Optimized Index Structures for Querying RDF from the Web. In *Latin american web congress (la-web 2005)* (pp. 71–80).

Hartig, O., Bizer, C., & Freytag, J.-C. (2009). Executing sparql queries over the web of linked data. In *International semantic web conference* (pp. 293–309).

Hernández, D., Hogan, A., & Krötzsch, M. (2015). Reifying rdf: What works well with wikidata? *SSWS@ ISWC*, *1457*, 32–47.

Hernández, D., Hogan, A., Riveros, C., Rojas, C., & Zerega, E. (2016). Querying wikidata: Comparing sparql, relational and graph databases. In *International semantic web conference* (pp. 88–103).

Hernández, D. (2020). *A core sparql fragment.*

Hogan, A. (2017). Canonical forms for isomorphic and equivalent RDF graphs: algorithms for leaning and labelling blank nodes. *ACM TWEB*, *11*(4), 1–62.

Hogan, A. (2020). The semantic web: Two decades on. *Semantic Web*, *11*(1), 169–185. Retrieved from `https://doi.org/10.3233/SW-190387` doi: 10.3233/SW-190387

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., ... Zimmermann, A. (2020). Knowledge Graphs. *CoRR*, *abs/2003.02320*. Retrieved from https://arxiv.org/abs/2003.02320

Hogan, A., Reutter, J. L., & Soto, A. (2020). In-database graph analytics with recursive SPARQL. In *The semantic web - ISWC 2020 - 19th international semantic web conference, athens, greece, november 2-6, 2020, proceedings, part I* (Vol. 12506, pp. 511–528). Springer. Retrieved from https://doi.org/10.1007/978-3-030-62419-4_29 doi: 10.1007/978-3-030-62419-4_29

Hogan, A., Riveros, C., Rojas, C., & Soto, A. (2019). A worst-case optimal join algorithm for SPARQL. In *The semantic web - ISWC 2019 - 18th international semantic web conference, auckland, new zealand, october 26-30, 2019, proceedings, part I* (Vol. 11778, pp. 258–275). Springer. Retrieved from https://doi.org/10.1007/978-3-030-30793-6_15 doi: 10.1007/978-3-030-30793-6_15

Hutchison, D., Howe, B., & Suciu, D. (2017). LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation. In *ACM SIGMOD workshop on algorithms and systems for mapreduce and beyond (beyondmr@sigmod)* (pp. 2:1–2:10). ACM.

IETF. (2012). *URI Template*. https://tools.ietf.org/html/rfc6570.

Junemann, M., Reutter, J. L., Soto, A., & Vrgoc, D. (2016). Incorporating API data into SPARQL query answers. In *Proceedings of the ISWC 2016 posters & demonstrations track co-located with 15th international semantic web conference (ISWC 2016), kobe, japan, october 19, 2016* (Vol. 1690). CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-1690/paper77.pdf

Kalinsky, O., Etsion, Y., & Kimelfeld, B. (2017). Flexible Caching in Trie Joins. In *International conference on extending database technology (edbt)* (pp. 282–293). OpenProceedings.org.

Kalinsky, O., Mishali, O., Hogan, A., Etsion, Y., & Kimelfeld, B. (2018). Efficiently charting RDF. *CoRR*, *abs/1811.10955*. Retrieved from http://arxiv.org/abs/1811.10955

Kaminski, M., & Kostylev, E. V. (2016). Beyond well-designed sparql. In *19th international conference on database theory (icdt 2016).*

Kankanamge, C., Sahu, S., Mhedbhi, A., Chen, J., & Salihoglu, S. (2017). Graphflow: An active graph database. In *Proceedings of the 2017 ACM international conference on management of data, SIGMOD conference 2017, chicago, il, usa, may 14-19, 2017* (pp. 1695–1698). ACM. Retrieved from `https://doi.org/10.1145/3035918.3056445` doi: 10.1145/3035918.3056445

Khamis, M. A., Ngo, H. Q., Ré, C., & Rudra, A. (2016). Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems (TODS)*, *41*(4), 22.

Kobayashi, N., Ishii, M., Takahashi, S., Mochizuki, Y., Matsushima, A., & Toyoda, T. (2011). Semantic-json. *Nucleic Acids Research*, *39*, 533–540.

Kochut, K. J., & Janik, M. (2007). Sparqler: Extended sparql for semantic association discovery. In *The semantic web: Research and applications* (pp. 145–159). Springer.

Kostylev, E. V., Reutter, J. L., Romero, M., & Vrgoc, D. (2015). SPARQL with Property Paths. In *International semantic web conference (iswc)* (pp. 3–18). Springer.

Kostylev, E. V., Reutter, J. L., & Ugarte, M. (2015). CONSTRUCT queries in SPARQL. In *ICDT* (pp. 212–229).

Krepska, E., Kielmann, T., Fokkink, W., & Bal, H. E. (2011). HipG: parallel processing of large-scale graphs. *Operating Systems Review*, *45*(2), 3–13.

LDBC. (2019). *Graphalytics Benchmark Suite.* (`https://graphalytics.org/`)

Libkin, L. (2004). *Elements of finite model theory*. Springer.

Libkin, L., Reutter, J. L., Soto, A., & Vrgoč, D. (2018). Trial: A navigational algebra for RDF triplestores. *ACM Trans. Database Syst.*, *43*(1), 5:1–5:46.

Linked movie database. (n.d.). http://linkedmdb.org/.

Lisena, P., Meroño-Peñuela, A., Kuhn, T., & Troncy, R. (2019). Easy web api development with sparql transformer. In *International semantic web conference* (pp. 454–470).

Low, Y., Gonzalez, J. E., Kyrola, A., Bickson, D., Guestrin, C., & Hellerstein, J. M. (2014). Graphlab: A new framework for parallel machine learning. *CoRR*, *abs/1408.2041*. Retrieved from `http://arxiv.org/abs/1408.2041`

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *ACM SIGMOD international conference on management of data (sigmod)* (pp. 135–146). ACM Press.

Malyshev, S., Krötzsch, M., González, L., Gonsior, J., & Bielefeldt, A. (2018). Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *International semantic web conference (iswc)* (pp. 376–394). Springer.

Meroño-Peñuela, A., & Hoekstra, R. (2017). Automatic query-centric api for routine access to linked data. In *International semantic web conference* (pp. 334–349).

Miller, J. J. (2013). Graph Database Applications and Concepts with Neo4j. In *Southern association for information systems conference (sais).* AIS eLibrary.

Montoya, G., Skaf-Molli, H., & Hose, K. (2017). The odyssey approach for optimizing federated sparql queries. In *International semantic web conference* (pp. 471–489).

Montoya, G., Vidal, M., & Acosta, M. (2012). A heuristic-based approach for planning federated SPARQL queries. In *COLD 2012.*

Montoya, G., Vidal, M., Corcho, Ó., Ruckhaus, E., & Aranda, C. B. (2012). Benchmarking federated SPARQL query engines: Are existing testbeds enough? In *ISWC 2012* (pp. 313–324).

Mosser, M., Pieressa, F., Reutter, J. L., Soto, A., & Vrgoc, D. (2018). Querying apis with SPARQL: language and worst-case optimal algorithms. In *The semantic web - 15th international conference, ESWC 2018, heraklion, crete, greece, june 3-7, 2018, proceedings* (Vol. 10843, pp. 639–654). Springer. Retrieved from https://doi.org/10.1007/978-3-319-93417-4_41 doi: 10.1007/978-3-319-93417-4_41

Mosser, M., Pieressa, F., Reutter, J. L., Soto, A., & Vrgoc, D. (2019). Querying apis with SPARQL. In A. Hogan & T. Milo (Eds.), *Proceedings of the 13th alberto mendelzon international workshop on foundations of data management, asunción, paraguay, june 3-7, 2019* (Vol. 2369). CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-2369/short05.pdf

Mosser, M., Pieressa, F., Reutter, J. L., Soto, A., & Vrgoc, D. (2020). *Online version of querying apis with sparql (journal version).* https://www.sciencedirect.com/science/article/abs/pii/S0306437920301125. (Accessed on 2021-01-10)

Motik, B., Nenov, Y., Piro, R., Horrocks, I., & Olteanu, D. (2014). Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *AAAI*.

Müller, H., Cabral, L., Morshed, A., & Shu, Y. (2013). From restful to SPARQL: A case study on generating semantic sensor data. In *Iswc 2013* (pp. 51–66).

Neumann, T., & Weikum, G. (2008). RDF-3X: a RISC-style engine for RDF. *PVLDB*, *1*(1), 647–659.

Ngo, H. Q. (2018). Worst-case optimal join algorithms: Techniques, results, and open problems. In *Principles of database systems (pods)* (pp. 111–124).

Ngo, H. Q., Nguyen, D. T., Re, C., & Rudra, A. (2014). Beyond worst-case analysis for joins with minesweeper. In *Principles of database systems (pods)* (pp. 234–245).

Ngo, H. Q., Porat, E., Ré, C., & Rudra, A. (2012). Worst-case optimal join algorithms. In *Principles of database systems (pods)* (pp. 37–48).

Ngo, H. Q., Porat, E., Ré, C., & Rudra, A. (2012). Worst-case optimal join algorithms. In *PODS 2012* (pp. 37–48).

Ngo, H. Q., Ré, C., & Rudra, A. (2013). Skew strikes back: New developments in the theory of join algorithms. *arXiv preprint arXiv:1310.3314*.

Nguyen, D., Aref, M., Bravenboer, M., Kollias, G., Ngo, H. Q., Ré, C., & Rudra, A. (2015). Join processing for graph patterns: An old dog with new tricks. In *Grades* (p. 2).

Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, *25*(3-4), 241–273. Retrieved from `https://doi.org/10.1023/A:1018930122475` doi: 10.1023/A:1018930122475

Open Link Virtuoso. (2015). http://virtuoso.openlinksw.com/.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). *The PageRank citation ranking: Bringing order to the Web* (Tech. Rep.). Stanford InfoLab.

Pérez, J., Arenas, M., & Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, *34*(3).

Pérez, J., Arenas, M., & Gutierrez, C. (2010). nSPARQL: A navigational language for RDF. *J. Web Sem.*, *8*(4), 255–270.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., & Vrgoč, D. (2016). Foundations of JSON Schema. In *Www 2016* (pp. 263–273).

PostgreSQL documentation. (n.d.). `http://www.postgresql.org/docs/current/interactive/queries-with.html`.

Prud'hommeaux, E., & Buil-Aranda, C. (2013). SPARQL 1.1 Federated Query. *W3C Recommendation*, *21*.

Ramakrishnan, R., & Gehrke, J. (2000). *Database management systems*. McGraw Hill.

Reutter, J. L., Romero, M., & Vardi, M. Y. (2017). Regular queries on graph databases. *Theory of Computing Systems*, *61*(1), 31–83.

Reutter, J. L., Soto, A., & Vrgoc, D. (2020). *Preprint of recursion in sparql (journal version).* http://www.semantic-web-journal.net/content/recursion-sparql-0. (Accessed on 2021-01-10)

Reutter, J. L., Soto, A., & Vrgoč, D. (2015). Recursion in SPARQL. In *The semantic web - ISWC 2015 - 14th international semantic web conference, bethlehem, pa, usa, october 11-15, 2015, proceedings, part I* (pp. 19–35).

Rietveld, L., & Hoekstra, R. (2013). YASGUI: not just another SPARQL client. In *ESWC 2013* (pp. 78–86).

Rodriguez, M. A. (2015). The Gremlin graph traversal machine and language. In *Symposium on database programming languages (dbpl)* (pp. 1–10). ACM.

Saleem, M., Potocki, A., Soru, T., Hartig, O., & Ngomo, A.-C. N. (2018). Costfed: Cost-based query optimization for sparql endpoint federation. *Procedia Computer Science*, *137*, 163–174.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The Graph Neural Network Model. *IEEE Trans. Neural Networks*, *20*(1), 61–80.

Schmidt, M., Hornung, T., Lausen, G., & Pinkel, C. (2009). Spˆ 2bench: a sparql performance benchmark. In *Data engineering, 2009. icde'09. ieee 25th international conference on* (pp. 222–233).

Senanayake, U., Piraveenan, M., & Zomaya, A. (2015, 08). The Pagerank-Index: Going beyond Citation Counts in Quantifying Scientific Impact of Researchers. *PLOS ONE*, *10*(8), 1-34.

Shao, B., Wang, H., & Li, Y. (2013). Trinity: a distributed graph engine on a memory cloud. In *SIGMOD international conference on management of data (sigmod)* (pp. 505–516). ACM.

Stringer, B., Meroño-Peñuela, A., Loizou, A., Abeln, S., & Heringa, J. (2015). Scry: Enabling quantitative reasoning in sparql queries. In *Swat4ls* (pp. 214–215).

Stutz, P., Strebel, D., & Bernstein, A. (2016). Signal/Collect12. *Semantic Web Journal*, *7*(2), 139–166.

Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications.

Thompson, B. B., Personick, M., & Cutcher, M. (2014). The Bigdata®RDF Graph Database. In *Linked data management.* (pp. 193–237).

Urzua, V., & Gutiérrez, C. (2019). Linear Recursion in G-CORE. In *Alberto mendelzon international workshop on foundations of data management (amw)* (Vol. 2369). CEUR-WS.org.

Vardi, M. Y. (1995). On the complexity of bounded-variable queries. In *Pods* (Vol. 95, pp. 266–276).

Veldhuizen, T. L. (2014). Leapfrog Triejoin: A simple, worst-case optimal join algorithm. In *Icdt* (pp. 96–106).

Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., ... Colpaert, P. (2016). Triple pattern fragments: a low-cost knowledge graph interface for the web. *Journal of Web Semantics*, *37*, 184–206.

Vrandečić, D., & Krötzsch, M. (2014). Wikidata: A free collaborative knowledgebase. *Comm. ACM*, *57*, 78-85.

Weiss, C., Karras, P., & Bernstein, A. (2008). Hexastore: sextuple indexing for semantic web data management. *PVLDB*, *1*(1), 1008–1019.

Wikidata query service. (n.d.). `https://query.wikidata.org/`.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2019). A Comprehensive Survey on Graph Neural Networks. *CoRR*, *abs/1901.00596*.

Xin, R. S., Gonzalez, J. E., Franklin, M. J., & Stoica, I. (2013). GraphX: a resilient distributed graph system on Spark. In *International workshop on graph data management experiences and systems (grades).* ACM Press.

Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., & Stoica, I. (2013). Shark: SQL and rich analytics at scale. In *Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2013, new york, ny, usa, june 22-27, 2013* (pp. 13–24). ACM Press.

Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019). How Powerful are Graph Neural Networks? In *International conference on learning representations (ICLR).* OpenReview.net.

YAGO: A High-Quality Knowledge Base. (n.d.). `http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/`.

Yakovets, N., Godfrey, P., & Gryz, J. (2013). Evaluation of SPARQL property paths via recursive SQL. In *Amw.*

Yakovets, N., Godfrey, P., & Gryz, J. (2015). WAVEGUIDE: evaluating SPARQL property path queries. In *EDBT 2015* (pp. 525–528).

Yannakis, T., Fafalios, P., & Tzitzikas, Y. (2018). Heuristics-based query reordering for federated queries in sparql 1.1 and sparql-ld. *arXiv preprint arXiv:1810.09780*.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., … Stoica, I. (2016). Apache Spark: a unified engine for big data processing. *Commun. ACM*, *59*(11), 56–65.

Zeng, K., Yang, J., Wang, H., Shao, B., & Wang, Z. (2013). A Distributed Graph Engine for Web Scale RDF Data. *PVLDB*, *6*(4), 265–276.

APPENDIX

APPENDIX A. ONLINE RESOURCES

Here we present a list with online resources related to this thesis. It is possible to find the previous publications and the source code of our projects in `https://adriansoto.cl/research`. The list of concrete resources is available below.

Recursion in SPARQL resources

In `https://adriansoto.cl/RecSPARQL/` it is possible to find all the resources related to Recursive SPARQL:

- There is a link to the GitHub with the source code.
- There are instructions for loading all the datasets and running the queries.

In-database Graph Analytics with SPARQL

The supplementary material for Chapter can be found at `https://adriansoto.cl/files/SPARQAL.zip`.

A Worst-case Optimal Join Algorithm for SPARQL resources

In `https://cirojas.github.io/leapfrog-benchmark/` it is possible to find the source code and the instructions to run the code. Also in `https://zenodo.org/record/4035223#.YAIJaZP0lhE` it is possible to find the details for the Wikidata Benchmark, such as the concrete queries and the dataset.

Querying APIs with SPARQL

The supplementary material for the chapter can be found at `https://github.com/alanezz/IS-SPAPI`.

APPENDIX B. RECURSIVE SPARQL QUERIES

Here we present some of the queries that we ran throughout this work. Note that here we declare the queries using the clauses `FROM` and `FROM NAMED`. This is because some systems need the declaration of the graphs that are used in `GRAPH` clauses.

Queries from Subsection 3.5.1

The query Q_A is represented by the following recursive query:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS{
    CONSTRUCT {<http://data.linkedmdb.org/resource/actor/29539>
    <http://relationship.com/collab> ?act}
    FROM NAMED <http://db.ing.puc.cl/temp>
  FROM <Quad.defaultGraphIRI>
  WHERE {
    {?mov <http://data.linkedmdb.org/resource/movie/actor>
    <http://data.linkedmdb.org/resource/actor/29539> .
    ?mov <http://data.linkedmdb.org/resource/movie/actor> ?act}
  UNION {
    {?mov <http://data.linkedmdb.org/resource/movie/actor> ?act1} .
    {?mov <http://data.linkedmdb.org/resource/movie/actor> ?act} .
    GRAPH <http://db.ing.puc.cl/temp>
    {<http://data.linkedmdb.org/resource/actor/29539>
    <http://relationship.com/collab> ?act1}}
    }
    }
  SELECT ?z FROM NAMED <http://db.ing.puc.cl/temp>
  WHERE {GRAPH <http://db.ing.puc.cl/temp>
  {<http://data.linkedmdb.org/resource/actor/29539>
  <http://relationship.com/collab> ?z}}
```

The following is the formulation of the query Q_B:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS
  {
  CONSTRUCT {<http://data.linkedmdb.org/resource/actor/29539> ?dir ?act}
  FROM NAMED <http://db.ing.puc.cl/temp>
  FROM <Quad.defaultGraphIRI>
  WHERE
```

```
7    {
8    {?mov <http://data.linkedmdb.org/resource/movie/actor>
9    <http://data.linkedmdb.org/resource/actor/29539> .
10   ?mov <http://data.linkedmdb.org/resource/movie/actor> ?act .
11   ?mov <http://data.linkedmdb.org/resource/movie/director> ?dir}
12   UNION
13   {{?mov <http://data.linkedmdb.org/resource/movie/director> ?dir} .
14   {?mov <http://data.linkedmdb.org/resource/movie/actor> ?act1} .
15   {?mov <http://data.linkedmdb.org/resource/movie/actor> ?act} .
16   GRAPH <http://db.ing.puc.cl/temp>
17   {<http://data.linkedmdb.org/resource/actor/29539> ?dir ?act1}}
18     }
19     }
20   SELECT ?y ?z FROM NAMED <http://db.ing.puc.cl/temp>
21   WHERE {GRAPH <http://db.ing.puc.cl/temp>
22   {<http://data.linkedmdb.org/resource/actor/29539> ?y ?z}}
```

The following is the formulation of the query Q_C:

```
1  WITH RECURSIVE http://db.ing.puc.cl/temp AS
2    {
3    CONSTRUCT {<http://data.linkedmdb.org/resource/actor/29539>
4    <http://relationship.com/collab> ?act}
5    FROM NAMED <http://db.ing.puc.cl/temp>
6    FROM <Quad.defaultGraphIRI>
7    WHERE
8    {
9    {?mov <http://data.linkedmdb.org/resource/movie/actor>
10   <http://data.linkedmdb.org/resource/actor/29539> .
11   ?mov <http://data.linkedmdb.org/resource/movie/actor> ?act .
12   ?mov <http://data.linkedmdb.org/resource/movie/director> ?dir .
13   ?dir <http://data.linkedmdb.org/resource/movie/director_name> ?x .
14   ?y <http://data.linkedmdb.org/resource/movie/actor_name> ?x}
15   UNION
16   {{?mov <http://data.linkedmdb.org/resource/movie/director> ?dir} .
17   {?dir <http://data.linkedmdb.org/resource/movie/director_name> ?x} .
18   {?y <http://data.linkedmdb.org/resource/movie/actor_name> ?x} .
19   {?mov <http://data.linkedmdb.org/resource/movie/actor> ?act1} .
20   {?mov <http://data.linkedmdb.org/resource/movie/actor> ?act} .
21   GRAPH <http://db.ing.puc.cl/temp>
22   {<http://data.linkedmdb.org/resource/actor/29539>
```

```
  <http://relationship.com/collab> ?act1}}
  }
  }
  SELECT ?z FROM NAMED <http://db.ing.puc.cl/temp>
  WHERE {GRAPH <http://db.ing.puc.cl/temp>
  {<http://data.linkedmdb.org/resource/actor/29539>
  <http://relationship.com/collab> ?z}}
```

The following is the formulation of the query Q_D:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS
{
CONSTRUCT {
  <http://yago-knowledge.org/resource/Berlin>
  <http://yago-knowledge.org/resource/isLocatedIn> ?x1
  }
  FROM NAMED <http://db.ing.puc.cl/temp>
  FROM <urn:x-arq:DefaultGraph>
  WHERE {
    {
      <http://yago-knowledge.org/resource/Berlin>
      <http://yago-knowledge.org/resource/isLocatedIn> ?x1
    }
    UNION
    {
      ?y <http://yago-knowledge.org/resource/isLocatedIn> ?x1 .
      GRAPH <http://db.ing.puc.cl/temp> {
        <http://yago-knowledge.org/resource/Berlin>
        <http://yago-knowledge.org/resource/isLocatedIn> ?y
      }
    }
  }
}
SELECT * FROM NAMED <http://db.ing.puc.cl/temp>
FROM <urn:x-arq:DefaultGraph>
WHERE {
  ?z <http://yago-knowledge.org/resource/dealsWith> ?v .
  GRAPH <http://db.ing.puc.cl/temp> {
  ?x ?y ?z }
}
```

The following is the formulation of the query Q_E:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS
{
CONSTRUCT {?x0 <http://yago-knowledge.org/resource/isMarriedTo> ?x1}
FROM NAMED <http://db.ing.puc.cl/temp>
FROM <urn:x-arq:DefaultGraph>
WHERE {
{ ?x0 <http://yago-knowledge.org/resource/isMarriedTo> ?x1 .
  ?x1 <http://yago-knowledge.org/resource/owns> ?y }
UNION
{ ?x0 <http://yago-knowledge.org/resource/isMarriedTo> ?y .
  GRAPH <http://db.ing.puc.cl/temp> {
    ?y <http://yago-knowledge.org/resource/isMarriedTo> ?x1
  }
}
}
}
WITH RECURSIVE http://db.ing.puc.cl/temp2 AS
{
CONSTRUCT {
  ?x0 <http://yago-knowledge.org/resource/isLocatedIn>
  <http://yago-knowledge.org/resource/United_States>
}
FROM NAMED <http://db.ing.puc.cl/temp2>
FROM <urn:x-arq:DefaultGraph>
WHERE {
{
  ?x0 <http://yago-knowledge.org/resource/isLocatedIn>
  <http://yago-knowledge.org/resource/United_States>
}
UNION
{ ?x0 <http://yago-knowledge.org/resource/isLocatedIn> ?y .
  GRAPH <http://db.ing.puc.cl/temp2> {
    ?y <http://yago-knowledge.org/resource/isLocatedIn>
    <http://yago-knowledge.org/resource/United_States>
  }
}
}
}
SELECT *
```

```
FROM NAMED <http://db.ing.puc.cl/temp>
FROM NAMED <http://db.ing.puc.cl/temp2>
FROM <urn:x-arq:DefaultGraph>
WHERE {
  ?z1 <http://yago-knowledge.org/resource/owns> ?x2 .
  GRAPH <http://db.ing.puc.cl/temp> { ?x1 ?y1 ?z1 } .
  GRAPH <http://db.ing.puc.cl/temp2> { ?x2 ?y2 ?z2 }
}
```

Queries from Subsection 3.5.2

The following are the queries generated by the GMark benchmark:

```
PREFIX : <http://example.org/gmark/>
SELECT * WHERE { ?x0 (:p16/^:p16) ?x1 . ?x1 (:p16/^:p16)* ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 ((:p23/^:p23)|(:p25/^:p23)) ?x1 .
?x1 ((:p23/^:p23)|(:p25/^:p23))* ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 (:p25/^:p25) ?x1 . ?x1 (:p25/^:p25)* ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 (^:p22/:p16)* ?x1 . ?x1 (^:p19/:p20) ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 (:p0/:p22/^:p23) ?x1 . ?x1 (:p24/^:p24)* ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 ((:p23/^:p23)|(:p25/^:p23)) ?x1 .
?x1 ((:p23/^:p23)|(:p25/^:p23))* ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 ((^:p15/:p18)|(^:p18/:p15))* ?x1 .
?x1 ((^:p15/:p8/^:p13)|(^:p15/:p8/^:p14)) ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 ((:p21/^:p21)|(:p21/^:p22)) ?x1 .
?x1 ((:p21/^:p21)|(:p21/^:p22))* ?x2 .
?x2 (:p16/^:p21)* ?x3 }
```

```
29
30 PREFIX : <http://example.org/gmark/>
31 SELECT *  WHERE { ?x0 (^:p23/:p24) ?x1 .
32 ?x1 (^:p23/:p24)* ?x4 .
33 ?x0 (^:p17/:p21)* ?x2 .
34 ?x0 (^:p25/:p25)* ?x3 }
35
36 PREFIX : <http://example.org/gmark/>
37 SELECT *  WHERE { ?x0 (:p16/^:p23) ?x1 .
38 ?x1 (:p24/^:p24)* ?x2 .
39 ?x2 (:p23/^:p23)* ?x3 }
```

The same queries written in Recursive SPARQL can be found at `https://alanezz.github.io/RecSPARQL`.

Queries from Subsection 3.4.3

The following is SPARQL rewriting of the query Q1 computing Bacon number of length at most 5:

```
1  SELECT ?act WHERE{{?mov
2  <http://data.linkedmdb.org/resource/movie/actor>
3  <http://data.linkedmdb.org/resource/actor/29539> .
4  ?mov <http://data.linkedmdb.org/resource/movie/actor> ?act}
5
6  UNION { ?mov <http://data.linkedmdb.org/resource/movie/actor>
7  <http://data.linkedmdb.org/resource/actor/29539> .
8  ?mov <http://data.linkedmdb.org/resource/movie/actor> ?act2  .
9  ?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
10 ?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act}
11
12 UNION {?mov <http://data.linkedmdb.org/resource/movie/actor>
13 <http://data.linkedmdb.org/resource/actor/29539> .
14 ?mov <http://data.linkedmdb.org/resource/movie/actor> ?act3  .
15 ?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act3 .
16 ?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
17 ?mov3 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
18 ?mov3 <http://data.linkedmdb.org/resource/movie/actor> ?act  }
19
20 UNION {?mov <http://data.linkedmdb.org/resource/movie/actor>
```

```
21  <http://data.linkedmdb.org/resource/actor/29539> .
22  ?mov <http://data.linkedmdb.org/resource/movie/actor> ?act4  .
23  ?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act4 .
24  ?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act3 .
25  ?mov3 <http://data.linkedmdb.org/resource/movie/actor> ?act3 .
26  ?mov3 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
27  ?mov4 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
28  ?mov4 <http://data.linkedmdb.org/resource/movie/actor> ?act  }}
```

Other rewritings are similar and can be found at `https://alanezz.github.io/RecSPARQL`.

Queries over the Wikidata Endpoint

- **Q1**: Sub-properties of property `P276`:

```
1  SELECT ?subProperties WHERE {
2    ?subProperties wdt:P1647* wd:P276
3  }
```

- **Q2**: Horse lineages:

```
1  SELECT ?horse WHERE
2  {
3    ?horse wdt:P31/wdt:P279* wd:Q726
4  }
```

- **Q3**: Parent taxon of the Blue Whale:

```
1  SELECT ?taxon WHERE {
2    wd:Q42196 wdt:P171* ?taxon
3  }
```

- **Q4**: Metro stations reachable from Palermo Station in *Metro de Buenos Aires*:

```
1  SELECT ?metro WHERE
2  {
3    wd:Q3296629 wdt:P197* ?metro
4  }
```

- **Q5**: Actors with finite Bacon number:

```
1 SELECT ?actor WHERE
2 {
3   ?actor (^wdt:P161/wdt:P161)*
4     wd:Q3454165
5 }
```

Number of outputs for the bigger graphs in GMark

The number of outputs for the bigger graphs can be found in Figures B.1 and B.2.

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
RecSPARQL	50190	8153	3188	7	345	8153	6116	308	4	2134
Jena	208437	181043	178465	409	1547	181043	16730	189628	179168	11022
Virtuoso	3624482	-	5256	409	1561	-	-	-	968	13002

FIGURE B.1. Number of outputs for the GMark queries over the graph G_2.

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
RecSPARQL	74967	12015	4719	17	533	12015	16040	487	4	2942
Jena	311559	270506	266777	766	2674	270506	-	-	-	15951
Virtuoso	-	-	7743	766	2716	-	-	-	1384	18726

FIGURE B.2. Number of outputs for the GMark queries over the graph G_3.

Results of Recursive Queries in PSQL

As a reference, we present the results for the running times of the Yago queries in PostgreSQL.

Q1	Q2	Q3	Q4	Q5
60	0.025	739	136	265

APPENDIX C. SERVICE-TO-API QUERIES

Transformed Berlin Benchmark Queries

- Q1:

```
PREFIX ex: <http://example.org/>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type %ProductType% .
  SERVICE <http://localhost:5000/features/{label}>{
    ($.values[*]) AS (?v1)
  }
  FILTER(?v1 = %ProductFeature1%)
  SERVICE <http://localhost:5000/features/{label}>{
    ($.values[*]) AS (?v2)
  }
  FILTER(?v2 = %ProductFeature2%)
  ?product bsbm:productPropertyNumeric1 ?v3
  FILTER (?v3 > 500)
}
```

- Q2:

```
PREFIX ex: <http://example.org/>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
  %Product% rdfs:label ?label .
  %Product% rdfs:comment ?comment .
  %Product% bsbm:producer ?p .
  ?p rdfs:label ?producer .
  SERVICE <http://localhost:5000/features/{label}>{
    ($.values[*]) AS (?f)
  }
  SERVICE <http://localhost:5000/textual/{label}>{
    ($.p1, $.p2, $.p3) AS
    (?propertyTextual1, ?propertyTextual2, ?propertyTextual3)
  }
  SERVICE <http://localhost:5000/numeric/{label}>{
    ($.p1, $.p2) AS
      (?propertyNumeric1, ?propertyNumeric2)
  }
  %Product% bsbm:productPropertyTextual4 ?p4 .
  %Product% bsbm:productPropertyTextual5 ?p5
}
```

- Q3:

```
PREFIX ex: <http://example.org/>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type %ProductType% .
  SERVICE <http://localhost:5000/features/{label}>{
    ($.values[*]) AS (?f)
  }
  FILTER(?f = %ProductFeature%)
  ?product bsbm:productPropertyNumeric1 ?p1 .
  FILTER ( ?p1 > %value%)
  ?product bsbm:productPropertyNumeric2 ?p2
  FILTER (?p2 < %value2% )
}
```

- Q4:

```
PREFIX ex: <http://example.org/>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type %ProductType% .
  SERVICE <http://localhost:5000/features/{label}>{
    ($.values[*]) AS (?v1)
  }
  FILTER(?v1 = %ProductFeature%)
  SERVICE <http://localhost:5000/features/{label}>{
    ($.values[*]) AS (?v2)
  }
  FILTER((?v2 = %ProductFeature2% || ?v2 = %ProductFeature3%))
  ?product bsbm:productPropertyNumeric1 ?p1 .
  FILTER (?p1 > %value%)
}
```

- Q5:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX ex: <http://example.org/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type %ProductType% .
  %Product% rdfs:label ?label2
  FILTER (%Product% != ?product)
  %Product% bsbm:productFeature ?f .
  ?product bsbm:productFeature ?f
  SERVICE <http://localhost:5000/numeric/{label}>{
    ($.p1) AS (?simp1)
  }
  SERVICE <http://localhost:5000/numeric/{label2}>{
    ($.p1) AS (?origp1)
  }
  FILTER (?simp1 < (?origp1 + 120) && ?simp1 > (?origp1 - 120))
  SERVICE <http://localhost:5000/numeric/{label}>{
    ($.p2) AS (?simp2)
  }
  SERVICE <http://localhost:5000/numeric/{label2}>{
    ($.p2) AS (?origp2)
  }
  FILTER (?simp2 < (?origp2 + 500) && ?simp2 > (?origp2 - 500))
}
```

- Q7:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX ex: <http://example.org/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
  %Product% rdfs:label ?label .
  ?offer bsbm:product %Product% .
  ?offer ex:id ?id
  SERVICE <http://localhost:5000/offer/{id}>{
    (
      $.price, $.vendor,
      $.country
    ) AS (?pr, ?vendor, ?country)
  }
  FILTER(?country = "http://downlode.org/rdf/iso-3166/countries#GB")
  ?review bsbm:reviewFor %Product% .
  ?review ex:id ?id2 .
  SERVICE <http://localhost:5000/review/{id2}>{
    ($.revName, $.revTitle) AS
      (?revName, ?revTitle)
  }
  ?review bsbm:rating1 ?rating1 . ?review bsbm:rating2 ?rating2
}
```

- Q8:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX ex: <http://example.org/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
  %Product% rdfs:label ?label .
  ?review bsbm:reviewFor %Product% .
  ?review ex:id ?id2 .
  SERVICE <http://localhost:5000/review/{id2}>{
    (
      $.revName, $.revTitle,
      $.revText
    ) AS (?revName, ?revTitle, ?revText)
  }
  ?review bsbm:rating1 ?rating1 .
  ?review bsbm:rating2 ?rating2 .
  ?review bsbm:rating3 ?rating3 .
  ?review bsbm:rating4 ?rating4
}
```

- Q10:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX ex: <http://example.org/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
  %Product% rdfs:label ?label .
  ?offer bsbm:product %Product% .
  ?offer ex:id ?id .
  SERVICE <http://localhost:5000/offer/{id}>{
    (
      $.price, $.vendor,
      $.country
    ) AS (?price, ?vendor, ?country)
  }
  ?offer bsbm:deliveryDays ?devDays .
  FILTER(?devDays < %value%)
}
```

- Q12:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX ex: <http://example.org/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
  %Offer% bsbm:product ?p .
  ?p rdfs:label ?label .
  %Offer% ex:id ?id
  SERVICE <http://localhost:5000/offer/{id}>{
    ($.price, $.vendor)
      AS (?price, ?vendor)
  }
  %Offer% bsbm:deliveryDays ?devDays .
  %Offer% bsbm:offerWebpage ?offerURL .
  %Offer% bsbm:validTo ?validTo
}
```