Dissertation zur Erlangung des Doktorgrades der Technischen Fakultät der Albert-Ludwigs-Universität Freiburg

Distributed Processing of Navigational Query Languages for RDF

Martin Przyjaciel-Zablocki

Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Institut für Informatik

2017

Dekan

Prof. Dr. Oliver Paul

Referenten

Prof. Dr. Georg Lausen
Albert-Ludwigs-Universität Freiburg

Prof. Dr. Reinhard Pichler
Technische Universität Wien

Datum der Promotion

28.02.2017

Abstract

In recent years, we have witnessed the evolution from a *"Web of Documents"* to a highly-interlinked *"Web of Data"*, in which so-far human-consumable information is given a well defined machine-processable meaning. Driven by the wide adoption of Semantic Web technologies and in particular the so-called RDF data model, data from various domains and in multiple languages is becoming more and more interconnected. This facilitates the emergence of *semantic knowledge bases* such as DBpedia, YAGO, Microsoft's Satori, and Google's Knowledge Vault. However, querying such semantic knowledge bases, which have often a high degree of diversity in the structure and vocabulary, poses new challenges for query languages and their respective implementations. Despite considerable work that has been done in this area, recent work has proven that important properties in RDF data exist which cannot be captured by current RDF query languages including SPARQL 1.1 and its extensions.

In order to exploit the real potential of such data, RDF query languages need (1) to capture *all variants* inherent to the triple-based model of RDF and (2) to allow one to query RDF data along with its ontology and schema and (3) to enable querying path-based connections between arbitrary resources. Given the *graph-like* structure of highly-interconnected knowledge bases, we found expressive *navigational* queries to be well-suited to capture the aforementioned requirements, since they provide valuable information about the interlinking between arbitrary things. With this in mind, we propose two RDF query languages, namely RDFPath and TriAL-QL, which we believe constitute the first major contribution of this dissertation. Both languages aim at retrieving information by means of navigational queries and are meant to *complement* standardized languages like SPARQL. RDFPath is an intuitive navigational query language that enables one to traverse the graph structure of RDF along with its schema description. Its path-based semantics makes it possible to output complete paths, i.e. all resources traversed along a path. TriAL-QL is an SQL-like language built upon the *Triple Algebra with Recursion* (TriAL*). In contrast to many other approaches TriAL* is a compositional algebra, where the output is again RDF data. We further introduce E-TriAL*, an extension of TriAL* that adds more recursive expressions and supports provenance to describe the origin of a triple. In order to preserve the compatibility with other RDF management systems, both languages support the mapping of their results to RDF triples by using an ontology.

While the constant growth of semantically-annotated data and an increasing interest in cross-domain knowledge bases, justifies such expressive, navigational querying languages, it raises also the need for novel approaches that enable their evaluation for large data sizes. Therefore, we investigate next how to evaluate RDFPath and TriAL-QL queries against *web-scale* RDF data, which leads to the second major contribution of this dissertation. We investigate the application of Hadoop, the de-facto standard platform for processing *Big Data*, to distribute the workload associated with the evaluation of our languages on a cluster of machines. Apart from the widely deployed infrastructures, we see the main advantages of the Hadoop ecosystem in its continuous development which is reflected by novel frameworks and layers that are added continuously. Furthermore, we benefit, as we will also demonstrate in our work, from the concept of a common data storage by means of HDFS (also called *data lake*) that can be accessed by all applications built on top of Hadoop.

Our first implementation is the RDFPath MapReduce Processor, which is conceived for data-intensive and complex analytical RDFPath queries. It implements an efficient path serialization for HDFS, in addition to that two merge-join strategies for MapReduce are proposed to further optimize the overall query performance and scalability. MapReduce is highly scalable and reliable but due to its batch-oriented workflow and high latencies it is not well suited for rather selective queries evaluated against a small subset of the data. In such cases, one might expect more *interactive* query response times, thus in the order of seconds. We therefore continue our work with the TriAL-QL Engine and the RDFPath Engine which take advantage of the current momentum in *in-memory* SQL-on-Hadoop solutions. Both engines are implemented on top of *Impala*, a massive parallel SQL query engine and *Spark*, a fast general-execution framework for large-scale data processing while sharing *one* unified data store in HDFS. We investigate for all of our systems various *execution algorithms*, multiple *data storage strategies* and provide *optimizations* for the most important query patterns. A comprehensive evaluation examines the performance and scaling properties of our engines with respect to different execution strategies and in comparison to other competitive RDF management systems.

Zusammenfassung

In den letzten Jahren konnten wir die Entwicklung vom einem *“Internet der Dokumente”* hin zu einem hochgradig vernetzten *“Internet der Daten”* erleben. Dabei werden Informationen, die bisher nur von Menschen konsumiert werden konnten, mit ihrer semantischen und maschinenverarbeitbaren Bedeutungen verknüpft. Der zunehmende Einsatz von Semantic Web Technologien, hierzu zählt insbesondere das sogenannte RDF-Modell, ermöglichen dabei eine immer engere Verknüpfung von Daten aus verschiedenen Domänen und in unterschiedlichen Sprachen. Dies können wir insbesondere in semantischen *Wissensdatenbanken* wie DBpedia, YAGO, Microsoft’s Satori und Google’s Knowledge Vault beobachten, die zunehmend an Bedeutung gewinnen. Allerdings stellt die Abfrage semantischer Wissensdatenbanken aufgrund der verschiedenartigen Strukturen und des unterschiedlichen Vokabulars in den Daten erhebliche Herausforderungen an Abfragesprachen und deren Implementierungen. Trotz bedeutender Fortschritte in diesem Bereich haben daher kürzlich publizierte Arbeiten gezeigt, dass RDF Daten wichtige Informationen modellieren können, die mit aktuellen RDF Abfragesprache, wie SPARQL 1.1 und seinen Erweiterungen, nicht abgefragt werden können.

Um das Potential solcher Daten besser ausnutzen zu können, müssen RDF Abfragesprachen (1) *alle Varianten*, die dem Tripel-basierten Modell von RDF inhärent sind, abfragen können sowie (2) es erlauben, RDF Daten zusammen mit ihrer Ontologie und ihrem Schema gemeinsam in einer Anfrage auszuwerten und (3) es ermöglichen, pfadbasierte Verbindungen zwischen beliebigen Ressourcen anzufragen. Angesichts der *graphbasierten* Struktur solcher dicht vernetzten Wissensdatenbanken sehen wir ausdrucksfähige *Pfadanfragen* als gut geeignet an, um diesen Anforderungen gerecht zu werden, da sie wertvolle Informationen zur Verlinkung der Daten liefern können. Vor diesem Hintergrund stellen wir zwei RDF Abfragesprachen, RDFPath und TriAL-QL, vor, welche den ersten wesentlichen Beitrag dieser Dissertation darstellen. Beide Sprachen zielen darauf ab, Informationen unter Verwendung von Pfadanfrage zu extrahieren und sind dazu gedacht, existierende Sprachen wie SPARQL dahingehend zu *ergänzen.* RDFPath ist eine intuitive Pfadanfragesprache, die es ermöglicht, die graphbasierte Struktur von RDF zusammen mit Schema Informationen abzufragen. Die pfadbasierte Semantik von RDFPath unterstützt dabei die Ausgabe vollständiger Pfade, das heißt es werden alle Ressourcen entlang eines Pfades mit ausgegeben. TriAL-QL ist eine SQL-ähnliche Sprache, die auf der *Triple Algebra with Recursion* (TriAL*) beruht. Im Unterschied zu vielen anderen Ansätzen ist (TriAL*) eine kompositionelle Algebra, das heißt das Ausgabeformat entspricht der Eingabe und ist ein RDF Graph. Darüber hinaus führen wir

E-TriAL* ein, eine Erweiterung von TriAL*. Diese ermöglicht zusätzliche rekursive Ausdrücke und bietet die Möglichkeit, die Herkunft von Tripeln (Provenance) zu beschreiben. Um die Kompatibilität mit anderen RDF Management Systemen aufrechtzuerhalten, unterstützen beide Ansätze die Abbildung ihrer Ergebnisse in RDF Tripels, wobei eine eigene Ontologie verwendet wird.

Die konstante Zunahme an semantisch annotierten Daten aus unterschiedlichen Domänen und das ansteigende Interesse an deren Verknüpfung rechtfertigt solche ausdrucksstarken Abfragekonstrukte. Zugleich begründet sie auch die Notwendigkeit für neue Ansätze, welche es ermöglichen, solche Abfragesprachen auf großen Datensätzen auszuwerten. Daher lag im Folgenden unser Interesse darauf, wie RDFPath und TriAL-QL Anfragen auf *web-scale* RDF Daten ausgewertet werden können. Im zweiten wesentlichen Beitrag dieser Dissertation untersuchen wir deshalb die Verwendung von Hadoop, die de-facto Standard Plattform für *Big Data*, um die Berechnungen, die mit der Evaluation von unseren Sprachen einhergehen, auf einem Cluster an Maschinen zu verteilen. Über die reine Verbreitung solcher Infrastrukturen hinaus sehen wir die Hauptvorteile des Hadoop Ökosystems in dessen kontinuierlicher Weiterentwicklung, welche durch zahlreiche neue Frameworks, die in das Ökosystem eingepflegt werden, begleitet wird. Weiterhin profitieren wir von dem Konzept eines gemeinsamen Datenspeichers im HDFS (auch als *data lake* bezeichnet), auf welchen von allen Applikationen zugegriffen werden kann, die auf Grundlage von Hadoop implementiert sind. Auch dies werden wir mit unserer Arbeit demonstrieren.

Unsere erste Implementierung ist der RDFPath MapReduce Processor, welcher für daten-intensive und komplexe RDFPath Anfrage konzipiert ist. Er enthält eine effiziente Pfad-Serialisierung in HDFS und zusätzlich zwei Merge-Join Strategien für MapReduce, welche die Performance und Skalierbarkeit unseres Ansatzes weiter optimieren. MapReduce ist hochskalierbar und zuverlässig, allerdings aufgrund seiner Batch-basierten Arbeitsweise und hoher Latenz nicht geeignet für selektive Anfragen, die nur eine kleine Untermenge der Daten anfragen. In solchen Szenarien erwarten wir *interaktive* Antwortzeiten in der Größenordnung von Sekunden. Wir führen daher als nächstes die TriAL-QL Engine und die RDFPath Engine ein, welche die aktuelle Dynamik in der Entwicklung von *Hauptspeicher-basierten* SQL-on-Hadoop Lösungen ausnutzen. Beide Systeme verwenden sowohl *Impala*, ein massives paralleles SQL Anfrage System, als auch *Spark*, ein vielfältig einsetzbares cluster-gestütztes Framework, das für die Verarbeitung großer Datenmengen ausgelegt ist. Zur Datenspeicherung wird auf ein gemeinsames Datenschema im HDFS zugegriffen. Für die beiden letzten Systeme werden zudem unterschiedliche *Evaluationsstrategien*, verschiedene Konzepte zur *Partitionierung der Daten* sowie *Optimierungen* für die wichtigsten Anfragemuster vorgestellt. Umfangreiche Experimente untersuchen die Performance und Skalierbarkeit unserer Systeme im Hinblick auf die zahlreichen implementierten Strategien und insbesondere auch im Vergleich zu kompetitiven RDF Management Systemen.

Contents

Part I.

Preface

1. Introduction

Contents

In the past decade, we have witnessed the *World Wide Web* becoming the first source of knowledge in all areas of life. This web of interconnected HTML documents aims at presenting content to humans [KBM08]. The recent evolution from a *web of documents* to a *web of data and things* reflects the vision of a unified knowledge base, in which human-readable information is given a well defined *machine-processable* meaning [BHL01]. This concept goes back to an article of Tim Berners-Lee, one of the inventors of the World Wide Web. He envisaged a new web called *Semantic Web* [BHL01], where each *thing* in the world, whether a person, a movie, a location or any other concept can be identified by a unique name called Internationalized Resource Identifier (IRI). Knowledge representation is then what one obtains after describing the relationships between arbitrary things in the world, identified by their IRIs. For example, consider the concept *"Five Weeks in a Balloon"* appearing in some web page. Whereas for human readers interested in adventure novels, it might be obvious that this is the title of a book from Jules Verne, a machine programmed to extrapolate the semantics in texts might interpret it as advertisement for a vacation. An annotation with its IRI gives this term a machine-processable unique meaning, which can be then used to provide, for instance, more accurate meta-data.

Such knowledge can be created by means of the Resource Description Framework (RDF) [MMM14], the W3C standard for modeling Semantic Web data. The underlying building blocks of RDF are triples (s,p,o) that can be interpreted as statements with the meaning "a subject s has the predicate p with the object o". For example, we could model the beforehand mentioned book of Jules Verne with the following set of triples:

```
@base : <http://example.org/ontology/books/>;
<FiveWeeksBallon>    type     <Novel>.
<FiveWeeksBallon>    title    "Five Weeks in a Balloon".
<FiveWeeksBallon>    author   <JulesVerne> .
<JulesVerne>         type     <Person> .
<JulesVerne>         name     "Jules Verne" .
```

However, the real benefits of using RDF in comparison to other semi-structured data models, e.g. csv, rely not solely on the ability to annotate arbitrary things with their *machine-processable* meaning, but also in the usage of ontologies and schema descriptions from, e.g. *Schema.org*. They facilitate the integration of various cross-domain sources of information resulting in large, interconnected knowledge bases, such as DBpedia [BLK$^+$09], YAGO [HSBW13], Microsoft's Satori, and Google's Knowledge Vault [DGH$^+$14].

Related work done in the context of *life science* and *drug discovery* demonstrate the potential of such cross-domain knowledge bases [ABE$^+$09b, CDJ$^+$10, WDS$^+$12, CKK$^+$13]. Biological and chemical RDF data is used in that field to *"generate understanding about the way small molecules affect biological systems"* [WDS$^+$12]. A major challenge is hereby the search for *relationships* and *paths* that go across different datasets, where complete paths explaining the interconnection between arbitrary things become indispensable [WDS$^+$12]. Further examples include the integration of social network data [Mik04, Mik07] or open government data [SOBL$^+$12, SO13]. Navigational queries are therefore among the most natural questions for such types of data, as they are equipped with features to explore the graph-like structure of the data along with its ontology in order to provide valuable information about the interlinking of resources [EAL$^+$15, PBA$^+$16].

Moreover, recent work [LRV13, AGP14, RSV15] has shown that important properties exist in RDF data which cannot be captured by current RDF query languages such as SPARQL 1.1 and its derivatives. This includes also SPARQL Property Paths, the navigational component of the W3C standard query languages for RDF in its latest specification. One of the fundamental issues related to these languages is the fact that they are rooted in traditional graph databases, where the so-called *Regular Path Queries* (RPQs) [CMW87, CM90, CDLV03] are used as navigational primitives to traverse the graphical structure. Although the RDF model is very similar to underlying models of graph databases, it exhibits some crucial differences which hamper the applicability of languages based on RPQs. In a standard graph model, an edge label comes from a finite alphabet and cannot appear as a node, i.e. strictly distinct identifiers are used for nodes and edges. On the contrary, in RDF a subject or object resource (node in graph model) is allowed to be reused again at the place of a predicate (edge in graph model). In this way, two RDF triples such as (s_1, p_1, o_1) and (p_1, s_2, p_2) are valid triples in a RDF data management system but cannot be modeled correctly in a traditional graph databases. Consequently, languages based on RPQs also cannot navigate through such constructs. One may argue that this is not a dominant pattern in most published RDF data, which is truly the case. However, it is fundamental for ontologies and schema descriptions, which constitute the real benefit of the Semantic Web by making it possible to interlink various application fields in one knowledge base.

This is the motivation for the first part of this dissertation, in which we propose two expressive, navigational RDF query languages namely RDFPath [PSHL11, PSHL12] and TriAL-QL [PSL15a, PSL15b, PSL17]. Both languages cover several querying features, that have been identified as crucial for the Semantic Web. On the one hand, RDFPath features a fully path-based semantics that allow one to output the complete path which was traversed by a navigational expression. On the other hand TriAL-QL is a SQL-like language based on the *Triple Algebra with Recursion* (TriAL*) proposed in [LRV13]. In contrast to many other approaches TriAL* is a compositional algebra and hence closed, i.e. the output is again a set of triples rather than graphs or bindings. Further, with E-TriAL* we introduce an extension of TriAL*, which features more expressive recursions and a mechanism to deal with provenance, i.e. it allows one to track the origin of triples as a part of the query language.

The amount of available RDF data increases continuously due to the wide adoption of Semantic Web technologies in many diverse application fields ranging from the description of gene attributes in biology to the annotation of sports events. As a result, processing the workload associated with querying has become a challenging task in terms of computational costs [DGH$^+$14]. Expressive querying features are therefore just one side of the coin. The ability to evaluate them against web-scale RDF data is the other. We distinguish between three groups of RDF management systems, which differ in how large the RDF data is that can be processed: (1) centralized systems, (2) specialized distributed systems, and (3) distributed systems built on top of existing Big-Data frameworks.

Centralized systems operate on top of a single powerful machine, often equipped with specialized hardware components to enhance the performance. Common examples for such systems are *Sesame* [BKH02], Virtuoso [EM10], *Jena* [CDD$^+$04], *RDF-3X* [NW10] and *3store* [HG03]. In case of an increased amount of RDF data, resources such as processing power, main memory or hard disks can be improved. This strategy is known as *scale-up*. Although very powerful, such systems are limited in their scalability and are known to not be very cost efficient at larger scales.

In case of specialized distributed systems the workload parallelization is implemented as part of the RDF management system rather than relying on an underlying distributed framework such as Hadoop [Whi15]. Interesting approaches include *Virtuoso Cluster* [BEP14], *TriAD* [GSMT14], *Dream* [HRN$^+$15], *4store* [HLS09], *YARS2* [HUHD07] and *Clustered TDB* [Owe09]. Some of these are extensions of centralized systems, e.g. RDF-3X [HAR11, GHS14] or Jena [Owe09], which have to be independently installed on each machine in the cluster. Scaling in such systems is achieved by adding further machines, which is denoted as *scale-out* strategy. The main drawbacks of these systems are (1) the need for a dedicated infrastructure that has to be maintained solely for the purpose of querying RDF, and (2) the fact that the initial graph partitioning used for spreading data across machines is often done in a centralized way, being the bottleneck for large-scale RDF data [LL13].

The last group of RDF management systems is the one built on top of existing Big Data frameworks, such as MapReduce [DG04] or SQL-on-Hadoop [TSJ$^+$09, KBB$^+$15, AXL$^+$15] solutions. In recent years, the Hadoop ecosystem has become the de-facto standard for processing Big Data. Large infrastructures are deployed in research or industry and supported by major Cloud providers such as *Amazon Elastic Compute Cloud (EC2)*. Due to its robustness, reliability and scalability while being able to run on heterogeneous commodity hardware, Hadoop gained lot of attention in manifold application fields. Consequently, there have also been many different Semantic Web tasks implemented on top of Hadoop ranging from the evaluation of SPARQL queries [HAR11, HMM+11, SPZL11, SPNL14, SPSL16] to large-scale OWL reasoning [UKM+12].

However, to the best of our knowledge, not much work has been done on using Hadoop for the evaluation of expressive, navigational RDF query languages as RDFPath and TriAL-QL. At most, SPARQL 1.1 Property Paths are supported via *Virtuoso Cluster* as being the only available *distributed* RDF management system that runs on a cluster of machines [BEP14]. Implementations of RDF query languages having a higher expressiveness than Property Paths can only be found in centralized systems, such as in extensions of Sesame and Jena, which lack the support for querying web-scale RDF data. Moreover, distributed RDF management systems on Hadoop are so far able to evaluate SPARQL 1.0 queries. This means that there is no support for expressive navigational queries that allow one to query RDF data together with its ontology while providing meaningful results [HAR11, HMM+11, SPZL11, SPNL14, SPSL16].

This gap is the motivation for the second part of this dissertation in which we believe the major contributions of our work have been made. With the goal of processing expressive navigational queries of both languages, RDFPath and Trial-QL, on web-scale RDF data, we investigated the usage of Hadoop-based solutions for distributing the workload on a cluster of machines. The main reasons why we have chosen Hadoop were, first of all its rich ecosystem of frameworks that are continuously evolved and adapted to novel concepts for distributed computing. Secondly, there exists already widely deployed infrastructures as explained above. Thirdly, we believe in the concept of using HDFS as a shared *data pool*, that can be accessed by various Semantic Web applications without the need for data duplications or movement. That way, we encourage the usage of our languages and implementations as supplementary tools, for e.g. preprocessing the data where an Hadoop-based SPARQL engine like Sempala [SPNL14] or S2RDF [SPSL16] can use our results again as its input.

We started with a purely MapReduce-based evaluation of RDFPath queries which lead to the development of our RDFPath MapReduce Processor. Our prototype demonstrates that it is very well suited for data-intensive or complex analytical tasks [PSHL11, PSHL12] with good scalability properties. To further improve its performance, we designed an optimized join strategy which turned out to also be applicable in other scenarios. However, in case of more selective queries which are evaluated against a small subset of the data, we expected to get an answer in *interactive* time, i.e in the order of seconds to a few minutes. We realized this was difficult to achieve with MapReduce because this paradigm is not intended to fulfill those requirements. Therefore, we investigate in a further step if it is possible to achieve this using the current trend in SQL-on-Hadoop solutions and designed TriAL-QL Engine and RDFPath Engine, two distributed query processors with support for both, data-intensive, analytical tasks but also interactive querying. In the core, both, *Impala* [KBB$^+$15], a massive parallel SQL query engine on Hadoop and *Spark* [AXL$^+$15], a fast general execution framework for large-scale data processing are used while sharing *one* unified data store in HDFS [PSL15a, PSL15b, PSL17].

These challenges give rise to many interesting problems with multiple interpretations that are worth being investigated. In this dissertation we will mainly focus on answering the following questions, which also reflect the structure of this work:

1. How has the Semantic Web evolved in recent years? What are the *representative* datasets and benchmarks on which we may base our experiments and which reflect the recent challenges and opportunities?
2. Which languages can be used to query RDF and what are their navigational primitives? Where do we see the need for more expressive querying features?
3. How have we designed RDFPath, an intuitive yet expressive navigational RDF query language equipped with functionalities to traverse both the RDF data together with its ontology?
4. How can we adopt *Triple Algebra with Recursion* (TriAL*) to fulfill our requirements as the core of a navigational query language? What is a user-friendly syntax for the algebraic notation of TriAL*? How can we enrich the results by incorporating provenance?
5. How can we obtain compatibility with other RDF management systems and languages, which in turn allow us to introduce both of our languages as complementary approaches to existing solutions?
6. How to evaluate our languages against *web-scale* RDF data and support both (1) ETL-like, *data-intensive* tasks and (2) *interactive querying* in cases of queries with a high selectivity? What are suitable algorithms for the evaluation of recursive patterns and how can they be further optimized?
7. What are the performance and scaling properties of our implementations with respect to different *evaluation strategie*s and in comparison to other competitive RDF management systems?

1.1. Contributions

The main contributions of this dissertation can be summarized as follows. We design two navigational RDF query languages, namely RDFPath and TriAL-QL which aim at retrieving meaningful information from highly diverse, interconnected knowledge graphs.

- RDFPath stands out with a fully path-based semantics and allows one to evaluate queries while fully exposing the paths along with the answer, i.e. it includes all resources along a path, rather than just confirming the existence of a connection between two resources.
- TriAL-QL is a SQL-like language based upon *Triple Algebra with Recursion* (TriAL*) [LRV13], an expressive compositional algebra for RDF that subsumes most previous languages.
- We further introduce E-TriAL*, an extension of TriAL* that includes more recursive expressions and supports the tracking of provenance for newly-derived triples.
- Both languages allow to output RDF triples which encode their computed paths by means of an ontology. This in turn facilitates their usage as complementary languages in the stack of established Semantic Web languages and tools, where they could be used for, e.g. preprocessing certain interconnections.

To evaluate queries of both languages on web-scale RDF data, we develop *three* query engines on top of Apache Hadoop with HDFS as a common data-pool.

- The RDFPath MapReduce Processor translates RDFPath queries to a sequence of MapReduce jobs using an efficient path serialization provided by our implemented data store on top of HDFS. With regard to the optimizations two kinds of merge joins are presented, which improve the overall query performance and scalability.
- Our TriAL-QL Engine is implemented on top of both *Impala* and *Spark* while sharing *one* unified data store in HDFS.
- Our RDFPath Engine is also implemented on top of both *in-memory* frameworks and shares *one* unified data store in HDFS.
- For both engines we investigate multiple algorithms for the evaluation of recursive expressions and propose execution strategies, which specify how translated queries are composed and when intermediate results need to be materialized. Further optimizations include algorithms for special patterns, e.g. the connectivity between two given resources.
- We present different data storage strategies, which make use of Parquet, an efficient columnar storage format, for data partitioning.
- A comprehensive evaluation assesses the performance and scaling properties of our engines and their various execution strategies in comparison to other competitive RDF management systems.

Together, the query languages and the implemented query engines constitute the major contribution of this work. The following paragraph contains a more detailed view on our contributions. It includes all relevant publications, provides the most complete picture of our work, and answers the main research questions raised at the end of the introduction.

Contributions & Published Work. The growing adoption of Semantic Web technologies in various application fields promotes that new sources of information are interconnected to a large knowledge base. Therefore, Semantic Web data exhibits a high degree of diversity in its structure, demanding a redefinition of the requirements of modern RDF management systems and suitable query languages [AHOD14]. One crucial point in developing such systems and languages is therefore the usage of proper benchmarks with representative datasets to base our tests and experiments on, that capture this recent evolution. In [PSHT13] we discuss the issues of current RDF benchmarks and specify properties for a *representative* dataset that reflects the new challenges and opportunities of the Semantic Web. We bring together these properties into LastBench, a proposal for a benchmark on real-world social data from Last.fm.

[PSHT13] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Io Taxidou. Towards a SPARQL 1.1 Feature Benchmark on Real-World Social Network Data. In *Proc. of the ESWC2013 Workshops: First International Workshop on Benchmarking RDF Systems (BeRSys), Montpellier, France*, CEUR Workshop Proceedings 981, 2013.

Navigational queries are among the most important query types for interconnected and semantically-annotated data, as they provide valuable information about the interlinking of resources and are equipped with features to explore the underlying graph-like structure. However, most RDF languages including SPARQL 1.1 Property Paths have some inherent limitations, due to their rooting in traditional graph databases. They fail to cover all variants inherent to the triple-based model of RDF, which is particularly problematic when RDF data needs to be queried along with its ontology. Furthermore, the problem of reachability, if at all addressed, is often seen as an existential question, i.e. it answers whether there exists a connection between certain resources. Only limited work has been done on providing more meaningful results, e.g. that include complete paths with all traversed resources or on describing the origin of the result by means of provenance. We have therefore introduced in [PSHL11, PSHL12] RDFPath, an intuitive yet expressive, navigational query language for RDF which addresses some of these needs. It is equipped with functionalities to traverse paths of arbitrary length, allowing navigation through both RDF data and its ontology. The distinctive feature of RDFPath is its path-based semantics defined over an extension of RDF which we called RDFp. This allows us to provide complete paths to appear in the results of a query rather than just pairs of nodes or variable bindings.

Another aspect that emerges from the wide usage of Semantic Web technologies is related to the amount of RDF data created in various domains. In recent years we have observed, that the available RDF data has grown to the point where it is crucial to distribute the workload associated with querying the data to a cluster of machines [HAR11]. Along with RDFPath, we describe therefore in [PSHL11, PSHL12] our RDFPath MapReduce Processor which aims at the evaluation of RDFPath queries on web-scale RDF data. Queries are hereby automatically translated to a sequence MapReduce jobs and executed on a cluster of machines. It uses our RDFp Store, a storage schema built on top of Hadoop for paths which features its own serializable format for efficient comparison and retrieval of paths. To the best of our knowledge, RDFPath MapReduce Processor was the first navigational RDF query language implemented using MapReduce. Furthermore, in order to improve the overall performance and scalability our RDFPath MapReduce Processor, which is based on *Reduce-Side* joins, we have published in [PSS$^+$13] two optimized join techniques, called *Map-Side Merge* joins. Both join techniques are computed completely in the map phase and support a cascaded execution while utilizing the reduce phase just for sorting. Bloom filters are used to remove dangling intermediate results. A comprehensive set of experiments confirms its applicability for the evaluation of RDFPath queries and also as a general-purpose join in MapReduce.

[PSHL11] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Georg Lausen. RDFPath: Path Query Processing on Large RDF Graphs with MapReduce. In *Proc. of the Workshop on High-Performance Computing for the Semantic Web (HPCSW), Heraklion, Greece*, volume 736 of *CEUR Workshop Proceedings*, 2011.

[PSHL12] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Georg Lausen. RDFPath: Path Query Processing on Large RDF Graphs with MapReduce. In *The Semantic Web: ESWC 2011 Workshops, Revised Selected Papers*, vol. 7117 of *LNCS*, pages 50–64. Springer Berlin Heidelberg, 2012.

[PSS$^+$13] Martin Przyjaciel-Zablocki, Alexander Schätzle, Eduard Skaley, Thomas Hornung, and Georg Lausen. Map-Side Merge Joins for Scalable SPARQL BGP Processing. In *Proc. of the IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom), Bristol, UK*, volume 1, pages 631–638, 2013.

We continue the work on expressive, navigational query languages and their distributed query engines in [PSL15a, PSL15b, PSL17]. First of all, we introduce TriAL-QL an SQL-like language for querying RDF. However, rather then specifying a new language from scratch, we decided to base it on an existent algebra called *Triple Algebra with Recursion* (TriAL*) which was originally proposed in [LRV13]. TriAL* is one of the most expressive algebras for RDF. This language subsumes almost all previously introduced RDF query languages, while adding novel features that are not expressible in most other languages based on the standard graph model [LRV13]. TriAL* is a closed language, where its basic idea is to work directly with triples rather than transforming the data into another representation. However, in practical scenarios we identified several shortcomings which hamper its usage. First of all, its algebraic notation is, similar to the relational algebra, not easily writable. Secondly, it is missing important recursive expressions that would allow one to bound the number of recursions. Thirdly, due to the implicit projection to triples, there is an inherent lose of information that diminishes the interpretability of the results. In order to overcome all these issues, we initially propose the Extended Triple Algebra with Recursion, or in short E-TriAL* that is equipped with more recursion capabilities and supports provenance to track the origin of triples. As a next step, we introduce a writable syntax called TriAL* Query Language (TriAL-QL), which is an easy to write and understand representation of TriAL*.

Whereas our previous work on RDFPath focuses on rather data-intensive, analytical tasks on web-scale RDF data, with the implementation of TriAL-QL we target a wider range of queries. For ETL-like workloads, which are typically processed *offline*, runtimes in the order of minutes to hour are acceptable. However, in cases of more selective queries requiring only a small subset of the data, getting results in *interactive time*, i.e. in the order of seconds to few minutes is desirable. As our experiments demonstrate, MapReduce is not well-suited for such scenarios, since its batch-oriented workflow materializes all intermediate results to disk which implies costly I/O operations. With the decreasing prices of main memory in recent years, one can observe the emergence of novel *in-memory* data processing systems for Hadoop such as Stinger for *Hive*, *Impala*, *Spark*, *Presto* and *Phoenix* which start to pave the way for scalable and interactive querying on large-scale data. Following this trend, we introduce in [PSL15b, PSL17] our TriAL-QL Engine, implemented on top of both, *Impala*, a massive parallel SQL query engine on Hadoop and *Spark*, a fast general execution framework for large-scale data processing, while sharing *one* unified data store in HDFS. For storing data, we propose two data storing strategies that make use of Parquet, an efficient columnar storage format. We use Vertical Partitioning and an adapted version of the Extended Vertical Partitioning strategy for storing input graphs. We propose multiple evaluation strategies for recursive expressions in TriAL-QL Engine. To that end, we make use of the well-studied problem of calculating the transitive closure (TC). We use two approaches as a starting point, namely the *semi-naive* and *smart* TC algorithms, and adapt them to perform well in our processing frameworks. Furthermore, we also investigate different execution strategies, which specify how translated SQL queries are composed for their execution and when intermediate results need to be materialized to disk. These and other aspects will be assessed by means of experiments on generated social networks with up to 1.8 billion triples. A follow-up work, which was not yet published at the point of writing this dissertation, investigates also the application of those strategies for RDFPath. Nevertheless, we have included the description of our RDFPath Engine and its experimental results into this dissertation.

[PSL15a] Martin Przyjaciel-Zablocki, Alexander Schätzle, and Adrian Lange. TriAL-QL: Distributed Processing of Navigational Queries. In *Proc. of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW), Lima, Peru*, volume 1378 of *CEUR Workshop Proceedings*, 2015.

[PSL15b] Martin Przyjaciel-Zablocki, Alexander Schätzle, and Georg Lausen. TriAL-QL: Distributed Processing of Navigational Queries. In *Proc. of the 18th International Workshop on Web and Databases (WebDB 15), Melbourne, Australia*, pages 48–54, 2015.

[PSL17] Martin Przyjaciel-Zablocki, Alexander Schätzle, and Georg Lausen. Querying Semantic Knowledge Bases with SQL-on-Hadoop. In *Proc. of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2017, Chicago, IL, USA*, pages 4:1–4:10, 2017.

In addition to the work discussed in this thesis, the author has also contributed to a series of other publications related to RDF querying on Hadoop. This work is part of the Ph.D. thesis of Alexander Schätzle [Sch16] and therefore is not discussed in this thesis. Nevertheless, we provide a brief summary to complete the picture and give an intuition of how it is related to the work discussed in this thesis. Most notably, in [PSH+12, SPD+12, SPHL14] we present the *Map-Side Index Nested Loop* (MAPSIN) join for MapReduce. It uses the indexing capabilities of HBase, a distributed NoSQL data store, to improve query performance of selective queries. Similar to our *Map-Side Merge* join [PSS+13], it is processed completely in the map phase to reduce costly data shuffling. Besides the development of query engines for our expressive, navigational query languages, we contributed also to a distributed engine for SPARQL, the W3C standard languages for querying RDF. PigSPARQL [SPHL11, SPL11, SPHL13] is our first work in that field. It is a processor for SPARQL 1.0 which translates an input query into an equivalent Pig Latin program, which is then compiled into a series of MapReduce jobs by Pig.

Again however the batch-oriented workflow of MapReduce prevents interactive querying times. Consequently, we investigate the applicability of novel *in-memory* data processing systems also for the evaluation of SPARQL queries. In [SPNL14], we present Sempala, a distributed SPARQL processor which translates queries into the SQL dialect of Impala. A so-called *Unified Property Table* is proposed, which exploits the columnar storage layout of Parquet, and is highly optimized for star-shaped queries. A comprehensive evaluation in [SPNL14] demonstrates the performance improvements by an order of magnitude on average compared to other SPARQL-on-MapReduce approaches. Due to the graph-like nature of RDF, we next investigate the suitability of a graph-parallel framework for processing SPARQL queries. S2X (SPARQL on Spark with GraphX) is a SPARQL engine built upon the GraphX API, a graph-parallel abstraction layer for Spark. In [SPBL15], we define a mapping from RDF to the *property graph* model of GraphX and propose an adopted algorithm from [FNR+13] to express the task of graph pattern matching in SPARQL 1.0 within the vertex-centric programming model of GraphX. Our experiments show acceptable results in comparison to MapReduce-based approaches, although the scaling with larger data sizes are much worse. Furthermore, we conclude that graph-parallel frameworks are not suitable for querying RDF as they are rather intended to perform iterative graph algorithms like PageRank. These results pushed us towards the decision of not to investigate the implementation of RDFPath and TriAL-QL on top of a graph-parallel framework but rather to data-parallel frameworks such as Impala and Spark which have good performance for our use cases

This is confirmed with our work in [SPSL15, SPSL16], where we describe S2RDF (SPARQL on Spark for RDF), a SPARQL query engine on top of SPARK. We propose a novel relational partitioning schema for RDF called *ExtVP* (Extended Vertical Partitioning) that is an extension of the well-known *Vertical Partitioning* (VP) schema introduced in [AMMH07]. In order to reduce the data overhead in VP, we choose as an optimization strategy the so-called selectivity threshold, which ensures that only beneficial partitions are stored. An extensive evaluation with other state-of-the-art SPARQL processors for Hadoop demonstrates the superior performance of S2RDF on very diverse query workloads. Due to these promising, we investigate the application of *ExtVP* also for our RDFPath Engine.

[SPHL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. PigSPARQL: Übersetzung von SPARQL nach Pig Latin. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, vol. P-180 of *LNI*, pages 65–84, Kaiserslautern, Germany, 2011.

[SPL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proc. of the 3rd International Workshop on Semantic Web Information Management (SWIM), Athens, Greece*, SWIM'11, pages 4:1–4:8, 2011.

[SPHL13] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. PigSPARQL: A SPARQL Query Processing Baseline for Big Data. In *Proc. of the ISWC 2013 Posters & Demos Track*, volume 1035 of *CEUR Workshop Proceedings*, pages 241–244, 2013.

[PSH+12] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, Christopher Dorner, and Georg Lausen. Cascading Map-Side Joins over HBase for Scalable Join Processing - Technical Report. *Computing Research Repository (CoRR)*, arXiv:1206.6293, 2012.

[SPD+12] Alexander Schätzle, Martin Przyjaciel-Zablocki, Christopher Dorner, Thomas Hornung, and Georg Lausen. Cascading Map-Side Joins over HBase for Scalable Join Processing. In *Proc. of the Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS + HPCSW), Boston, USA*, vol. 943 of *CEUR Workshop Proceedings*, pages 59–74, 2012.

[SPHL14] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. Large-Scale RDF Processing with MapReduce. In *Large Scale and Big Data - Processing and Management*, pages 151–182. Auerbach Publications, 2014.

[SPNL14] Alexander Schätzle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg Lausen. Sempala: Interactive SPARQL Query Processing on Hadoop. In *The Semantic Web: ISWC 2014 - 13th International Semantic Web Conference*, vol. 8796 of *LNCS*, pages 164–179, Riva del Garda, Italy, 2014.

[SPBL15] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thorsten Berberich, and Georg Lausen. S2X: Graph-Parallel Querying of RDF with GraphX. In *Biomedical Data Management and Graph Online Querying, VLDB 2015 Workshops Big-O(Q) and DMAH, Revised Selected Papers*, vol. 9579 of *LNCS*, pages 155–168. Springer International Publishing, 2015.

[SPSL15] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF Querying with SPARQL on Spark - Tech. Report. *Computing Research Repository (CoRR)*, arXiv:1512.07021, 2015.

[SPSL16] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF Querying with SPARQL on Spark. In *Proc. of the VLDB Endowment (PVLDB)*, 9(10): 804–815, 2016.

The big picture of implemented RDF management systems is shown in Figure 1.1, where we illustrate the intended usage and respective technology upon which each system is based.

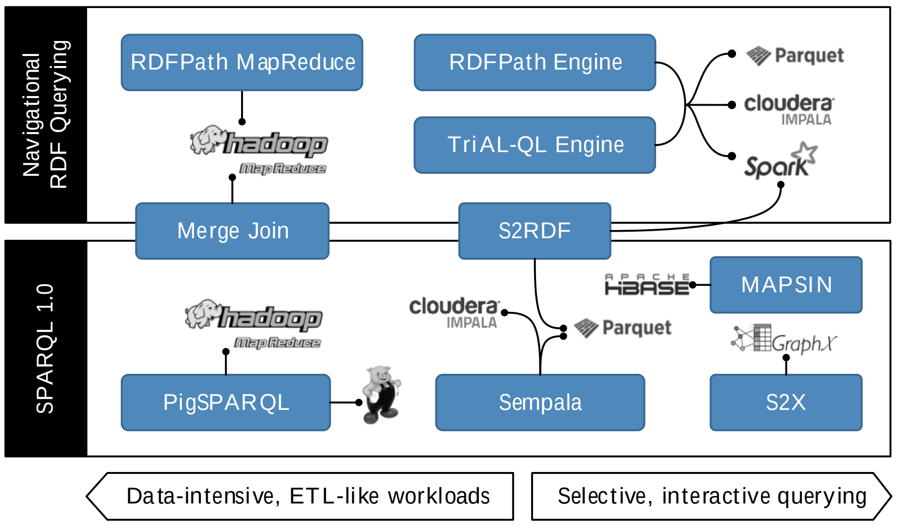


Figure 1.1.: Overview of RDF management systems and contributions. RDFPath MapReduce Processor, Merge Join, RDFPath Engine, TriAL-QL Engine are part of this dissertation whereas PigSPARQL, Mapsin, Sempala, S2x, and S2rdf are presented in the dissertation of Alexander Schätzle [Sch16].

Furthermore, in addition to topics related to RDF querying on Hadoop, the authors have also published and contributed to a series of publications related to *Music Recommender Systems*, which are not relevant for this dissertation and are therefore not discussed in this thesis. The work done in this area has only a few intersections with this dissertation, which we want to shortly explain. First, in [PHS+14] we develop a Music Recommender Management System called MuSe, whose back-end is built on top of the Hadoop ecosystem, and allows one to distribute the workload of computing recommendations on a cluster of machines. Similar to our work on querying RDF data, we use Spark to generate user-specific recommendations in real-time and store a copy of the data which had to reside in main-memory as Parquet files in HDFS. Furthermore, within the works in [FHZ+13, ZHP+14, HZF+13] we target mostly the so-called cold-start problem of recommender systems. This problem describes a scenario where a new user or a new item is entered into the system and there is not enough knowledge about them to produce accurate recommendations. Aiming at mitigating this problem, we attempt to enrich data about songs and their artists by means of semantic knowledge bases. Corresponding interfaces are incorporated in MuSe, but have not been used yet for music recommendations.

[PHS+14] Martin Przyjaciel-Zablocki, Thomas Hornung, Alexander Schätzle, Sven Gauß, Io Taxidou, and Georg Lausen. MUSE: A Music Recommendation Management System. In *Proc. of the 15th International Society for Music Information Retrieval Conference (ISMIR 2014)*, Taipei, Taiwan, 2014.

[ZHP+14] Cai-Nicolas Ziegler, Thomas Hornung, Martin Przyjaciel-Zablocki, Sven Gauß, Georg Lausen. Music Recommenders Based on Hybrid Techniques and Serendipity. In *Journal on Web Intelligence and Agent Systems (WIAS)*, 12(3): 235–248, 2014

[FHZ+13] Simon Franz, Thomas Hornung, Cai-Nicolas Ziegler, Martin Przyjaciel-Zablocki, Alexander Schätzle, Georg Lausen. On Weighted Hybrid Track Recommendations In *Proc. of the 13th International Conference on Web Engineering (ICWE 2013)*, Aalborg, Denmark, 2013.

[HZF+13] Thomas Hornung, Cai-Nicolas Ziegler, Simon Franz, Martin Przyjaciel-Zablocki, Alexander Schätzle, Georg Lausen. Evaluating Hybrid Music Recommender Systems In *Proc. of the 12th IEEE/WIC/ACM International Conference on Web Intelligence (WI 2013)*, Atlanta, USA, 2013.

Open-Source Code. All the implementations discussed in this dissertation have been open-sourced and are available for download. In addition to the project website hosted by the University of Freiburg, we have published our code on GitHub to facilitate the contribution of other researchers and the continuation of work in this field.

`http://github.com/martinpz`

`http://dbis.informatik.uni-freiburg.de/forschung/projekte/DiPoS`

1.2. Thesis Outline

The remainder of this dissertation consists of three parts. Part II. provides some foundation on benchmarking RDF data management systems.

- **Chapter 2** discusses desirable features of benchmarks and representative datasets required to develop modern RDF management systems. Furthermore, it presents LastBench, a benchmark proposal on real-world social data gathered from Last.fm.

Part III. of this dissertation focuses on the theoretical foundations and properties of RDFPath and TriAL-QL. This part is divided into three chapters:

- **Chapter 3** introduces navigational primitives used in graph query languages and provides a comprehensive comparison of RDF query languages.
- **Chapter 4** starts with a definition of RDFp, a flexible yet simple extension of RDF to represent paths in RDF graphs. After that, the syntax and semantics of RDFPath are introduced, which is a navigational query language for RDF that provides complete paths in the result.
- **Chapter 5** introduces our second language called TriAL-QL which is based on *Triple Algebra with Recursion* (TriAL*). We present our extension E-TriAL* which supports provenance to track the origin of triples and is equipped with more recursion capabilities.

Part IV. focuses on algorithmic and technical aspects of our implemented query engines for RDFPath and TriAL-QL. This part is divided into three chapters:

- **Chapter 6** presents our RDFPath MapReduce Processor, with the goal to evaluate RDFPath queries on web-scale RDF data. Section 6.5 introduces *Map Side Merge* joins as an optimization strategy.
- The first half of **Chapter 7** presents our TriAL-QL engine (7.2) implemented on top of two in-memory data processing frameworks, Impala and Spark. We discuss the data storage layout, multiple evaluation strategies and propose optimizations for most important patterns. Their impact on the performance is investigated with experiments.
- The second half of **Chapter 7** introduces our RDFPath engine (7.4). We describe an improved data storage layout, adapt evaluation algorithms from TriAL-QL, and present various execution strategies. Again, their impact on the performance is investigated by some individual experiments.
- **Chapter 8** presents a comprehensive evaluation in which our engines are compared with other competitive RDF management systems in terms of performance and scalability.

Finally, the last Part V. summarizes in **Chapter 9** this dissertation and outlines directions for future research.

Part II.

Benchmarks for the Semantic Web

2. Benchmarking RDF Data Management Systems

Contents

In recent years, we observe a steady growth of semantically-annotated data published in new domains. However, it is not only the size of data which poses new challenges, but rather the data's structure. Many new application fields arise where the data forms a highly-diverse knowledge graph with a rich taxonomy[1]. This leads to a high degree of diversity in the structure of real RDF data which in turn demands a rethinking of requirements for modern RDF management systems and their respective query languages [AHOD14]. One crucial point in developing such systems and their query languages are therefore proper benchmarks with representative datasets upon which to base our tests and experiments which capture this recent trend. While several proposals exist for how to model RDF benchmarks in a rather structured way, they are mostly derived from how we benchmark relational databases geared towards testing basic querying features like offered by SPARQL 1.0 [GPH05, SHLP08, SHM+09, SHLP09, BS09]. There has been limited work done on benchmarks with more realistic RDF data that capture the Semantic Web as it evolved in recent years [MLAN11, DKSU11].

Furthermore, we can see that it is not merely RDF that pushed the applicability of Semantic Web. The emergence of comprehensive ontologies and taxonomies models knowledge more and more accurately and enable to interconnect different sources of information, which provide the real benefits of semantically-annotated data. Utilizing such enriched structures is becoming indispensable in modern RDF data-management systems. However, this requires more sophisticated querying features than offered by current RDF query languages like SPARQL 1.1 and are therefore mostly neglected in RDF benchmarks [PSHT13, EAL+15].

[1]`https://datahub.io/organization`

With this in mind, this chapter will start with an overview of the most commonly-used RDF benchmarks and investigate their applicability for the languages and implementations discussed in this dissertation. In that sense, desirable features of a benchmark are introduced, which we consider to be relevant for this work. As we will see, none of the existent benchmark candidates would have met our criteria, and thus we decided to combine them into LastBench, a benchmark proposal on real-world social data from Last.fm. This will form the main contribution of this chapter, in which these three aspects hitherto discussed will be presented.

We start with the challenges that come along with real-world data and propose new strategies for obtaining realistic datasets by the use of sampling strategies known from analyzing online social networks. The algorithms' implementation is discussed in view of an efficient sampling of large-scale datasets from the web. For that, a distributed architecture was developed, which enables us to parallelize the sampling process. To that end, an algorithm for scaling real-world datasets is presented, which utilizes properties of our sampling strategies in order to produce more accurate subsets of the data. A short discussion on the latest state of LastBench concludes this chapter, where we will argue why the work on this benchmark has not been continued.

The main parts of this chapter were published in [PSHT13], where we proposed our initial draft of LastBench. The algorithms and specifications seen on the following pages are in a much more concrete form than shown in [PSHT13] but are conceptually the same. Here, however, the distributed implementation which is available for download[2], is presented for the first time.

We can summarize the contributions of this chapter as follows:

- First, we summarize in Section 2.1 seven features of a benchmark, which we consider to be relevant for the languages and implementations shown in this dissertation. We will then provide an overview of related benchmarks for RDF, where we explain why none of these solutions fit our needs.
- LastBench, our proposal for a real-world social benchmark for RDF management systems is presented in Section 2.2, where we start with a discussion on Last.fm and the data in which we are interested. Section 2.2.1 introduces our algorithms for sampling online social networks.
- A distributed architecture which implements our suggested strategies for sampling online social networks is described in Section 2.3.
- Section 2.3.1 explains how we leverage some properties inherent to our sampling strategies in order to down-scale LastBench data such that smaller datasets preserve their social network properties.
- A preliminary analysis of a sampled social graph from Last.fm investigates its social network properties and complements the proposal of LastBench in its latest state.

[2]`http://github.com/martinpz`

2.1. Benchmarks and their Characteristics

Social networks like Facebook, Twitter, Last.fm etc. dramatically change the way how people interact, collaborate and share information, turning the web into an highly interactive and personalized interlinked *"Web of Data"*. According to this perspective, one can also interpret a social network graph as structured semantic data interlinking people and objects that can also be represented in RDF. This is also underpinned by the already-underway effort of adding RDF to Facebook's Graph API [WT13]. On the other hand, there is a lack of real-world RDF benchmark data in general, and social network data in particular [DKSU11, VMS12]. Most of the existing RDF benchmarks like BSBM [BS09], LUBM [GPH05] or SP^{2}Bench [SHLP08, SHM$^+$09, SHLP09] use artificial data generated according to observed frequency distributions of a specific domain. While this approach enables the easy scaling of the size, generated datasets have little in common with real RDF data. They often resemble relational database benchmarks [MLAN11] with a high level of *structuredness* [DKSU11] and lack *structural properites*, e.g. correlations between users, which are a crucial aspect of social networks but are hard to discover [PBE12].

We recall that the main goal of this dissertation is to, (1) specify expressive, navigational query languages for RDF that are able to derive likely up-to-now *hidden pieces* of information within the data and (2) investigate strategies how to distribute the query processing in order to handle web-scale RDF data. With that and the evolution of semantically annotated data in mind, we will continue with an overview of desirable features for a benchmark which would help us in developing and testing our approaches.

- **Social Graphs:** Social networks have fundamentally changed our perception of the web and the way we interact with it. At the same time we have witnessed the vision of the Semantic Web picking up the pace. From a general perspective, the inherently complex, intertwined structure of a social network contains a flood of *semantic information* about users, objects and their relations. On the other hand, social graph structures are hardly covered by current state-of-the-art RDF benchmarks. Synthetic RDF generators in this area do not model all properties of a social network, especially *structural correlations*, e.g. friendship relationships which are correlated with the place of residence, and they are either neglected or underrepresented. The S3G2 data generator [PBE12] for *structure-correlated social graphs* that is used for the Social Network Intelligence Benchmark (SIB) [PBE12] focuses on exactly this aspect. It generates arbitrarily large social graphs with a predefined set of structural correlations. However, the authors emphasize that their generator cannot produce "realistic" social network data as these networks are expected to have many more (yet unknown) correlations.

- **Realness of Data:** The obvious solution to represent all relevant correlation might be simply the usage of real data rather than generated. The DBpedia SPARQL benchmark (DBPSB) [MLAN11] which is based on data dumps from Wikipedia is one of the few benchmarks that uses real data. However, compared to the size and dynamics of social graphs, it is considered to be rather small and also limited in growth. In addition, although it is truly collaboratively-created content, it lacks social interaction between users and their correlations as mentioned earlier. Another problem related to DBPSB resides in its inherent scalability issues, which we will outline next in more detail.

- **Scalability:** Scaling a real-world dataset, i.e. selecting an appropriate method for creating smaller (scale-down) and larger (scale-up) datasets out of the original one is a difficult problem. The authors in [MLAN11] propose two strategies to scale-down DBpedia. The first one is based on selecting the desired fraction of the dataset *randomly*. The second strategy is based on sampling that takes the actual *class distribution* into account, where the resulting fraction is closer to the full DBpedia graph with respect to the in- and out-degree and distinct nodes. For a scale-up of DBpedia, that is, creating datasets of multiple size of the original graph, the full dataset is duplicated and the namespaces are changed. However, both adding and removing triples to the end results in information loss as either some likely-important edges are removed or some non-representative ones are added. This is particularly problematic for benchmarking distributed systems, as it hampers comprehensive conclusions about the scalability of such systems with respect to the data size. Most benchmarks are therefore based on synthetic-data generators, that aim to mimic distributions and correlations observed in real data [GPH05, SHLP08, SHM$^+$09, SHLP09, BS09]. Such generators have the strength to produce datasets of arbitrary size while keeping their distributions and correlations. This simplifies a fair comparison of various systems and enables more comprehensive conclusions about the scalability of systems with respect to the data size.
- **Structuredness:** The authors in [DKSU11] studied the characteristics of datasets used in RDF benchmarks, where they revealed that synthetic datasets have very little in common with real RDF data [DKSU11]. They reason that this is based on the so-called level of *structuredness*, whereby highly structured, relational-like data like LUBM is considered to have a high level of structuredness. Here it is the case, that for each type of an instance almost the same properties are given. A low level of structuredness is common in rather unstructured data sources like Wikipedia/DBpedia which can be better represented as RDF data rather than in a relational database. Here we can observe that, even for instances of the same type, the actual properties which are present are often non-overlapping. Due to such crucial differences in the underlying structure, the level of structuredness plays a central role for data representation and optimizations, and affects also how we need to query the data, since me might need more sophisticated querying features in datasets

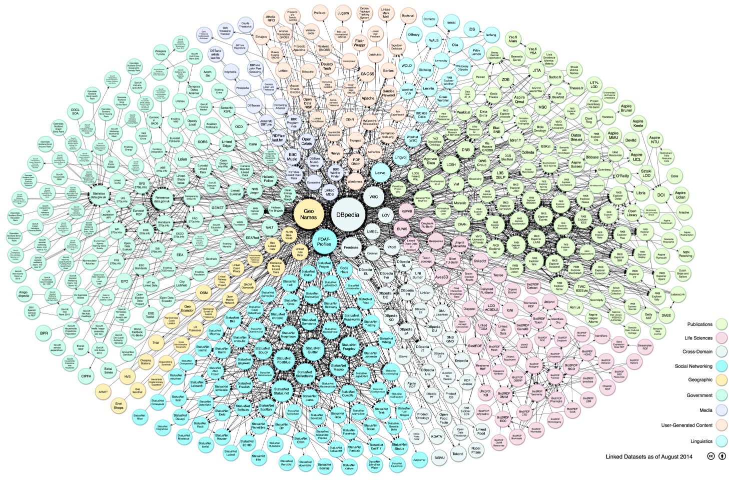


Figure 2.1.: Linking Open Data cloud diagram 2014[1]

with a low structuredness. This allow us to capture more fuzzy-querying features in order to retrieve desired information. Unfortunately, the data sets used in RDF benchmarks including LUBM [GPH05], SP^{2}Bench [SHM$^+$09], and BSBM [BS09] show a high level of structuredness in comparison to, e.g. DBpedia.

- **Diversity:** According to [DKSU11], a high diversity is attributed to datasets which have instances with various levels of structuredness ranging from low to high. The more common but less analytical interpretation is based on the key idea of the Semantic Web, where the usage of distinct IRIs enables interlinking across various domains and application fields. Hereby, a high diversity is obtained if a graph exhibits a heterogeneous structure, where probably multiple graphs from different domains are brought together by the use of Semantic Web principles. The largest example for highly-diverse data is the so-called LOD cloud[3] illustrated in Figure 2.1 which follows the Linked Data principles [BHB09], a collection of best practices for publishing and connecting semantic data on the web.

- **Supported Querying Features:** With the emergence of diverse RDF datasets which interconnect various domains by means of ontologies and taxonomies, complex and comprehensive knowledge graphs are created. Such knowledge graphs are revealed to be the real benefits of Semantic Web technologies, as they may contain even until-now *hidden* pieces of information emerged from a semantic interconnection of multiple data sources. However, retrieving such knowledge requires more sophisticated querying features than offered by e.g. SPARQL 1.1 and is therefore mostly neglected in RDF benchmarks [PSHT13, EAL$^+$15]. As a result, the ability to test such complex query-

[3]Max Schmachtenberg, Christian Bizer, Anja Jentzsch and Richard Cyganiak. http://lod-cloud.net/

ing capabilities is becoming more and more indispensable for the development of modern RDF data management systems. It is therefore crucial to (1) provide representative datasets that also contain more complex structural properties and ontologies, which allow (2) the performing of experiments with more sophisticated querying features that use for instance *topological information* described by ontologies.

- **Representative Queries:** The authors in [MLAN11] suggested the use of query logs in order to derive relevant templates for queries. However, it is an open question what representative queries actually are and how to define them. Experiments on more meaningful queries which exploit the real benefits of Semantic Web technologies go far beyond the capabilities of queries which are derived automatically from collected query logs, as they exhibit mostly rather simple querying expressions [MLAN11]. For this, experts with knowledge about the data are needed to express queries in order to test certain aspects. On the other hand, it is always questionable how representative and common such artificial queries might be in practice, since they are always geared towards a predefined use case.

To the best of our knowledge, there was – at the point of starting on this work – not one ready-to-use RDF benchmark available, which combined all those features in one concept. We will continue with a short overview on RDF benchmarks where we describe their relevance for this dissertation and characterize them in accordance with our desired features.

LUBM. While being originally developed to benchmark reasoning capabilities on RDF data, Lehigh University Benchmark (LUBM) [GPH05] evolved to become one of the most-used test suites for the performance of join evaluation strategies in distributed environments and single machine approaches. Its wide use originates from a scalable data generator and a set of predefined queries that break down into the computation of joins, without demanding more expressive operators. Consequently, it comes along with a rather structured and more relational dataset that does not capture any sophisticated navigational querying features but is well-suited for comparing the performance of join evaluation strategies. Due to its wide acceptance, which facilitates a better comparability to related work, we will use LUBM in order to investigate the performance of our map-side merge joins introduced in Section 6.5.3.

WatDiV. The Waterloo SPARQL Diversity Test Suite (WatDiv) [AHOD14] was developed particularly in light of highly diverse, heterogeneous RDF datasets discussed earlier. It proposes a data and query generator with queries that have a higher diversity resulting in much more varied workloads. For this, varying query shapes are used which test RDF management systems covering a much wider range of problems. However, the initial WatDiv does not contain long-chained, navigational queries. We have therefore contributed in designing an additional WatDiv use case called *Incremental Linear Testing* [SPSL16], which is part of the Ph.D dissertation of Alexander Schätzle [Sch16] and is therefore not discussed here. The queries were meanwhile added as an extension to the official set of tests[4]. Due to their long-chained shapes they are well-suited as the basis for our experiments in Section 8, where we compare the performance of our implementations with competitors. Nonetheless, WatDiV does not model social interactions between users; it is built around an e-commerce scenario and lacks the support for certain correlations common in social networks for which we need another benchmark.

LastBench. With our Social Last.fm Benchmark (LastBench) [PSHT13], we propose a real-world social network benchmark based on data gathered from *Last.fm*. The main idea is to model realistic data that captures the diversity of Semantic Web by providing resolvable links to various datasets while allowing to scale the dataset. The data itself enables the expression of comprehensive navigational querying patterns which still remain easy to understand, due to the well-known nature of the music-listening scenario. Moreover, by considering the complex structure of real interaction in social networks, we grasp various probably-unknown correlations which pose interesting challenges for RDF management systems and their query languages which were at the time of developing this proposal neglected by most of the existing benchmarks. The proposal of LastBench will be discussed in Section 2.2, where we mainly focus on how to obtain and scale the dataset. However, LastBench won't be used in this dissertation but its principles are nonetheless worth discussing as they were taken up by more recent approaches [EAL+15, AHOD14].

LDBC Social Network Benchmark (SNB). The LDBC Social Network Benchmark (SNB) [EAL+15, PBA+16] is designed to test various graph data management systems ranging from graph databases to RDF stores. Its main idea is to mimic operations of real social networks by providing both, (1) a suitable dataset generator that creates social network data with most common entities (e.g. person, posts, tags, cities,...) and (2) three different workloads that pose a wide range of interesting querying challenges. SNB is the follow-up work of Social Intelligence Benchmark (SIB) [PBE12] which came out of an EU project[5] and gained a lot of attention from both industry and academia. Due to its suitability for most examples and experiments required in the context of this dissertation, the adoption of industrial use-cases and the fast development under the head of the Linked Data benchmark (LDBC) Council[6] we decided to base some of our experiments on SNB.

[4]http://dsg.uwaterloo.ca/watdiv/#tests
[5]`http://cordis.europa.eu/project/rcn/105871_en.html`
[6]`http://www.ldbcouncil.org/`

Others. The SPARQL Performance Benchmark (SP^{2}Bench) [SHLP08, SHM+09, SHLP09] is built upon a data generator that creates data publications based on observed characteristics from DBLP. Its main idea relies on rather challenging, complex SPARQL queries; this is why it was also referred to as the gold-standard for benchmarking RDF management systems. However, the proposed queries are using only SPARQL 1.0 features and there is no way to express meaningful long-chained queries. The principles of the DBPedia SPARQL Benchmarks [MLAN11] were already mentioned multiple times. It features a real-world dataset based on Wikipedia, enhanced with ontologies and links to other data sources. However, scaling the dataset proved to be problematic, which limits the meaningfulness of experiments on distributed systems. The representativeness of automatically-generated queries is at least questionable since it contains mostly basic querying features. The Berlin SPARQL Benchmark (BSBM) [BS09] in its second versions features only SPARQL 1.0 features. More complex queries that support grouping, aggregations and sub-queries were introduced with the third version, whereby its e-commerce scenario is also impeding the construction of long-chained queries which are crucial for experiments of navigational querying features.

Table 2.1 shows an overview of the most-used RDF benchmarks with regard to our desired features. Please note, that this table is not intended to provide a comprehensive comparison of all relevant RDF benchmarks with their distinguishing feature. It is just meant to illustrate a briefly classification of the most-used benchmarks.

Table 2.1.: Overview of benchmarks using our features: † supports only basic graph pattern, ‡ supports grouping, aggregates and sub-queries

	Year	Realness of Data	Structuredness	Diversity	Social Graphs	Scalability	Predefined Queries	Querying Features	Classes	Properties	Scenario
LUBM	2005	×	high	low	×	✓	✓	SP 1.0 †	43	32	Universities
SP2Bench	2008	×	high	low	×	✓	✓	SP 1.0	8	22	Publications
BSBM v2	2009	×	high	low	×	✓	✓	SP 1.0	8	51	E-Commerce
BSBM v3	2011	×	high	low	×	✓	✓	SP 1.1 ‡	8	51	E-Commerce
DBPSB	2011	✓	low	high	×	×	×	SP 1.0 $	239	1200	Wikipedia
SIB	2012	×	mid	mid	✓	✓	✓	SP 1.1	14	53	Social Network
WatDiv	2014	×	mid	mid	×	✓	✓	SP 1.1	–	–	E-Commerce
SNB	2015	×	mid	mid	✓	✓	✓	SP 1.1	13	35	Social Network
SPB	2015	×	mid	mid	×	✓	✓	SP 1.1	–	–	Publishing
FEASIBLE	2015	✓	low	high	×	×	×	SP 1.1	–	–	Customizable
LastBench	2013	✓	mid	mid	✓	✓	–	–	–	–	Social Network

All in all, we can conclude that choosing a proper benchmarks is crucial for the development of a modern RDF management system. There is a wide variety of possible features, which might be relevant, dependent upon the intended use cases. In each case, compromises become inevitable since a *one-size-fits-all* solution does not exist which captures all kinds of desirable features, so it is in general advisable to use more than one benchmark. With LastBench, we propose an RDF benchmark that intends to fulfill most of the aforementioned features which we believe to be important for the query languages and systems presented within this dissertation.

2.2. Designing a Benchmark from Social Network Data

Social networks contain manifold *semantic information* about users, objects and their interactions. In order to capture such data and its inherent correlations as closely as possible, we need to overcome the conceptual shortcomings of synthetic data generators. For this, we proposed a benchmark that does not only mimic observed structural correlations but consists completely of real-world data gathered from social networks. However, a benchmark that uses data from social networks imposes multiple challenges that need to be considered. First of all, there are legal issues with data that can be used to reveal the true identity of a user. For this, an anonymization strategy need to be applied which ensures that no inferences from data about real persons can be drawn. Furthermore, large social platforms like Facebook make a profit from the knowledge contained in their data and often prohibit the usage of it at larger scales. Fortunately, with Last.fm we found a social network where its data (1) fulfills most previously mentioned features that we consider to be important for our work and (2) does not impose extensive legal restrictions on its usage.

Last.fm is a social music network with a community formed around the idea of sharing individual music listening behavior. It has a continuously-growing user-base with diverse information about millions of artists, events, tracks and users including billions of relations between all of them. Standardized identifiers in music collections and links to other commonly-used data sources like Wikipedia facilitate the interlinking with other domains which in turn provides more flexibility in composing a benchmark dataset. The collected data exhibits – which is of particular importance – characteristics of a social network which we will investigate in Section 2.3.2. From a legal aspect, the Last.fm Terms of Service (TOS)[7] allow the non-commercial usage of their data which is provided via a rich public API[8]. By making use of ontologies from the music domain and a mapping to an intuitive taxonomy, we can further enhance the knowledge-representation of our benchmark dataset. This in turn en-

[7]`http://www.last.fm/api/tos`, 2014
[8]`http://www.last.fm/api`

ables the support for more sophisticated querying features that for instance require reasoning capabilities to retrieve certain information. An inherent complex graph structure combined with many unknown correlations inside the data poses new challenges regarding query evaluation and enables more realistic, less predictable and therefore more meaningful experiments. These new features were neglected or not considered at all by most RDF benchmarks.

We will continue with a short overview of the Last.fm data that we use as a basis for LastBench. Afterwards we will introduce our data sampling strategies which aim for mainly two aspects: (1) being representative with regard to the whole dataset and (2) providing good coverage with many relations and entities of Last.fm. We will present our developed LastBench Crawler. This is followed by explaining our scaling strategy, which keeps the characteristics of the dataset by utilizing some properties of our sampling strategies. Lastly, we show a preliminary analysis of the underlying social graph of Last.fm followed by a discussion on the latest state of LastBench.

Last.fm Data for LastBench. Last.fm is an online music service with myriad relations between users, artists, tracks, etc. that constitute a highly-connected graph with a large variety of correlations. Figure 2.2 shows an outline of relevant entities and their relations, which we will consider for the benchmark dataset. As an example of properties, we added two classes, namely `user` and `track` in Figure 2.2.

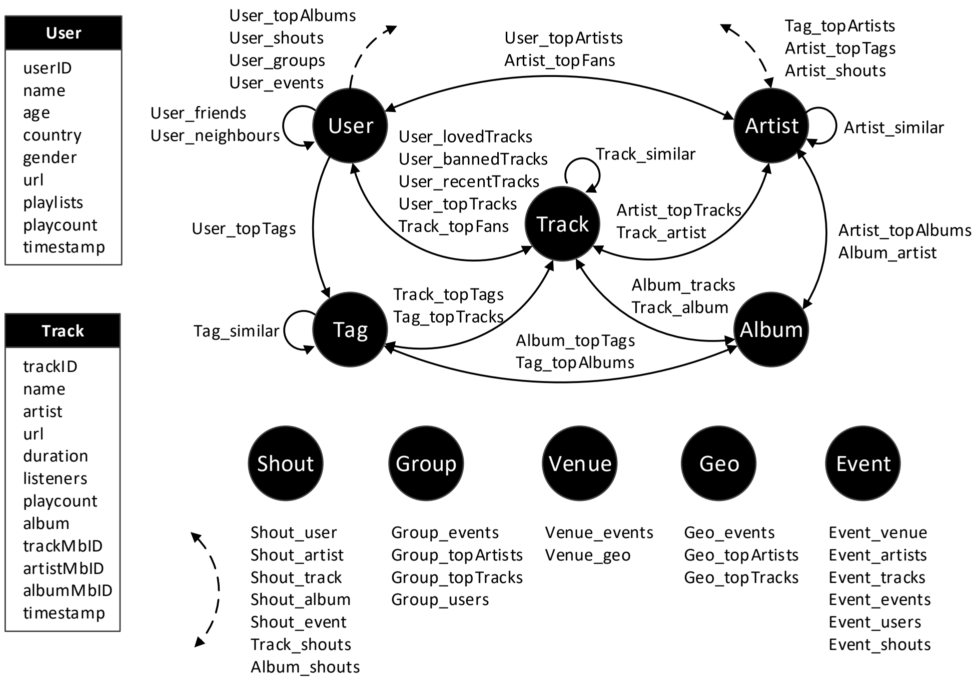


Figure 2.2.: Overview of relevant Last.fm classes and relations for LastBench

From this rather relational view on the data, we have to move towards a meaningful RDF dataset. For this, we have to introduce a mapping which reuses, where possible, existing taxonomies and ontologies such as the FOAF-Ontology[9]. Nonetheless, in order to enable testing of more comprehensive querying features that also capture reasoning and arbitrary recursions, it is also necessary to introduce also a few extensions. The work done so far focuses on data sampling strategies, which we will introduce in the next section. An exemplary RDF graph, which illustrates how a user and track could be represented is shown in the following snippet:

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix lb:   <http://dbis.informatik.uni-freiburg.de/lastbench/> .

lb:Bob a foaf:Person , lb:User ;
         foaf:age "30" ;
         lb:lovedTrack lb:track1 , lb:track2 , lb:track3 , lb:track4 .

lb:track1 a lb:Track ;
           lb:artist lb:artist1 ;
           lb:topFan lb:Bob , lb:Alice , lb:Ted .
```

2.2.1. Sampling Strategies for Last.fm (and other OSNs)

Next, we will discuss how to obtain a representative dataset from Last.fm with all of the entities, relations and properties presented earlier. As there is no complete data dump available for download, we rely on the official Last.fm API for data collection. With a developer key, we had access to all of the desired information. However, collecting data from the web is a non-trivial task especially where original characteristics and correlations need to be preserved [LF06]. In the context of *online social networks* (OSNs), the so-called *graph sampling techniques* are used to obtain a representative subset from a larger graph. They are well-studied in recent years and are used, for instance, in the analysis of Facebook, where representative subsets were crucial to derive trustworthy results [LF06, YLW10, GKBM10, CMF^{+}11, GKBM11]. These approaches can also be adopted for sampling our benchmark dataset from Last.fm [GBKM11].

There are mainly two types of strategies for graph sampling. The *first* considers nodes that are iteratively randomly chosen. With a certain size, the sampled graph will converge in such a way that it exhibits representative characteristics of the original graph. However, in order to achieve a uniform selection of nodes, knowledge about the range of values, e.g. IDs, is required. Furthermore, it cannot be guaranteed that the resulting graph is connected, which hampers meaningful experiments

[9]`http://www.foaf-project.org/`

where data size needs to be varied and one large connected graph component is a desirable feature [LF06]. For the *second* strategy, nodes are chosen by traversing the graph structure. A commonly used approach is the well-known *Breadth-First-Search* (BFS). But sampling OSNs with BFS introduces *biases* in terms of degree distribution and clustering coefficient [KMT10, YLW10] as it tends to visit nodes of high degree to the detriment of nodes with lower degree [GKBM10]. As we want our sampled graph to provide an as accurate as possible picture of the whole graph, including its correlations and representative characteristics, more sophisticated strategies need to be considered. The *Re-Weighted Random Walk* (RWRW) and *Metropolis-Hasting Random Walk* (MHRW) are two strategies based on a random traversal of the graph structure that are shown to produce accurate samples of large OSNs in recent studies [GKBM10, GBKM11, CMF^{+}11, GKBM11]. For the remainder of this chapter, we will use the term *"walk"* to refer to *"traversing the graph structure"* by one of both strategies.

The authors in [GBKM11] proposed to use *Re-Weighted Random Walk* (RWRW) for crawling Last.fm. Using this strategy one is able to correct the bias towards high degree nodes, but this correction is done afterwards [GKBM10] in a post-processing step. Thus, we choose to use *Metropolis-Hasting Random Walk* (MHRW) proposed in [GKBM10, CMF^{+}11, GKBM11], which corrects the bias directly during the *walk* by adapting transition probabilities at the cost of losing some efficiency compared to RWRW. This has mainly two advantages for us. First, it is easier to distribute the workload over a cluster of machines since the data will already be non-biased once it is crawled and there is no need for a centralized processing step afterwards. Secondly, as data grows incrementally, we can stop and resume the whole crawling step and have at each time a ready-to-use dataset that is not biased.

The basic strategy of MHRW (cf. Algorithm 2.2.1) is a *random walk* that starts at an initial node v, where the next node w is selected randomly out of all neighbors of v. Let assume p to be a randomly generated number such that $0 \leq p \leq 1$ and the degree of a node i is denoted by d_i. The randomly chosen neighbor w is visited next, iff it holds that $p \leq \frac{d_v}{d_w}$. Otherwise this step is repeated again by selecting a new random node out of all neighbors of v. According to this, nodes with a higher degree have a lower probability to be selected than nodes with a lower degree, which in turn corrects the bias. The end of a walk needs to determined by a convergence criteria as introduced in [GBKM11], which basically recognize if a random walk is already representative enough for the sampled social network. Generally, such criteria are based on some measurable characteristics which are applied on the data collected so far on a walk. If the measured characteristics do not changes any more with further steps or are less then a specified threshold, a walk is considered to be convergent and can be ended. A detailed description can be found in [GBKM11].

Another important aspect that we need to consider is the so-called *burn-in*. It denotes a set of nodes from the beginning of each walk, which need to be discarded in order to avoid the dependency from the initially-chosen seed node. The actual number of nodes that need to be discarded can be either determined afterwards by using the same approach as done for determining the convergence or just estimated with a few experiments.

Algorithm 2.2.1 : Extended MHRW

```
input  : $v$, initial seed node,
         $E_{mhrw}$, set of relevant edges for MHRW,
         $E_{2hw}$, set of relevant edges for 2-HW
output : all visited or seen *nodes* their *relations* and *properties*
1  while convergend criteria not satisfied do
2      $w \leftarrow getRandomNeighbor(v, E_{mhrw})$ ;        // following edges $\in E_{mhrw}$
3      $p \leftarrow random(0, 1)$ ;     // random rational number, where $0 \leq p \leq 1$
4      if $p \leq \frac{d_v}{d_w}$ then
5          $v \leftarrow w$;
6          $i \leftarrow randomInt(0, 2)$ ;          // random integer, where $0 \leq i \leq 2$
7          if $i > 0$ then
8              $2hw(v, E_{2hw}, i)$;
9          end
10     end
11 end
```

One of the main reasons for using Last.fm is that it not only describes a social graph with users, but it contains also manifold other entities with further relations. We refer to such a graph also as a so-called *multi-graph*. The basic MHRWs or RWRWs cannot sufficiently capture such multi-graphs and are therefore not able to consider this information satisfactorily. For this, we need to introduce some algorithmic modifications and extensions. The authors in [GBKM11] suggested a two-stage approach where in the first step individual RWRWs are applied on each relation that interconnects users, namely *friends*, *groups*, *events* and *neighbors*. Afterwards, all sampled datasets are grouped together. Their experiments have shown promising results. However, as we are not only interested in interconnections between users but also for further relations and entities as shown in Figure 2.2, we need a more sophisticated approach. For this, we modified the MHRW to work on *sets of relations* and introduced an extension for MHRW called *2-Hop-Walk* (2-HW). The main idea behind our MHRW extension is the ability to traverse graphs using a set of relevant relations as basis. So we gain the flexibility to run each MHRW on a different set of relations if desired. With this we are able to capture all kinds of multi-graphs. The basic goal behind 2-HW is, for each node visited by MHRW, to increase the number of links to other entities, so that we obtain a more diverse connected graph. For example, if we traverse mainly the social network of users with the MHRW, we can collect more information about the neighborhood of each user, e.g. favorite tracks or visited concerts. Here we might obtain intersections with other users who liked the same or similar tracks or had visited the same concert in another year. This in turn increases the probability of creating more connections within the sampled data. It also increases the density of sampled graphs while consequently reducing the overall diameter. Algorithm 2.2.1 shows the modified MHRW approach and Algorithm 2.2.2 its 2-HW extension. Note that, to improve readability of both algorithms, we illustrated solely the traversing aspects and left out any data-storing operations. A more technical view will be discussed in Section 2.3.

Algorithm 2.2.2 : $2hw()$: extension for MHRW

```
input  : $v$, initial seed node,
         $E_{2hw}$, set of relevant edges for 2-HW,
         $i$, recursion steps for 2-HW
output : all visited or seen *nodes* their *relations* and *properties*
1 if $i > 0$ then
2     $R \leftarrow getAllNeighbor(v, E_{2hw})$;
3     for $r \in R$ do
4         $2hw(r, E_{2hw}, i-1)$;
5     end
6 end
```

Example 2.1. Assume an iteration of MHRW which collects at first all properties of the currently-visited node v, e.g. for a user its `userID`, `name`, `age`, `country` and so on (cf. Figure 2.2). A 2-HW traverses then the desired neighborhood of v, where either all relations or just a relevant set of relations can be specified by the parameter E_{2hw}. Here, we could consider for example the relations `visitedEvents`, `lovedTracks`, or `groups`. The actual depth of the recursively traversed neighborhood is determined by a randomly generated value between 0 (skip neighborhood) and 2 (grasp the whole 2-hop neighborhood). Once we reach for example the set of visited events by following `visitedEvents`, 2-HW also retrieves, in the case of a recursive depth > 0, for each event e.g. the artists who performed or other users who participated. These randomly-generated numbers reduces, on the one hand, the overall structuredness, but on the other hand the averaged costs of executing 2-HW, since a depth of 2 can be rather costly in cases where many relations are considered. Once the 2-HW is completed, MHRW choses a neighbors of v to be visited next, while again considering either all relations or just those which are defined by the parameter E_{mhrw}. For instance this might be `friends` and `neighbors` or in cases of multi-graphs even `lovedTracks`. The sets of relations can be specified for each initialization of a new walk individually. This way, we bring in enough flexibility to traverse the graph we are interested in, but also create a network which exhibits desirable features. □

2.3. A Distributed Crawler for Last.fm

The first component that was required for the LastBench benchmark was a distributed *crawler* which implements the previously introduced sampling algorithms. We will refer to this component as LastBench Crawler, or just crawler in the following pages. For Last.fm, we could base our code on available JAVA-bindings[10] that simplified the communication with the Last.fm API[11]. However, they needed to be extended and adapted to comply with our algorithms. Our design goals for the crawler can be summarized as follows:

Scaling: The final dataset targets several hundred million to a billion entries. The technologies, storage schema and algorithms should consider this scale, as operations on data are becoming more expensive the larger the datasets get.

Parallelization: It is required to be able to run multiple *walks* in parallel on different machines while utilizing data collected by all individual parallel walks in real-time.

Fault-Tolerance: A walk should be resumable, if it is interrupted or it crashes. Failures might also appear while walking, e.g. blocked IP, corrupted data from Last.fm, etc. . Resuming an interrupted walk should always be possible without leading to strangling or corrupted data.

Performance: As we want to collect data as quickly as possible, performance and efficiency are of the utmost importance. The most expensive operations are Last.fm API calls. The longer we collect data, the more likely it will be that a walk reaches a node that has been already visited in the past. In such cases, no Last.fm API calls should be made at all. Furthermore, the more data is stored, the more expensive certain operations on the data will become, which makes it even more important to choose them carefully.

Figure 2.3 shows an overview of our architecture of which the implementation is available for download. A more detailed view on the tasks of each component is described in Table 2.2.

A centralized storage system that uses a dedicated infrastructure enables the parallelization of crawling processes by allowing instances to be run independently on arbitrary machines. Each instance will then communicate with the same storage system, which – due to its distributed architecture – is able to handle a high throughput of requests. To increase the efficiency of collecting data, the whole implementation is meant to start independent walks in parallel on multiple machines. For each new walk, as suggested by the authors in [GBKM11], a new random seed user is chosen by, e.g. crawling recent activities on the Last.fm Website, where the independence of a node is ensured by considering the discussed *burn-in*.

[10]`https://github.com/jkovacs/lastfm-java`
[11]`http://www.last.fm/api`

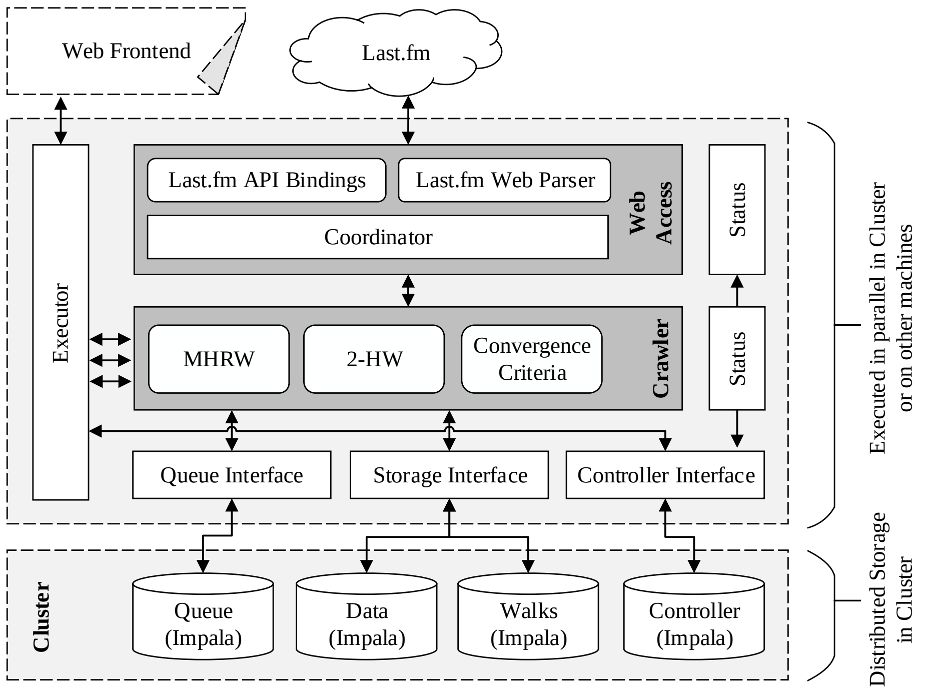


Figure 2.3.: Architecture of LastBench Crawler

In order to next present our approach for scaling the final LastBench benchmark dataset, we need to explain a few more technical details on how a walk is represented in the storage system (cf. *Walks* database in Figure 2.3). Table 2.3 shows the schema of a walk and its steps. We can see that each new walk gets an ID assigned, which is increased incrementally. In addition, the ID of the seed node, the chosen strategy and the current status is saved. Each instance of a crawler is then responsible for exactly one walk ID, where in each iteration, and thus visit of a new node, a new step is created. The status field ensures that stopped or interrupted walks can be resumed without introducing any corrupted information. For that, the last step which was inserted for a walk is restarted completely as it might not be processed completely. For that reason but also to increase the overall efficiency, the *Storage Interfaces* checks that there are no duplicate entries inserted, in cases where for instance a step crashed before it was finished or an entry was already processed in the past. Lastly, the final LastBench benchmark dataset is composed of a union of all finished, and thus convergent, walks.

2.3.1. Scaling LastBench Data.

One of the main challenges in using real-world data is the scaling of datasets, that is removing or adding some data in order to obtain a certain size. This is of particular interest in the area of distributed systems, where it is essential to investigate the scaling properties of RDF data management systems and their languages with regard

Table 2.2.: Description of LastBench Crawler components

Component	Description
Web Access	The actual crawling part, thus retrieving the data from Last.fm is contained in this component. Data is crawled by using the public API of Last.fm by means of an extended version the Last.fm API Bindings. As an example, the parsing of the website Last.fm is required in order to retrieve initial seed nodes.
Crawler	The crawler implements our modified extended MHRW algorithm with its 2-HW extension and convergence criteria as introduced in [GBKM11]. This defines the strategy how the Last.fm graph is sampled. Before continuing with a new node, i.e. using the *Web Access* component, it is checked whether the desired node has already been visited and is stored in the database. Revisiting nodes on a walk is allowed and important, but equally important is ensuring that doing so does not induce any additional Last.fm requests.
Storage Interface	There are two main data stores, to which access is managed by this interface. The *Data Store* contains all entities, relations and attributes collected by the walks. To this end, each entity is stored just once. The *Walks Store* contains meta-data about ongoing and completed walks. More information about how walks are represented will be discussed in the following section.
Queue Interface	It represents the queue of ongoing walks by individual *FIFO* list of nodes that have to be visited next. This is required, for instance, if a list of neighbors needs to be iteratively processed.
Controller	It keeps information about the status of ongoing and completed walks. Additionally, it provides information required to resume previously stopped or interrupted walks.
Executor	It starts, stops and resumes walks, controlled by the *Web Frontend*.
Status	It monitors ongoing walks and pushes information to the *Controller Interface*, e.g. failures, status updates, processed or dropped items, ...
Web Frontend	It provides information about ongoing walks, e.g., current progress, number of newly inserted and already present entities, number of request, failures and the performance of all components.

to varying data-sizes. Here, an error-prone scaling of the dataset might have a high impact on the evaluation complexity if, for instance, the majority of relevant nodes are removed accidentally. This yields less trustworthy results, which in turn hampers the usage of more meaningful real-world datasets for such experiments. *Up-scaling* a dataset means that some new data has to be added, which was not part of the original dataset. No matter how good the estimations for correlations and distributions are, we are not guaranteed to have full knowledge about all characteristics of the

Table 2.3.: Data schema for walks and the steps a walk is composed of

	Attribute	Description
walks	**walkID**	Identifier of a walk, increments for each new one
	startNode	ID of seed node
	strategy	Crawling strategy of walk, e.g. BFS, MHRW, 2-Hop & parameters
	status	0: initialized, 1: paused, 2: finished, 3: failed
steps	**walkID**	Reference to the walkID in walks
	walkStepID	Identifies a step of a walk, increments with each new step
	visitedNode	ID of the currently reached node
	startTS	Timestamp, when this step starts
	endTS	Timestamp, when the step ends
	status	0: initialized, 1: paused, 2: finished, 3: failed

dataset. That is, with additionally-generated information we will most likely change the properties of the original dataset by, e.g. missing some important connections for new entities. As a result, we decided against an *up-scaling* strategy for LastBench and focus on a proper *down-scaling* strategy, which – if the collected data is large enough and accommodates future growth – fits most use cases.

For that, we utilize two important properties, which are inherent to the representation and processing of a walk. For the *first* property, since for each walk it is ensured that (1) its burn-in was accounted for, i.e. its dependency from the initial seed node were removed and, (2) only finished walks were considered, i.e. the convergence criteria were always fulfilled, we can assume each individual walk to be representative of Last.fm. To the *second* property, a walk is always independent from all others, which is inherent to our implemented sampling strategies, since a crawling instance is, at each point in time, responsible for exactly one walk. That is why it is not necessary to consider *all* walks to get a representative dataset of Last.fm; even one walk would be enough. Conveniently, for down-scaling the LastBench dataset, we can utilize this property by reducing the total number of walks in the final dataset. More details are introduced with Algorithm 2.3.1, which is a deterministic version of our approach that composes for the same scale-down always the same subset of the original data. To obtain finer-grained subsets, in a last step where the specified down-scale cannot be achieved by a composition of complete walks, single steps are added until the desired size is reached.

2.3.2. A Preliminary Analysis on LastBench Data.

In a preliminary analysis of our data, we were interested in the underlying social network of Last.fm and its typical characteristics [WS98]. For that, we obtained a small sample dataset with 1.7 million users with close to 13.6 million friendship relationships using a Breadth-First Search (BFS) strategy since the previously-introduced

Algorithm 2.3.1 : Down-scaling LastBench dataset

```
input  : $S_t$, desired size of the down-scaled dataset in triples,
         $W$, ordered list with all walks
output : $R$, down-scaled dataset
1  $R \leftarrow \emptyset$;
2  $R_{tmp} \leftarrow \emptyset$;
3  $S_r \leftarrow 0$;
   // Adds complete walks while $S_r < S_t$;
4  while ($S_r < S_t$ && $W.hasNext()$) do
5      $R_{tmp} \leftarrow W.getNextWalk()$;
6      if $S_r + |R_{tmp}| < S_t$ then
7          $R \leftarrow R \cup R_{tmp}$;
8      else
           // Adds single steps of a walk while $S_r < S_t$;
9          while ($S_r < S_t$ && $R_{tmp}.hasNext()$) do
10             $R \leftarrow R \cup R_{tmp}.getNextStep()$;
11         end
12     end
13 end
```

crawling strategies were not available at the point of performing this analysis. Although BFS introduces a bias in terms of *degree distribution* and *cluster coefficient* [KMT10, YLW10], the results were sufficient to make a few preliminary conclusions about the data. First of all, we investigated the degree distribution which is crucial in order to characterize a network as a social network. The degree of a node is defined by the number of links incident to a node. In Figure 2.4 (a) the degree distribution is *skewed* with the majority of nodes having a low degree while very few nodes have significantly higher degree. This is a typical behavior of social networks. The clustering coefficient is another important characteristic of social graphs and represents the tendency of nodes to form tight clusters with its neighbors. This metric is defined as the number of links that exist between a node's neighbors divided by the maximum possible links that could exist among a node's neighbors. The clustering coefficient in a social network is higher than in other types of networks [NP03]. Figure 2.4 (b) depicts the average clustering coefficient with regard to degree. We can observe that low degree nodes demonstrate a higher clustering coefficient which means that there is a significant clustering among them. On the other hand, as the number of neighbors increases, the clustering coefficient drops. These results are consistent with previous research on social networks [MMG$^+$07].

Furthermore, short paths in the network indicate that nodes are *reachable* through a small number of hops. The average path length of the network is 4.2 and is even shorter than the expected famous *"six degrees of separation"* of Milgram's experiment [Mil67, WS98]. This surprisingly low average path length is influenced

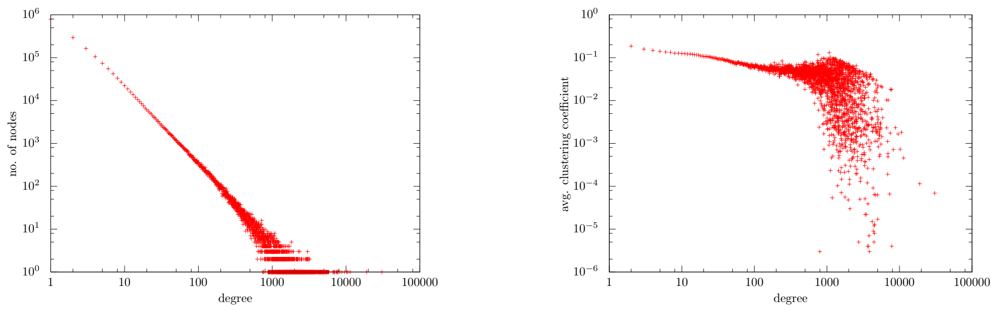


Figure 2.4.: (a) Degree distribution (left), (b) Avg. cc-degree distribution (right)

by the bias of BFS towards the high degree nodes which tend to reduce distances in the network. Another characteristic of social networks is the largest shortest path, the so-called diameter. The network has a diameter of 8 which again is low in comparison with other social networks [MMG+07] which is also probably due to the bias introduced by the crawling technique. The aforementioned skewed degree distribution, high clustering coefficient, and short path lengths are typical social network properties and indicate that our Last.fm data has small world characteristics and typical scale-free properties of a social network [WS98].

Overall, we can summarize that the measured metrics for the underlying social network of Last.fm, although obtained with BFS and not our extended MHRW strategy, already exhibit typical characteristics of a social network which was one of the requirements for our benchmark as stated in Section 2.1.

2.4. Discussion

With the evolution from a *"Web of Documents"* to a personalized and interlinked *"Web of Data"*, social networks are becoming an important source of semantically annotated data. Nonetheless, they are hardly covered by existing RDF benchmarks. With LastBench, we have proposed a benchmark based on real-world data gathered from the social music network Last.fm. It is worth noting that, although introduced solely for Last.fm, all of our proposed algorithms are also adoptable to work on other social networks, e.g. Twitter or Facebook. Overall, we can summarize the work which was done so far as follows:

- With Last.fm we found a rapidly growing real-world data basis for our benchmark, which enables us to obtain hundreds of millions to billions of triples. It contains many interesting entities, relations, and properties with a social network structure.
- In order to obtain a representative dataset, we proposed a modified version of MHRW and extended it with our 2-HW strategy.

- We introduced our distributed LastBench Crawler that implemented MHRW and 2-HW in order to sample data from Last.fm.
- We suggested a novel strategy for down-scaling the LastBench dataset that utilizes some properties of our sampling strategies to create representative sub-sets.
- A preliminary analysis confirmed typical social network properties for a small data sample.

Recalling our requirements for a benchmark introduced in Section 2.1 on page 21, we can conclude that – even though in a early stage – LastBench satisfies already five out of seven features, that we considered to be important for the development of a modern RDF data management system (cf. summarize in Table 2.1 on page 26). Nonetheless, there is still some work left in order to complete LastBench. The remaining tasks can be summarized as follows:

1. Obtaining larger datasets by a long-term execution of LastBench Crawler
2. Specification of schema and mapping into RDF(S)
3. Incorporation of varying levels of structuredness based on [DKSU11]
4. Implementation of the proposed algorithm for down-scaling datasets
5. Designing multiple workloads with query templates and parameters

Although LastBench proved to be a promising way towards a scalable real-word social benchmark, we decided not to continue the work on the remaining tasks. This is due to mainly two reasons: driven by a large EU project[12] the LDBC Social Network Benchmark (SNB) [EAL+15] gained a lot of attention from both industry and researchers as a follow-up work of Social Network Intelligence Benchmark (SIB) [PBE12]. Under the head of Linked Data Benchmark Council (LDBC), it benefits from a fast development and many real-world use cases that were gathered. Secondly, the development of a benchmark was never at the core of this dissertation, but appeared to be – due to the lack of proper benchmarks in the past – necessary to support our research goals: *the specification of expressive, navigational query languages and the development of distributed systems that are able to process those languages on web-scale RDF data.* Due to its suitability for most of our examples and experiments required in the context of this dissertation, we decided to use the SNB dataset rather than continuing the work on LastBench. As shown in Table 2.1, it covers most of our desirable features except that it uses a state-of-the-art data generator rather than real-world data. However, the authors had a special focus on including distributions and correlations similar to those expected in a real social network such as Facebook. Moreover, it also contains some sort of real data from DBPedia, which is used as property dictionaries which ensures that the attributes are realistic and correlated [PBA+16].

[12]`http://cordis.europa.eu/project/rcn/105871_en.html`

Part III.

Navigational Query Languages for RDF

3. Querying RDF Graphs

Contents

Over the past decade, considerable work has been done on languages which allow the querying of semi-structured data such as RDF, ranging from the adoption of classical graph database approaches to highly expressive extensions of SPARQL. In this chapter, we provide some foundations of basic navigational constructs used in diverse query languages and provide an overview of the most-relevant (navigational) RDF languages. We will sum up this chapter with a comparison of representative languages using a set of features, that has been widely recognized as being crucial for (navigational) queries on RDF data which in turn motivates the need for both of our languages, RDFPath (Chapter 4) and TriAL-QL (Chapter 5).

We can summarize the contribution of this chapter as follows:

1. Section 3.1 gives a short, example-driven introduction to some basic navigational primitives and its terminology.
2. Section 3.2 presents an overview of languages that have been used to query RDF data. It starts with a discussion of languages for (semi)-structured data and their deviations, which are often used as a basis for navigational expressions. After that, the RDF-specific languages are discussed followed by extensions for SPARQL which add more expressive navigational constructs.
3. Lastly, Section 3.3 concludes this chapter with a comprehensive comparison of RDF query languages based on a set of querying features that have been widely recognized as crucial for the Semantic Web.

3.1. Foundations of Graph Query Languages

Most query languages for RDF have their roots in traditional graph databases, where so-called *regular expressions* [Kle51] are used as navigational primitives to traverse the graph structure. We will therefore start with introducing their basic concepts and terms. A brief example-driven description serves the intended purpose sufficiently. More formal specifications can be looked up in the given citations.

Definition 3.1 (Graph Database). A *graph database* G over a finite alphabet Σ is defined as $G = (V, E)$, where

- V is a finite set of nodes,
- $E \subseteq V \times \Sigma \times V$ is the set of edges. An edge in G is then a triple $(v_i, a, v_j) \in E$ interpreted as an a-labeled edge from v_i to v_j in G.

□

Figure 3.1 shows an exemplary graph G, which will be used in the following.

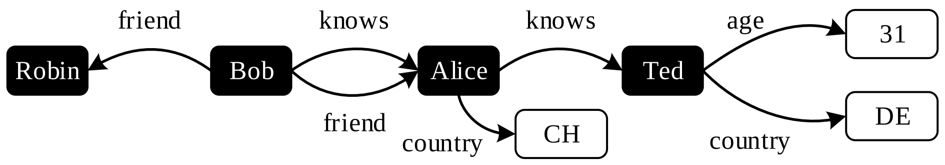


Figure 3.1.: Graph describing friendship relations between people

Regular Path Queries (RPQs) [CMW87, CM90, CDLV03] ask for pairs of nodes connected by a path of arbitrary length specified by means of so-called *regular expressions* [Kle51]. They allow the composition of patterns by describing *permitted* sequences of elements [MW95]. The basic operations include the grouping by parentheses, a Boolean "or" ($|$) , and quantifiers ($?$, $*$, $+$, $\{n\}$, $\{min, max\}$) to specify how often an element is allowed to occur. This language provides the fundamental basis for most navigational constructs in RDF query languages. More formally, assume a graph $G = (V, E)$, where V is a set of nodes and E is a set of edges. A RPQ has then the form $Q(x, y) = x \xrightarrow{L} y$, where:

- L is a regular language over some fixed finite alphabet Σ specified by a *regular expression.*
- The result of a query Q on graph G is then the set of all pairs of nodes (v_i, v_j) in G such that
  1. there exists a path π in G starting with v_i and ending with v_j,
  2. the label of π is a word from L [CMW87, CM90, CDLV03, LRV13].

An exemplary query that determines all pairs of users in a graph who are connected by a sequence of *knows* edges is described by

$$Q(x, y) = x \overset{knows^+}{\longrightarrow} y$$

Evaluated on the graph G in Figure 3.1, we obtain the following pairs of users:

$$\{(Bob,\ Alice),\ (Alice,\ Ted),\ (Bob,\ Ted)\}$$

Two-way Regular Path Queries (2RPQs) [CM90, CDLV00, CDLV03] extend RPQs by adding an inverse operation ($-$) that allows to traverse edges "backwards". With that, one can extend the above example to consider the edge *knows* in both directions (forward and backward). Consider for that the following 2RPQ query:

$$Q(x, y) = x \overset{(knows^+ \ | \ knows^-)}{\longrightarrow} y$$

Evaluated on the graph G in Figure 3.1, we obtain twice as many pairs, since *knows* is interpreted to be bidirectional in this case:

$$\{(Bob,\ Alice),\ (Alice,\ Ted),\ (Bob,\ Ted) \\ (Alice,\ Bob),\ (Ted,\ Alice),\ (Ted,\ Bob)\}$$

Conjunctive Regular Path Queries (Conjunctive RPQ, or CRPQs) [CM90] combine Conjunctive Queries (CQs) (cf. [Cod70, CM77] for more details) with RPQs, such that one can express conjunctions of RPQs (.. $\wedge$ ..) with existentially quantified variables (e.g. $\exists x$). A exemplary query can now ask for users which are, for instance, connected by a path of *knows* edges *and* by a path of $friend$ edges. The corresponding query is specified by:

$$Q(x, y) = (x \overset{knows^+}{\longrightarrow} y) \wedge (x \overset{friend^+}{\longrightarrow} y)$$

Evaluated on the graph G in Figure 3.1 we obtain, due to the restriction that pairs of users must be connected by both paths, just one pair:

$$\{(Bob,\ Alice)\}$$

A further natural extension are then Conjunctive Two-way Regular Path Queries (C2RPQs) including the inverse operation.

XML Path Language (XPath) [CD$^+$99, BBC$^+$03] is a W3C standardized language designed to query the tree structure of an XML document. With that scope it is certainly different to the querying constructs introduced so far. However, its rich yet intuitive navigational abilities cover many interesting features that include, for instance, different directions to navigate, diverse node tests, and branching. Due to this, the XPath-like syntax was picked up by various languages on other data models or serve as inspiration to design navigational querying constructs.

Nested (Regular) Path Queries (NPQ) [BFL09, ZLFB10, PAG10, BPR12] add existential tests to 2RPQs by means of nested expressions specified within square brackets and borrow the notation of branching from XPath. This way, they enable the expression of constraints within regular expressions that need to be satisfied along the path. More formally, a NPQ n is built from the following expressions:

$$n := \varepsilon \mid a \mid a^{+} \mid a^{-} \mid a^{*} \mid (n/n) \mid (n|n) \mid [n]$$

An exemplary query may ask for users connected by a sequence of *knows* edges, where each visited user along the path has to satisfy a constraint. Consider, for instance, the case where we want to ensure for each visited user the edge *country* exists:

$$Q(x, y) = x \overset{(knows[country])^{+}}{\longrightarrow} y$$

Evaluated on the graph G in Figure 3.1, we obtain just one pair of users, since there exists only one connection between users which satisfies the constraint of having an edge *country*:

$$\{(Alice,\ Ted)\}$$

3.2. Languages for Querying RDF

In the following, we provide a grouping of query languages which we want to introduce briefly[1]. Please note that, due to the high variety of query languages and many diverse properties they exhibit, groups are not necessarily distinct and therefore there may exist further possibilities of classifying them.

Querying (Semi)-Structured Data. Various languages deal with data represented as a graph, or so-called graph databases. Since RDF data can also be seen as a graph, these languages can be also used to query RDF data. Among them *G* [CMW87], its extensions *G+* [CMW88] and *GraphLog* [CM89] are the most influential proposals that support recursive patterns on graphs by utilizing *Regular Path Queries* (RPQ) [CMW87, CM90, CDLV03]. Due to this, navigational query patterns can be expressed, where the length of paths does not need to be known in advance, and thus can be of arbitrary length. This is useful in cases where we are interested in the transitive closure of a graph or the connectivity between arbitrary nodes. Essentially, query writing in those languages is based on specifying a graph pattern with annotated nodes and edges which are then again matched against the graph database.

[1]The basic concepts of RDF have been introduced in Chapter 1. A more formal description of RDF is not required at this point but can be looked up in Section 4.2 on page 58.

The idea of using RPQ has been also adopted for semi-structured data [Bun97, Abi97], i.e. incomplete data with an irregular structure. Substantial work has been done with *Lorel* [AQM$^+$97] and *UnQL* [BDHS96, BFS00], two languages built upon the so-called *object exchange model* (OEM) [PGW95] for modeling data. *StruQL* [FFLS97] is a language which claims to have a higher level of expressiveness than its former competitors, as it introduces a few enhancements to OEM regarding the support of schema but only UnQL supports the creation of new graphs as output. However, both the graph data model and the OEM for semi-structured data exhibit some inherent drawbacks which hamper their applicability for querying RDF data in practice. First of all, they use distinct identifiers for nodes (objects and subjects in RDF) and edges (property in RDF), where for instance the two RDF triples (s_1, p_1, o_1) and (p_1, s_1, p_2) cannot be modeled correctly. Next, they are restricted to so-called *simple paths*, which do not exhibit cycles. Furthermore, RPQs do not allow the expression of constraints within their regular expressions, e.g. a filter inside the recursive pattern.

Navigational & Rule-based RDF Languages. Rule-based languages describe another relevant class of query languages for RDF. *Triple* [SD01, SD02, DSB$^+$05] is a query, inference and transformation language derived from F-Logic [KL89], where triples are represented as F-logic expressions [HBEV04]. Those expressions can also query the schema of RDF, and thus its ontology, and apply transformations on RDF data. *N3Logic* (N3) [BCK$^+$08, BLC11] is an extension to the RDF model but is not meant to be a classical query language. Nonetheless, its rules can be used for querying, as they allow the composition of nested graphs and have support for variables. The main idea behind that approach is to have the same language for representing data and adding some sort of logic that describes further knowledge about the data like provenance, for instance. One of the first purely path-based language for RDF is *Versa* [OO02, OO03], which is inspired by XPath [CD$^+$99, BBC$^+$03]. It is based on an XPath-like syntax to traverse the XML serialization of RDF. The actual path is specified by a list of resources, where the result is a set of selected resources. Those resources can be further processed by externally-definable functions [FLB$^+$06].

SQL-like RDF Languages. SQL-like languages for querying RDF include *RDF Query Language* (RQL) [KAC$^+$02, KMA$^+$03, KMA$^+$04], its extension *Sesame RDF Query Language* (SeRQL) [BK03] and the two predecessors of SPARQL, namely *SquishQL* [MSR02] and *RDF Data Query Language* (RDQL) [MSR02, Sea04]. Common to RQL and SeRQL is their support for RDF with schema information [KBM08]. However, in comparison to more recent approaches [BFL09, PAG10, LRV13, AE14], they are rather restricted in this respect due to a strict separation between data (description of resources), schema (classification of resources) and meta-schema (meta-classification). In addition, certain preconditions on the data need to be satisfied, e.g. a property requires to have both domain and range clearly defined and cycles in the hierarchical schema data are forbidden [FLB$^+$06]. Both languages support variables on nodes and edges of the RDF graph. SquishQL introduced the concept of a conjunctive *triple-pattern* with variables in arbitrary positions, which correspond basically to *Conjunctive Regular Path Queries* (Conjunctive RPQ, or CRPQs) [CM90]. A select clause identifies the variables whose binding are returned as result. Both features were also present in a further derivative of SquishQL called RDQL. It is meant to provide a simple "low-level language" [Sea04] for RDF, but for that reason there is no support for querying schema information. The authors suggest this as a possible feature of an implementation rather than a query language. The concept of triple-pattern and variable bindings used in SquishQL and RDQL formed for the basis of their successor SPARQL [PS08].

SPARQL. SPARQL is the W3C recommended declarative query language for RDF [MMM14]. A SPARQL query defines a graph pattern P that is matched against an RDF graph G. This is done by replacing the variables in P with elements of G such that the resulting graph is contained in G (pattern matching). The most basic construct in a SPARQL query is a *triple pattern*, i.e. an RDF triple where subject, predicate and object can be variables, e.g. $(?s\ p\ ?o)$. If there exists a mapping between the triple pattern and a subgraph of the RDF graph, the variables of the triple-pattern are bound to the respective values of the subgraph. A set of triple patterns concatenated by AND (.) is then called a *basic graph pattern* (BGP). The result of a BGP is defined to be the intersection of all subsets defined by the corresponding triple patterns and can be computed by joining the results of all triple patterns on their shared variables. In the case of multiple possible mappings, sets of variable bindings are created. We refer to them as a bag of *solution mappings*. For a detailed definition of the SPARQL syntax we refer the interested reader to the official W3C Recommendation [PS08]. A formal definition of the SPARQL semantics can also be found in [PAG09]. The recent SPARQL 1.1 [HSP13] recommendation introduces further important operators including sub-queries, aggregates and most notably Property Paths which will be discussed next.

SPARQL Property Paths. SPARQL *Property Paths* [HSP13] are the navigational component of SPARQL 1.1 and correspond in its essence to Conjunctive Regular Path Queries (C2RPQs) [CM90]. Multiple property paths are then conjuncted by means of variables in the aforementioned triple-pattern. To understand the shortcomings with regard to the evaluation of *Property Paths*, we need to introduce some basic terminology of the semantics of SPARQL as proposed in [PAG09]. Let the set $\mathcal{T}$ of RDF *terms* be defined as $(\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$, thus a union of all IRIs ($\mathcal{I}$), blank nodes ($\mathcal{B}$) and literals ($\mathcal{L}$). Variables in SPARQL query are denoted by $\mathcal{V}$ and represent an infinity and disjoint set from $\mathcal{T}$. For simplicity, we will skip the complete semantics of SPARQL and assume a query result to be determined by a so-called multi-set of partial functions $\mu : \mathcal{V} \rightarrow \mathcal{T}$, mapping variables to RDF terms with respect to the function μ. Note that, in accordance to this definition, the result is composed of assigned variables *only*. However, the basic notation of a Property Path pattern is defined as $(\mathcal{T} \cup \mathcal{V}) \times path \times (\mathcal{T} \cup \mathcal{V})$, where *path* is composed from the following property path expressions:

$$path := i \mid {}^\wedge path \mid !(i_1 \ldots i_n) \mid (path \; / \; path) \mid (path \mid path) \mid path^* \mid path^+ \mid path^?$$

We can see that variables are allowed to be at the beginning or the end of a Property Path pattern but there exists no expression that would provide access to the complete path connecting both. At best, a pattern can be divided into separate parts and joined by means of an intermediate variable. However, this yields more complex queries and is not applicable to patterns involving the Kleene star operator (e.g. $path^* \mid path^+$), where the size of the path is not known in advance. The same implications apply for most other RDF languages, where the predicate is replaced by a regular expression (i.e. a RPQ) to describe a navigational pattern but no concept of a path variable exists. This includes also more expressive extensions of SPARQL like, for instance, RPL [BFL09, ZLFB10] and nSPARQL [PAG10, BPR12] which we will discuss in the following paragraph. They are both restricted to determine the existence of a certain path in a graph, and thus output its start and end point but do not provide a mechanism to retrieve further information about the path.

Extending SPARQL. Although having emerged as the standard query language for RDF in its first specification, SPARQL 1.0 provided just limited support for navigational querying. Solely paths of fixed length were expressible by means of conjunctive triple-patterns. This fact motivated several extensions of SPARQL, which address the need for more expressive navigational querying that include also paths of arbitrary length. The first considerable proposals are *SPARQLeR* [KJ07] and *SPARQ2L* [AMS07]. Both introduce a so-called *path variable* in a graph pattern to represent matched paths and use filter operations on that variable to constrain valid paths. Using a path variable allows access to all individual elements such that complete paths become retrievable. This is a valuable and important functionality, since it allows to discover paths of arbitrary length and output them. One drawback that both languages have in common is their inconvenient and inconsistent syntax. The only way how to describe a desired path pattern is by means of *filter functions* and so-called *meta-properties.* A filter function takes the *path variable* and a regular expression as parameters, where only those paths are permitted which satisfy the regular expression. A meta-property is used to restrict individual resources on the path and has the structure of a triple-pattern but with a different semantics. Both languages are only briefly introduced, with a focus on their prototype implementations. *SPARQLeR* suffers from a lack formally-defined syntax and semantics, thus one has to guess how certain expressions are evaluated and what is expressible. Further, due the use of regular expressions (i.e. RPQ) they exhibit just limited support for querying RDF data along with is schema information. Moreover, both languages support only simple paths, which means that a resources is allowed to appear at most once on a path, which further limits their applicability. Nonetheless, *SPARQLeR* and *SPARQ2L* highlighted the need to discover and output complete paths of arbitrary length in RDF graphs and proposed two novel solutions that aim at mitigating the shortcomings of SPARQL 1.0 in that respect. The *Corese Search Engine* [CDF04] can also be seen as an extension of SPARQL. It supports inference rules to capture a fragment of RDFS and enables limited navigational querying. But only paths of fixed length are supported.

More recent extensions of SPARQL include *nSPARQL* [PAG10, BPR12], *RPL* [BFL09, ZLFB10] *PSPARQL* [ABE09a], its extension *cpSPARQL* [AE14], and as last *SPARQL with Recursions* (RecSPARQL) [RSV15]. The main idea behind nSPARQL are so-called *nested regular expressions* (NREs) (or *nested path queries* (NPQ)) which combine regular expressions with nested existential tests, an inverse operator, and the notation of branching, borrowed from XPath [CD$^+$99, BBC$^+$03]. With that, structural properties can be queried that cannot be captured by paths alone, which enables the posing of many interesting and natural queries. One example are chains of users, connected by a friendship relationship, where additionally along this chain each person has to live in the same country.

The main difference to *conjunctive regular path queries* (CRPQs) [CM90] is hereby, that the length of this chain of users can be of arbitrary length, whereas the number of conjunctions in CRPQ is always fixed [BPR12]. To illustrate that, we consider how to simulate the previous query in CRPQ. This can only be done by conjunctions of successively extended paths, where for each user a variable is required in order to check for the existence of information about a country. One drawback of nSPARQL is its definition on binary relations, which again hampers meaningful path-based results as offered by SPARQLeR and SPARQ2L, for instance. On the other hand, their syntax and semantics is well defined and a detailed analysis of their evaluation complexity is provided. PSPARQL is an approach that picks up both points: more meaningful results and a well defined syntax and semantics. It extends SPARQL by regular expressions and, importantly, supports variables inside them. This way, one can retrieve *individual* intermediate elements of the path rather than being restricted to the first and last resource. With cpSPARQL, the authors proposed a further extension of their work by adding constraints to their regular expressions. RPL is another navigational query language that combines NPQs with direction modifiers and negation. The main motivation is an easy integration into RDF languages like SPARQL. Due to the common underlying concept of NPQs, the language is quite similar to nSPARQL with regard to its syntax and expressiveness. However, languages based on NPQs have some inherent restriction due to their origins from graph databases. In fact, there are crucial differences between RDF and the classical graph models which become more conspicuous with the increased complexity of schema data in RDF. In a classical graph model, it is for instance not possible to model the same resource in predicate (edge) and subject/object (node) position without losing its uniqueness [LRV13, AGP14]. This clear distinction between nodes and edge labels is inherent to most query languages where its navigational component is based on regular expressions. A general purpose recursion for SPARQL 1.1 was proposed in [RSV15], called *SPARQL with Recursions* (RecSPARQL). Its basic idea is to mimic recursions in SQL by means of a recursive operator but, analogous to SQL, it is not meant to be used for navigational patterns due its cumbersome syntax for such queries. *Extended CRPQs* (ECRPQs) [BLLW12] describe a substantially different way of how to provide more expressiveness in navigational queries. Variables are introduced which permit comparisons of paths. While this is out of reach for languages based on NPQs [BLLW12], they in turn support complex branching patterns which are inexpressible with ECRPQs. However, ECRPQs are not meant to be an RDF querying language on its own, but rather a new building block for more expressive languages and further investigation on expressiveness and complexity in evaluation.

More Expressive RDF Languages. In recent work, it has been recognized that there exist further important properties in RDF data, which cannot be captured by the languages introduced so far [LRV13, AGP14, RSV15]. A few examples have been discussed in the previous paragraph and more will be shown in Chapter 5. As a result, the *Triple Algebra with Recursion* (TriAL*) [LRV13] and *Triple Query Language* (TriQ) [AGP14] have been proposed which have shown to subsume previous RDF querying approaches such as property paths and languages using NPQs. Both languages add more expressive navigational capabilities than is supported by NPQs and can be evaluated in combined (low-degree) polynomial time. TriQ is defined as a general Datalog extension that captures SPARQL queries enriched with the OWL 2 QL profile, whereas TriAL* is a closed language that works directly with triples including recursion over *triple joins*. The descent of TriAL* from relational algebra and its inherent compositionality led us to the decision to base some of our work, presented in Chapter 5, on TriAL* rather than on TriQ. A detailed introduction to TriAL* will follow in the respective chapter.

3.3. A Feature-based Comparison of RDF Query Languages

Next, we compare the features of the most-influential query languages which have been recognized to be important for querying semi-structured data in numerous scientific studies [Abi97, MKA+02, PAL+02, HBEV04, AGH04, AG05, AG08, KBM08, Woo12]. Since most of the querying features are self-explanatory by their name, we moved their explanation to Appendix A (page 219). There, an overview of all querying features is listed, including a short explanation, their *source of origin*, and changes which were required to avoid ambiguities. The first inspiration for the set of features used for our comparison goes back to [Abi97] where path patterns have already been highlighted as a crucial feature. From the work in [AGH04, AG05] we adopt (1) seven features, (2) three out of seven graph database query languages, which proved to be the most expressive ones in [Alk08] and, (3) six RDF query languages[2]. In [HBEV04, KBM08], the authors compared RDF languages using 12 distinct features from which we considered nine to be relevant for our work. Their investigated query languages where actually the same as in [AGH04, AG05]. To these we added further languages which we have discussed in this section, namely SPARQL 1.1 (Property Paths), nSPARQL, RPL, TriQ, TriAL and compared them to both of our languages, RDFPath and TriAL-QL which will be proposed in Chapters 4 and 5, respectively. Furthermore, we added also additional features for comparison including *Querying Topology*, *Output Paths*, *Closure*, and *Sorting*. All querying features are categorized in four groups based on the sort of task they solve.

The resulting comparison is shown in Table 3.1. We can see a high diversity in supported querying features, where the differences are becoming fewer the more recent a proposed language is. Not surprisingly, those languages which we categorized as "*more expressive RDF languages*" are at the forefront in terms of performance. We can further see that, as described earlier, languages based on NPQs are denoted as having only partial support for querying the topology of a graph, and thus its schema and data at once. Only TriQ, TriAL, RecSPARQL and TriAL-QL claim to capture *all* querying constructs inherent to the *triple based* model of RDF, and provide a smooth integration of querying schema and data. Both of our languages, RDFPath and TriAL-QL exhibit a high coverage, where RDFPath stands out with – in the case of RDF languages – rarely-integrated analytical querying features and the ability to output paths. TriAL-QL, however, strongly benefits from TriAL as its underlying algebra which we extended by means of provenance which provides us with the partial-supporting of output paths. Overall, we can conclude that there does not exists a one-size-fits-all language. Each one has its strengths and weaknesses, which provides space for new concepts and further improvements.

For a further comparison of RDF query languages we refer to [HBEV04, AG05, FLB+06]. A detailed introduction that uses sample data and exemplary queries is provided in [BBFS05]. A comprehensive study on graph database models, which provides foundations of graph query languages used in this section, can be found in [AG08], and in a less extensive but more recent survey [Woo12]. To the best of our knowledge, this is the first study that covers such a wide range of RDF query languages ranging from classic graph database query approaches such as G+ to highly expressive SPARQL extension as proposed with RecSPARQL.

[2]Due to the lack of further information or any publications, we could not consider RxPath in our work.

Table 3.1.: A comparison of querying features of languages for RDF, where "×" indicates no support, "○" partial support, and "●" full support. The table is inspired by numerous previous studies such as [Abi97, PAL+02, HBEV04, AGH04, AG05, AG08, KBM08]

	Basic					**Navigational**													**Analytical**			**Relational**				**Others**	
Language	Namespaces	Constraints	Language	Lexical Space	Value Space	Graph Pattern	Adjacent Resources	Adjacent Predicates	Fixed-length Path	Path Variable	Inverse Path	Non-simple Path	Recursion (Regular Expression)	Constrained Regular Expression	Optional Pattern	Entailment (Reasoning)	Querying Topology	Output Paths	Degree of Resource	Distance between Resources	Shortest Path	Union	Difference	Aggregation	Sorting	Closure	Collections & Containers
RQL	●	●	×	●	●	●	○	○	●	×	×	×	○	×	○	●	×	×	○	×	×	●	●	●	×	×	●
SeRQL	●	●	●	●	●	●	○	×	○	×	×	×	×	×	●	●	×	×	×	×	×	●	●	×	×	●	○
RDQL	○	●	×	●	○	●	○	○	○	×	×	×	×	×	×	○	×	×	×	×	×	×	×	×	×	×	○
Triple	●	●	×	●	×	●	○	○	○	×	×	×	●	×	×	○	×	×	×	×	×	●	×	×	×	×	×
N3	●	●	●	●	×	●	○	○	○	×	×	×	●	×	×	○	×	×	×	×	×	●	×	●	×	●	●
Versa	×	●	×	●	×	●	○	○	×	×	×	×	○	×	●	×	×	○	×	×	×	●	○	●	●	×	○
Corese	●	●	●	●	●	●	●	●	●	●	×	×	×	×	●	●	×	×	×	×	×	●	●	○	×	×	○
SPARQ2L	●	●	●	●	●	●	●	●	○	●	×	●	●	○	●	×	×	●	×	×	×	●	●	×	×	○	○
SPARQLeR	●	●	●	●	●	●	●	●	○	●	●	×	●	○	●	○	×	●	×	×	×	●	●	×	×	○	○
LOREL	×	●	×	×	●	●	●	○	●	●	×	×	●	×	×	×	×	○	×	×	×	●	×	●	×	×	×
G+	×	×	×	×	○	●	●	●	●	×	●	×	●	×	×	×	×	●	●	●	×	×	×	×	×	×	×
GraphLog	×	×	×	×	○	●	●	●	●	×	●	×	●	×	×	×	×	○	●	●	×	×	×	×	×	×	×
StruQL	×	×	×	×	○	●	●	●	●	×	×	×	●	×	×	×	×	○	●	●	×	×	×	×	×	×	×
SP. 1.0	●	●	●	●	●	●	●	●	●	×	×	×	×	×	●	×	×	×	×	×	×	●	●	×	●	○	○
SP. 1.1	●	●	●	●	●	●	●	●	●	×	●	×	●	×	●	○	○	×	●	×	×	●	●	●	●	○	○
RPL	●	●	●	●	●	●	●	●	●	×	●	●	●	○	●	●	○	×	×	×	×	●	●	×	×	×	○
PSPARQL	●	●	●	●	●	●	●	●	●	●	×	●	●	×	●	●	○	×	×	×	×	●	●	○	×	○	○
cpSPARQL	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	×	×	×	●	●	○	×	○	○
nSparql	●	●	●	●	●	●	●	●	●	×	●	●	●	●	●	●	○	×	×	×	×	●	●	○	●	○	○
TriQ	×	●	●	×	×	●	●	●	●	×	●	●	●	●	●	●	●	×	×	×	×	●	●	×	×	×	○
TriAL	×	●	●	×	×	●	●	●	●	●	●	●	●	●	●	●	●	×	×	×	×	●	●	×	×	●	○
RecSPARQL	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	×	●	×	×	●	●	●	●	○	○
RDFPath	●	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	○	●	●	●	○	●	●	●	●	○	○
TriAL-QL	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	×	×	×	●	●	×	×	●	○

4. RDFPath

Contents

4.1. Motivation

The advent of rich knowledge bases comes along with a high degree of diversity in the vocabulary used, for instance, in different domains but also fundamental structural differences across various interconnected data sources. Navigational queries are among the most important query types for such semantically-annotated data as they provide valuable information about the interlinking of resources and are equipped with features to explore the underlying structure. In particular queries that allow us to (1) traverse paths of arbitrary length, (2) involve tests on data values and (3) browse the topology of a graph have been recognized as crucial for querying [PAG10, LRV13, AGP14, RSV15].

In Chapter 3, several languages with navigational capabilities that address some of these needs have been discussed. Most of the previous work in this area has its roots either in traditional graph databases [CMW87, CM90, CDLV03] or is inspired by XPath [CD$^+$99, BBC$^+$03]. As a consequence, the problem of reachability, if addressed at all, is often seen as an existential question that answers whether there exists a connection between two resources or not. That is also the case for nSPARQL [PAG10, BPR12] and PSPARQL [ABE09a, AE14], two languages which are based on *nested path queries* (NPQ)) (cf. Section 3.1 page 44) thus combine regular expressions with nested existential tests. NPQs are also used as navigational primitives for RDFPath, but in contrast to nSPARQL and PSPARQL, which are extensions of SPARQL, RDFPath is a self-contained language enabling a much more intuitive syntax for path-shaped queries. Furthermore, its fully path-based semantics outputs complete paths, i.e. all resources traversed along a path. There has been only limited work done on languages that offer expressive navigational querying capabilities for RDF and in addition a granular view of what the actual path between two resources looks like. SPARQLeR [KJ07] and SPARQ2L [AMS07] support the extraction of complete paths but offer only basic navigational querying features based on *regular path queries* (RPQs). Additionally, they either lack a formally-defined syntax and semantics [KJ07] or have an inconvenient syntax, where paths need to be described by filter constraints [AMS07].

RDFPath is an intuitive yet expressive navigational query language for RDF designed to fit this gap. Equipped with functionalities to traverse paths of arbitrary length and the ability to apply tests on data values which are not part of the main path, it already captures some aforementioned properties that have become highly relevant for querying RDF. However, one of the main goals behind RDFPath is to provide more meaningful results that include *complete* paths between two connected resources. Such paths describe the detailed trace, i.e. they refer to each traversed resource along the path. To do so, RDFPath uses expressions that work natively over paths, rather than transforming the data into, e.g., pairs of nodes. This way, not just each individual intermediate step in RDFPath gets a set of paths as input and produces a set of paths as output, but also the final result is supposed to be a set of paths. However, we understand RDFPath as a supplementary language that should be integrable into existing workflows and environments which use diverse languages and tools developed for the Semantic Web. For that, we need to ensure the compatibility with RDF, its standardized data model. We will therefore introduce in RDFPath a mapping from resulting paths back to RDF triples which preserves the information about traversed resources, while making them available for post processing with another RDF management system.

Another important feature captured by RDFPath is some sort of reasoning, where we allow the querying of RDF data along with its ontology and schema. This is of particular interest in cases where we want to utilize the topology of a graph [PAG10, RSV15] or to track provenance [MC13]. In contrast to graph database models, an edge label in RDF (property) may also serve as a source or destination of another edge. RDFPath supports the traversal of many variants inherent to the triple-based model including *property-subject* relationships and allow the combination of RDF data with its schema in a smooth way.

Furthermore, there is limited support for aggregations in RDFPath. Even through a rather restricted usage, they provide meaningful numbers that can be used to reason about computed paths in order to provide a better understanding of the results. To the best of our knowledge, RDFPath is the first RDF querying language that combines all hitherto mentioned features in one unified concept and provides an intuitive yet expressive language.

Most of the results from this Chapter were published in [PSHL11] and in [PSHL12]. Some of the results, such as the semantics of RDFPath, that were missing in these publications are presented here for the first time. Overall, we can summarize the contributions of this chapter as follows:

1. In Section 4.2, after some preliminaries on RDF we propose RDFP, a flexible yet simple extension of RDF to represent paths in RDF graphs. RDFP supports so-called *path statements* which are composed of traversed resources. This representation of paths forms the basis for RDFPath.
2. RDFPath, along with its most important expressions, is introduced in Section 4.3. Hereby, we will identify important patterns covered by RDFPath expressions and illustrate their usage in a running example. To this end, a complete RDFPath syntax is provided which contains not only navigational expressions, but also constraints and result modifiers.
3. The formal semantics of RDFPath, where RDFP is used as a basis, is presented in Section 4.4. A few formalized notations, algorithms for newly introduced operators, and a discussion on the properties of RDFPath complement this section.
4. We close up this Chapter with Section 4.6, where we discuss a mapping from RDFP results into RDF triples followed by Section 4.7 which points out some limitations of RDFPath and motivates the follow-up work with TriAL-QL.

4.2. A Data Model for Paths

In this section we will first recall the definition of RDF as introduced in [MMM14] and discuss some shortcomings that hamper important querying features when using RDF to represent intermediate results. Afterwards, we will introduce our new data model, RDFP, as an extension of RDF to overcome these issues.

4.2.1. Preliminaries

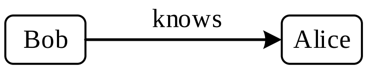
The RDF data model expresses information about arbitrary resources by the use of so-called *RDF statements* with the triple-based structure:

```
<subject> <predicate> <object>
```

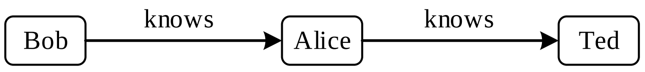
This way, every single piece of information about a resource is modeled as a triple (*s,p,o*), that can be interpreted as "a subject s has the predicate p with the object (value) o". More complex relationships are composed by sets of triples and are called *RDF Graphs.* More formally, assume $\mathcal{I}, \mathcal{B}, \mathcal{L}$ to be pairwise disjoint, countably infinite sets of international resource identifiers (IRIs), blank nodes and literals, respectively. Then, an RDF triple t is defined as:

$$t = (s, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$$

A set $\mathcal{G}$ of triples is called RDF Graph. From a graph database perspective, a triple (*Bob, knows, Alice*) can be also depicted as two nodes *Bob* and *Alice* connected by the edge *knows*:



An RDF graph with multiple triples, e.g., $\{(Bob,\ knows,\ Alice), (Alice,\ knows,\ Ted)\}$ that share a common resource (in this case *Alice*) can be illustrated as a connected graph:



Shortcomings of RDF Serializations. The data model of RDF uses statements with a ternary structure to describe the relation between two resources. More complex relationships that involve multiple resources have to be modeled by additional statements. With Notation 3 (N3) [BLC11], there exists a W3C team submission, for a superset of RDF with support for more complex relationships. Most notably for us are two shorthands. Firstly, there exists a shorthand for subjects with multiple properties, e.g., several statements that refer to the same subject. As an example, we can consider the RDF graph consisting of the two triples (*Alice, knows, Bob*) and (*Alice, age, 21*), which would be abbreviated by using a semicolon. Secondly, there exists a shorthand for subjects and properties having multiple objects (also known as multi-value properties). An exemplary RDF graph for this case is {(*Alice, knows, Bob*), (*Alice, knows, Ted*)}, where a comma is used to merge both triples. The N3 document containing both shortened statements will be then:

```
<Alice> <knows> <Bob>, <Ted>;
        <age> 21.
```

Unfortunately, there is no support for path-based structures modeled by, e.g., {(*Bob, knows, Alice*), (*Alice, knows, Ted*)}. This is also the case for all other common serialization formats for RDF including Turtle, N-Triples and RDF/XML (see [MMM14] for a comparison). Consequently, it is not possible to represent paths that have a length of more than *one* in a compact way, using, e.g. just one single statement. Given that RDF is intended to be just an abstract data model one may argue that this does not play a role. After all, RDF is (1) usually transformed into another representation after loading and (2) the information can be simply derived by querying the data. However, in this chapter we will show that introducing a lightweight concept of paths – directly into RDF – forms a basis that facilitates much more meaningful results of navigational RDF query languages, while at the same time simplifying querying.

4.2.2. Representing Paths

RDFp is our extension of RDF that introduces the concept of paths in order to tackle the aforementioned shortcomings of RDF and current navigational query languages. Conceptually, it can also be seen as a new form of serialization format for RDF that includes additional features. The basic idea behind RDFp is to define an RDF statement that summarizes multiple ternary statements which model a path-based structure into one composed statement. The general structure of such a composed statement, that expresses the (path-based) relationships between n resources is a n-ary tuple as follows:

```
<resource_1> <resource_2> <resource_3> ... <resource_n>
```

Definition 4.1 (RDFp Path). Let $\mathcal{I}, \mathcal{B}, \mathcal{L}$ be pairwise disjoint, countably infinite sets of international resource identifiers (IRIs), blank nodes and literals, respectively. An RDFp *path statement* (or short *path*) of length n is an n-ary tuple p formally defined as:

$$p = (r_1, r_2, r_3, ..., r_n) \in \underbrace{(\mathcal{I} \cup \mathcal{B}) \times (\mathcal{I} \cup \mathcal{B})...(\mathcal{I} \cup \mathcal{B})}_{r_1,\ ...,\ r_{n-1}} \times \underbrace{(\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})}_{r_n}$$

□

Definition 4.2 (RDFp graph). A set of RDFp paths p is then called an RDFp graph $\mathcal{P}$. □

Definition 4.3 (RDFp entailment). Each RDF triple t of an RDF graph $\mathcal{G}$, is also a valid path p of an RDFp graph $\mathcal{P}$. Hence, it is true that $\mathcal{G} \subseteq \mathcal{P}$ and all RDF graphs are also RDFp graphs. □

Example 4.1. Consider two triples $(Bob,\ knows,\ Alice)$ and $(Alice,\ knows,\ Ted)$ that can be also interpreted as two RDFp paths with length three. Following Definition 4.1, we can combine both triples and represent them as a new RDFp path $(Bob,\ knows,\ Alice,\ knows,\ Ted)$ of length five. □

Example 4.2. Next, we are considering a small RDF graph $\mathcal{G}$ containing also topological information about the predicate `knows`.

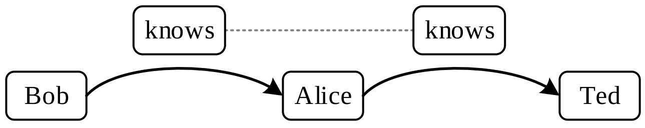
$$\{(Bob,\ knows,\ Alice),\ (Alice,\ knows,\ Ted),\ (Ted,\ age,\ 32),\\ (knows,\ type,\ friendship)\}$$

An RDFp graph $\mathcal{P}$ with exemplary compositions from $\mathcal{G}$ might contain the following RDFp paths:

$$\{(Bob,\ knows,\ Alice,\ knows,\ Ted),\\ (Bob,\ knows,\ Alice,\ knows,\ Ted,\ age,\ 27),\\ (Bob,\ knows,\ type,\ friendship),\\ (Alice,\ knows,\ type,\ friendship),\\ (Bob,\ knows,\ Alice,\ knows,\ type,\ friendship)\}$$

Note that, since $\mathcal{G} \subseteq \mathcal{P}$ all original triples from $\mathcal{G}$ are also interpreted as paths and could be included in $\mathcal{P}$. □

RDFp paths provide a lightweight and rather general concept for *arbitrary* paths in RDF graphs that also covers highly-relevant property-subject relationships. As shown in Example 4.2, properties may also appear as a source or destination (subject and object, respectively) of another edge. Those relations cannot be expressed using a standard graph model as intermediate representation. However with RDFp, as we have seen in this section, there are no such restrictions and an IRI may appear at any place and in any order within a RDFp statement, as is the case for RDF. Due to this generality we drop, for its graphical interpretation, the commonly used approach of illustrating predicates as labeled edges. Instead we follow a notation from [PAG10, LRV13] and depict each RDF term as a node. In case of our previous Example 4.1, we illustrate the RDFp path (*Bob, knows, Alice, knows, Ted*) as:



4.3. RDFPath Syntax

RDFPath is a fully path-based language using RDFP as an underlying data model. Accordingly, all RDFPath expressions are defined on paths and not pairs of nodes, which is a crucial difference to many previous approaches. A schematic overview of the input and output specification is illustrated in Figure 4.1. We can see that there is no need to transform the graph into another representation. The input for an RDFPath query is RDFP (or RDF), and both intermediate results and the final output are represented as RDFP. More precisely, each RDFPath expressions takes paths as input and returns paths as output, thus it is a *closed* language with respect to the RDFP data model. This allows expressions to be easily composed and as a result it simplifies query writing. Furthermore, since we understand RDFPath as a complementary to existing RDF query languages, e.g. SPARQL, we need to ensure the compatibility of RDFPath results with other RDF management systems. We will therefore introduce (near the end of the chapter) a mapping from RDFP to the triple-based RDF model.

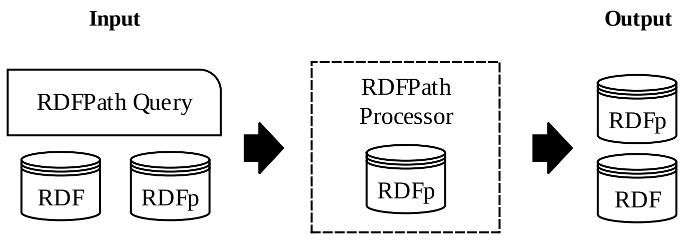


Figure 4.1.: Input/Output specification for RDFPath queries

The querying expressions that will be introduced next combine various features from graph databases languages like RPQs [CMW87, CM90, CDLV03] and NREs [PAG10, BPR12] and most notably from the XML language XPath [CD⁺99, BBC⁺03]. Further elements are derived from relational languages and adopted to work on paths. All in all, we will have a simple yet expressive language for querying RDF Graphs with an underlying path-based data model that provides us many querying capabilities. Figure 4.2 shows an example through which we will illustrate the usage of RDFPath expressions in this section. It describes the relationship between four persons including some additional demographic information. Next, we will start introducing the main syntactical expressions of RDFPath and demonstrate their usage on exemplary queries.

Namespaces. When using the full RDF triple notation, each IRI has to be written out completely, which results in very long queries that are hard to write and read. The RDF Primer [MMM14] proposes, analogously to XML, a shorthand that simplifies both serialization of RDF and query writing by introducing so-called *namespaces*. A complete IRI can be shortened by substituting its namespace with an abbreviation (prefix). For example, if we want to abbreviate the

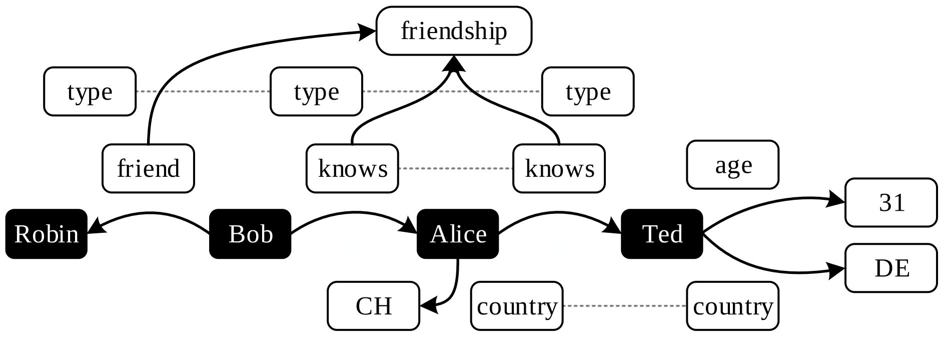


Figure 4.2.: RDF example describing friendship relations between four people, where same resources are connected by a dotted line

resource `http://dbpedia.org/resource/Marco_Polo` we can define a prefix `dbp` for the namespace `http://dbpedia.org/resource/` and write `dbp:Marco_Polo` as a shorthand. We adopt this concept for RDFPath and allow the defining of arbitrary namespaces, analogous to SPARQL [PS08], by preceding a query as follows:

```
@prefix rdf  : <http://www.w3.org/1999/02/22-rdf-syntax-ns\#>;
@prefix dbp  : <http://dbpedia.org/resource/>;
@prefix foaf : <http://xmlns.com/foaf/0.1/>;
```

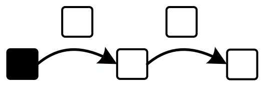
Context. An RDFPath query starts with a specification of the initial *context* [CD+99]. This will be the first node of a path or, in other words, the starting node. As an example, to indicate that all of our paths should start with, e.g. `Bob`, we specify this resource by its IRI at the beginning of a query. There are also queries where we do not want to make any restrictions on the context, thus allowing paths to start at all resources in the graph. This can be expressed by using a so-called *wildcard selection* denoted by the symbol "$*$" instead of a fixed IRI. A middle way between both options is to define the context as a set $\mathcal{R}$ of resources $\{r_1, ..., r_n\}$ enabling multiple starting nodes. After that, we specify the path we want to follow which is separated by the symbol "`:`" from the context specification. The basic notation of a context ctx is then defined as $ctx \in \{i, *, \mathcal{I}'\}$ where $i \in \mathcal{I}$, $\mathcal{I}' \subseteq \mathcal{I}$ and $\mathcal{I}$ is the countably infinite set of international resource identifiers (IRIs). Assuming an arbitrary RDFPath pattern "$/rp$", a query with context definition is written as follows:

$$ctx\ :/rp$$

The followed exemplary queries illustrate the usage of all three cases whereby the friendship is expressed by the predicate `knows`. Query (1) asks for friends of Bob, (2) for friends of any resources (i.e., everything) and (3) for friends of Bob or Alice:

```
          Bob :/knows
            * :/knows
  {Bob, Alice} :/knows
```

Traversing steps. The most crucial expression in RDFPath that forms also the main building block for each query is the traversing operator which allows us to navigate through the graph structure. Analogous to Regular Path Queries (RPQs) [CMW87] and XPath [CD+99], its shape can be illustrated as follows using a filled square to indicate the initial context:



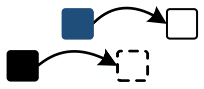
As shown in the example queries above, the property (edge) we want to follow first, is specified directly after the context. We refer to this as a *basic traversing step*. More complex patterns are possible by combining multiple traversing steps, each separated by the symbol "/" and indicating a further edge to follow next. Again, we are allowed to use wild card selections (denoted by "$*$") which simply select all properties (outgoing edges) or define sets $\mathcal{R}$ composed of multiple resources $r_1, ..., r_n$ analogous to the context definition. If we consider for instance our running example, we are now able to ask also for the Friends-of-a-Friend. To do so, we need to traverse the graph structure by following the *knows* property successively (1). We can also ask for any resources that are reachable from Bob within three traversing steps (2) or apply sets (3), similar to the context definition. All three query types are exemplary illustrated as follows:

```
Bob :/knows /knows
Bob :/* /* /*
Bob :/{knows, friend} /{knows, friend} /knows
```

More formally, assume a single traversing step t to be defined as $t \in \{i, *, \mathcal{I}'\}$ where $i \in \mathcal{I}$, $\mathcal{I}' \subseteq \mathcal{I}$ and $\mathcal{I}$ is the countably infinite set of international resource identifiers (IRIs). An RDFPath pattern rp that combines n single traversing steps is then defined as $rp = t_1 \ / \ t_2 \ /.../ \ t_n$, with $n \geq 1$. Furthermore, as a shorthand to avoid repetitive pattern rp we can replace similar subsequent patterns $rp_i \ / \ rp_i \ / \ ...$ by $rp_i(m)$, with m indicating the number of repetitions for rp_i. This way, we can shorten the three exemplary queries above as follows:

```
Bob :/knows(2)
Bob :/*(3)
Bob :/{knows, friend}(2) /knows
```

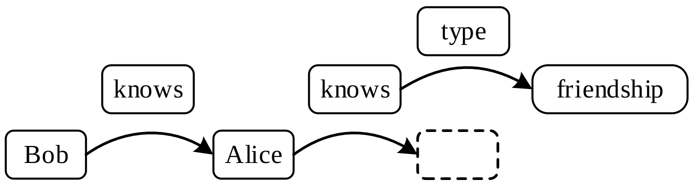
So far, a traversing step provides capabilities to browse RDF as a graph-like structure with a fixed distinction between edges and nodes. However, as discussed earlier an edge label in RDF (predicate) does not come from a finite alphabet and may also appear as a source or destination (subject and object, respectively) of another edge. Consequently, to capture also this pattern, we need to introduce a further traversing step that breaks the distinction of edges and nodes in order to also cover such kinds of navigational movements. Basically, we need a traversing step which leads not to the object, but to the property itself, thus allowing the initialization of further traversing steps which start from the predicate. The corresponding shape can be illustrated as follows:



We express this new traversing step by the symbol "\". An exemplary query that illustrates its usage by querying the topology of a graph is the following:

```
Bob :/knows \knows /type
```

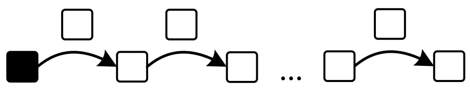
Considering our running example (Figure 4.2), the upper query would match the following path:



For the remaining notations in this chapter, we will need to distinguish multiple times between both traversing steps. To sum them up, for two RDFPath expressions rp_i, rp_j we write

- rp_i / rp_j for querying the graph graph-structure of RDF, by means of *object-subject* relationships in RDF,
- $rp_i \setminus rp_j$ for querying the topology of a graph (thus its schema or ontology) by means of *predicate-subject* relationships in RDF.

Recursions. Reachability problems, like the question whether a path exists between two given resources, has been widely recognized as an important feature for querying RDF Graphs [PAG10, LRV13, AGP14, RSV15]. The shape of such a query exhibits the following pattern:



Essentially, we adopt the concept of recursion as suggested in [PAG10] but in contrast, we are working on paths, thus arbitrary n-ary relations (cf. Section 4.2) rather than binary relations. The basic notation of a recursive RDFPath pattern rp, which in turn can be composed of multiple traversing steps, supports the following expressions:

$$rp(*) \mid rp(+) \mid rp(min, max) \mid rp(min, *) \mid rp(*, max)$$

where,

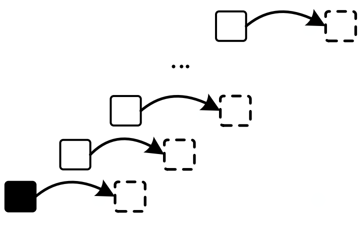
- * represents *zero* or *more* matches,
- + represents *one* or *more* matches,
- ***min*, *max*** represent at least *min* and at most *max* matches.

We next consider again our running example from Fig. 4.2 and demonstrate the usage of recursion on the following three queries:

```
1 Bob :/knows(*)
2 Bob :/knows(*,3)
```

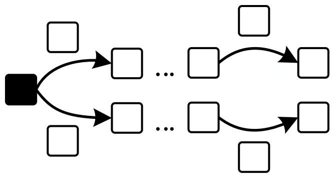
In (1) we determine all those users, that are reachable by following the *knows* predicate an arbitrary number of times and start at Bob. To restrict the search to a certain depth m, and thus compute just the so-called m-hop neighborhood of Bob we write "$(*, m)$" instead of "$*$" as shown in (2)

Furthermore, if we use the second traversing operator "\", which refers not to the object but to the property itself, along with recursion, we are now able to capture the followed connectivity pattern:



These patterns are of particular interest in cases where we want to query, e.g., the topology of a graph [PAG10, RSV15] or to track provenance [MC13]. Usually, such information is described by an underlying ontology which encoded knowledge by means of property-subject relationships, e.g. subclass relations.

Branching. Due to the high diversity in RDF vocabulary it is often hard to write queries that match exactly what we want. Even within one graph we might need to consider different patterns to capture what we are interested in. This becomes even more important if we want to e.g., query merged RDF graphs from different domains. Introducing the concept of alternative patterns as known from XPath [CD+99, BBC+03] enables us to capture the diversity of RDF and provide much more flexibility in query writing. In RDFPath, we refer to this concept by the term *branching.* We can illustrate its general shape as follows, where two alternative branches are invoked:



Syntactically, two alternative branches rp_1 and rp_2 are separated by the symbol || and grouped in parentheses. A branch might not only contain individual traversing steps but also more complex constructs including recursion and filters. In our running example, we want to express two alternative patterns that start from Bob but again share a common property that determines the age afterwards. This can be expressed by a nested-branching expression as follows:

```
Bob :/(/knows /knows || /friend /friend) /age
```

Filters. Checking for data values along a path is one of the main distinctive features between traditional path languages for traversing graph edges (e.g RPQs, CRPQs, C2RPQs [CMW87, CM90, CDLV00, CDLV03]) and more advanced approaches such as (NREs) [PAG10]. RDFPath comes along with an expressive set of filter constructs that were recognized as important querying features for RDF [PAG10, LRV13] allowing, for example, the performing of tests on the topology along a main path. Likewise in XPath [CD+99], square brackets "[ ]" are used to indicate a Boolean filter expression, where the *last* resources of each path reached by the RDFPath pattern is tested. The basic notation of a filter $filter$ that constraints the paths reached by an RDFPath pattern rp is then:

$$rp[filter]$$

where $filter$ supports the following expressions

$$rp \mid (filter \,\&\&\, filter) \mid (filter \mid\mid filter) \mid \star \text{ value} \mid \star\ f(\text{value}) \mid f() \mid (rp \star rp)$$

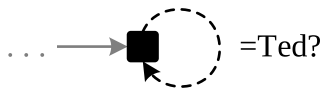
and $\star \in \{=, >, <, != \}$ [CD+99, HSP13] denote to equality and inequality tests that compare data values with a number or text and evaluate to true or false, $f()$ are *unary tests* for, e.g., testing the data type [BCF+02] of the data values, and *logical* "&&" and "||" *operators* combine multiple nested filter expression.

Next, we will categorize supported filters expressions in three categories and explain them more detailed.

1. **Basic filters:** The first type of filter corresponds to the upper description and is applied directly on the *last* resource of a path that is reached with the prior traversing step. We can compare this resource with a value or perform a unary test on it. This allows us, for instance, to test whether we have reached a certain resource on our path. An exemplary RDFPath query that illustrates the usage of a basic filter in order to check if there is a path between `Bob` and `Ted` that follows the property `knows` is specified as:

   ```
   Bob :/knows /knows[=Ted]
   ```

   The corresponding filter is expressed as `[=Ted]` and refers to the last resources of our path after following the property `knows` twice. The actual test can be illustrated as follows:

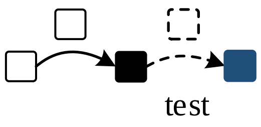
2. **Nested filters:** Nested subexpressions allow the application of filters on data values which are not part of the traversed path. To do so, traversing steps are used to define so-called sub-paths, which branch from the current path. Their syntactical rules resemble previous definitions and thus they might be nested, contain recursions, or again further filters. For a traversed sub-path, again only the *last* resources reached by the RDFPath pattern are taken into account for the filter expression. Applicable filter conditions stay the same, thus both arithmetic and unary functions are supported. More formally, assuming rp_1, rp_2, and rp_3 to be RDFPath patterns, nested filters are then constructed as follows:

   $$rp_1[filter_1]],$$
   $$rp_1[rp_2\ [filter_2]],$$
   $$rp_1[rp_2\ [rp_3\ [filter_3]]], ...$$

   where,

   - rp_1 specifies the main path in which we are interested (output path),
   - rp_2, rp_3 are sub-paths, which branches from the main path and are solely used for the evaluation of filter expressions,
   - $filter_1$ is the outer filter that restricts valid paths for the main path specified by rp_1.
   - $filter_2$ and $filter_4$ are inner filters that restricts valid paths for the sub-paths specified by rp_2 and rp_3, respectively.

The corresponding pattern shape which is traversed by nested filter expression will be illustrated with dotted lines, whereas the node from which the filter is applied is drawn in black, and the resource on the path that is used for comparison is highlighted in blue (gray):



An exemplary query (cf. Figure 4.3 (a)) that asks if the age of a user is above 21 by means of a nested filter is written as:

```
Bob :/knows /knows[/age > 21]
```

3. **Complex nested filters:** In addition, arithmetic functions can also be applied on *two* sub-paths. Such a construct is important if we want to ensure that multiple resources which are not traversed on the main path fulfill certain criteria or have a common property. However, a sub-path might not select just one single resource that can be easily compared with another one. Here, we need to handle two sets of resources rather than two single data values. We assume a test to be true if there is at least one pair of resources from both sets that fulfills the filter. Formally we write:

   $$rp_1[rp_2 \;\star\; rp_3]$$

   where,

   - rp_1 specifies the main path we are interested in,
   - rp_2 and rp_3 are sub-paths, which branch from the main path and define sets of reached resources,
   - $\star \in \{=, >, <, != \}$ is an arithmetic function that compares two sets of resources defined by sub-paths with each other. It evaluates to true if there exists at least one pair of resources that satisfies the test.

   The corresponding pattern shape can be illustrated as follows, where a test is applied on two resources reached by different sub-paths.

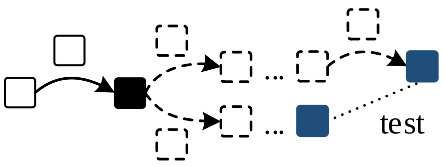


Figure 4.3 illustrates complex nested filters on our running example using the following RDFPath query:

```
Bob :/knows [/country = /knows /country]
```

The query (cf. Figure 4.3 (b)) compares the country of a user to the country of a friend. It is important to note that such sub-paths are solely used to evaluate filter expression and are not included in the resulting RDFp paths.

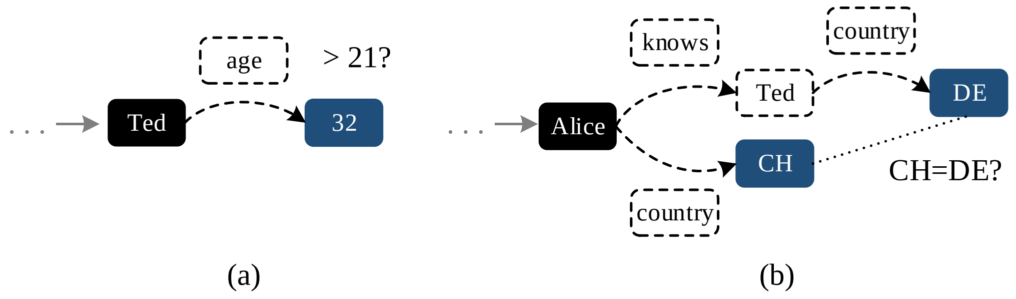


Figure 4.3.: (a) Nested filter and (b) Complex nested filter in RDFPath

Aggregate Functions. Beyond navigational capabilities and various data value tests, there is also limited support for aggregation functions in RDFPath. However, in contrast to relational algebra, their usage is strongly restricted since it breaks the closeness of RDFPath, and thus does not provide paths as output but instead single values. As a result, they are merely applicable as the last expression of an RDFPath query as a so-called *result-modifier*. Nonetheless it is an interesting feature since it provides some valuable information that can be used, for instance, to also reason about computed paths. To understand the concept of aggregations in RDFPath, we have to recall that a result of an RDFPath query is a set of RDFp path with variable length and probably even diverse data types. This semi-structured, non-columnar format prevents us from applying aggregation functions on various positions like in relational algebra. Indeed, we have to restrict aggregations to work solely on the last resources of a path. Which is, if we recall how a set of RDFp statements is composed, the only possible position for literals. In other words, when using aggregate functions, an RDFPath query works more like a path-based selector for values that we want to aggregate. From this set of selected values only those values are considered which are compatible with the specified aggregate functions. Non-compatibles values, e.g. strings when computing an average numerical value, are discarded. Syntactically, if we assume rp is the last pattern of an RDFPath query and we have an aggregate function $result()$ we write:

$rp\ .result()$

A complete list of supported functions can be found in Table 4.2. Two exemplary RDFPath queries that ask (1) for the number of Friends-of-Friend of Bob, which are reachable by following the property `knows` an arbitrary number of times and (2) for their average age are illustrated in the following:

```
Bob :/knows(*).count()
Bob :/knows(*) /age.avg()
```

The complete syntax of RDFPath is summarized in two tables. Table 4.1 describes the navigational expressions and Table 4.2 the filter conditions and result modifiers.

Table 4.1.: Syntax of RDFPath expressions

RDFPath Syntax	**Description**
$query := \langle ctx \rangle : \langle path \rangle \ . \ \langle result \rangle$	RDFPath query with optional result modifier
$ctx := iri \mid \{\mathcal{I}\} \mid *$	Context specification by IRI, set of IRIs or wildcard
$path := /rp$	Traversing step for querying the graph structure
$\mid \ \backslash rp$	Traversing step for querying the topology
$rp := iri$	IRI representing path of length *one*
$\mid \ (*)$	Wildcard matching all IRIs
$\mid \ \{\mathcal{I}\}$	Set of IRIs
$\mid \ (rp)$	Grouping of path expressions
$\mid \ {}^\wedge rp$	Inverse path expression
$\mid \ rp \ \ path$	Iterative traversing step
$\mid \ (path \mid\mid path)$	Two alternative traversing steps
$\mid \ rp \ ?(path)$	Optional traversing step with *zero* or *one* match
$\mid \ rp(*)$	Recursive traversing with *zero* or *more* matches
$\mid \ rp(+)$	Recursive traversing with *one* or *more* matches
$\mid \ rp(n, m)$	Recursive traversing with n to m matches
$\mid \ rp(n, *)$	Recursive traversing with at least n matches
$\mid \ rp(*, m)$	Recursive traversing with at most m matches
$\mid \ rp[\ filter\ ]$	Filter restrict valid paths based on node tests

Table 4.2.: Syntax of filters and result-modifiers in RDFPath

RDFPath Syntax	**Description**
$filter := path$	Check for existence of matching IRIs or whole paths
$\mid \ filter \ \&\& \ filter$	Combines different tests by using a logical operator
$\mid \ filter \mid\mid filter$	Combines different tests by using a logical operator
$\mid \ \star$ value	Equality and inequality tests with $\star \in \{=, >, <, ! =\}$
$\mid \ \star \ f$(value)	Test using predefined functions, e.g. $prefix$(http)
$\mid \ f()$	Unary test using predefined functions, e.g. $isInt()$
$\mid \ path \ \star \ path$	Checks if there is one pair from both sets s.t. $\star$ is fulfilled
$result := count()$	Counts results
$\mid \ sum()$	Aggregates results & sums up numerical literals
$\mid \ min()$	Aggregates results & determines min. numerical literal
$\mid \ max()$	Aggregates results & determines max. numerical literal
$\mid \ avg()$	Aggregates results & computes average numerical literal
$\mid \ triple()$	Projects *first*, *second* and *last* IRI of path to a triple
$\mid \ project(list$ int)	Projects n-ary paths to m-ary paths, where $n \geq m$
$\mid \ limit$(int)	Limits the number of results by val

4.4. RDFPath Semantics

To define the semantics of RDFPath, we are taking our previously-introduced data model RDFP as a basis, on which we introduce all necessary algebraic operators. For a navigational query, the most crucial point is the traversing operator that allows us to query the graph structure. Traversing the graph structure can be intuitively computed by *composing* two individual RDFP paths (in this simple case triples) like (*Bob, knows, Alice*) and (*Alice, knows, Ted*) into a new path (*Bob, knows, Alice, knows, Ted*). In order to define this traversing operator more formally, we first need to introduce a few formalizations. After that, we will introduce our semantics of RDFPath. We conclude this section with algorithms for evaluating the most crucial operations defined in the semantics.

Definition 4.4 (Mapping Paths to Elements). Assume $p = (a_1, a_2, ..., a_n)$ to be an arbitrary RDFP path. To select individual elements of a path p we introduce the mapping functions $\pi_i(p)$, that retrieves the i-*th* element, such that $\pi_i(p) = a_i$. In addition, the first element of a path p is denoted by $first(p)$, the last one by $last(p)$ and the property of a path is assumed to be (in accordance to RDF triples) the second element, hence we can write:

$$
\begin{aligned}
\pi_1(p) &= first(p) = a_1 \\
\pi_n(p) &= last(p) \quad = a_n \\
\pi_2(p) &= prop(p) \ \ = a_2 \\
\pi_i(p) &= a_i
\end{aligned}
$$

□

Definition 4.5 (Mapping Paths to Values). Let $\mathcal{P}$ be an RDFP Graph and $p \in \mathcal{P}$. In order to determine the length of a path p, we define the function $length(p)$, such that for a path $p = (a_1, a_2, ..., a_n)$, $length(p) = n$. Furthermore, using this definition we can now compute the minimal and maximal length of all paths $p \in \mathcal{P}$ and obtain the number of paths in $\mathcal{P}$ as follows[1]:

$$
\begin{aligned}
min(\mathcal{P}) &= min\{\ length(p_i) : p_i \in \mathcal{P}\ \} \\
max(\mathcal{P}) &= max\{\ length(p_i) : p_i \in \mathcal{P}\ \} \\
count(\mathcal{P}) &= |\mathcal{P}|
\end{aligned}
$$

□

[1]Note that the notion of $min()$ and $max()$ in Definition 4.5 is *not* related to the aggregation functions which are applied as a result-modifier as shown in Table 4.2. Instead, they introduce some basic formalisms required for subsequent definitions.

Example 4.3. Consider an RDFp Graph $\mathcal{P} = \{p_1, p_2, p_3\}$ with

$$\begin{gathered} p_1 : (Bob,\ knows,\ Alice), \\ p_2 : (Bob,\ knows,\ Alice,\ knows,\ Ted), \\ p_3 : (Bob,\ knows,\ Alice,\ knows,\ Ted,\ age,\ 31) \end{gathered}$$

Applying our newly introduced functions on $\mathcal{P}$ and some exemplary paths $p \in \mathcal{P}$ we obtain the followed elements and values:

$$\begin{aligned} first(p_1) &= Bob & count(\mathcal{P}) &= 3 & length(p_2) &= 5 \\ last(p_3) &= 31 & min(\mathcal{P}) &= 3 \\ \pi_6(p_3) &= age & max(\mathcal{P}) &= 7 \end{aligned}$$

□

Definition 4.6 (Projection). Analogous to relational algebra where projection is used to remove columns, we introduce projection in RDFPath to eliminate, e.g. the i-*th* element from all paths. Given $\mathcal{P}$ to be an RDFp graph, the projection is applied on each path $p \in \mathcal{P}$ and formally defined as:

$$\pi_{i1,i2,\dots,ij}(\mathcal{P}) = \{(\pi_{i1}(p), \pi_{i2}(p), ..., \pi_{ij}(p)) \mid p \in \mathcal{P}\}$$

Please note, that for $\pi_m(p)$ with $m > length(p)$ where the m-*th* element certainly does not exist, no element is added to the path by this operation.

□

Definition 4.7 (Sub-paths). Analogous to projection, we define a function that shortens a path $p \in \mathcal{P}$ by a fixed number of elements, which are removed from the end. We denoted it by $\delta_k(p)$ for individual paths and by $\delta_k(\mathcal{P})$ if we apply this function on a set of paths respectively if we remove the last k elements, formally defined as:

$$\begin{aligned} \delta_k(p) &= (\pi_1(p), \pi_2(p), ..., \pi_{length(p)-k}(p)) \\ \delta_k(\mathcal{P}) &= \{\delta_k(p) \mid p \in \mathcal{P}\} \end{aligned}$$

In the case of a path $p_i \in \mathcal{P}$ with $length(p_i) \leq k$, the path p_i is discarded after applying $\delta_k(\mathcal{P})$.

□

Definition 4.8 (Path Composition). Let $p = (a_1, a_2, ..., a_n)$ and $q = (b_1, b_2, ..., b_m)$ be two arbitrary RDFp paths. Two RDFp paths p, q are called *compatible* iff it holds that $last(p) = first(q)$, thus a_n and b_1 refer to the same IRI. The composition $(. \circ .)$ between two compatible paths p and q is then defined as:

$$p \circ q = (a_1, a_2, ..., a_n) \circ (b_1, b_2, ..., b_m) = (a_1, a_2, ..., a_{n-1}, b_1, b_2, ..., b_m)$$

Note that a_n is dropped by the composition since it is equal to b_1 and would simply induce redundant elements. □

Example 4.4. Consider an RDFp Graph $\mathcal{P} = \{p_1, p_2, p_3\}$ with

$$\begin{gathered} p_1 : (Ted,\ country,\ DE), \\ p_2 : (Bob,\ knows,\ Alice,\ knows,\ Ted), \\ p_3 : (Bob,\ knows,\ Alice,\ knows,\ Ted,\ age,\ 31) \end{gathered}$$

Applying our newly introduced functions on $\mathcal{P}$ and some exemplary paths $p \in \mathcal{P}$ we obtain the followed RDFp paths:

$$\begin{aligned} \pi_{1,3}(\mathcal{P}) &= \{(Ted,\ DE),\ (Bob,\ Alice),\ (Bob,\ Alice)\} \\ \pi_{1,5}(\mathcal{P}) &= \{(Ted),\ (Bob,\ Ted),\ (Bob,\ Ted)\} \\ \pi_{5,6,7}(\mathcal{P}) &= \{(Ted),\ (Ted,\ age,\ 31)\} \\ \delta_2(\mathcal{P}) &= \{(Ted),\ (Bob,\ knows,\ Alice),\ (Bob,\ knows,\ Alice,\ knows,\ Ted)\} \\ \delta_1(p_2) &= (Bob,\ knows,\ Alice,\ knows) \\ \delta_3(p_1) &= \emptyset \\ p_2 \circ p_1 &= (Bob,\ knows,\ Alice,\ knows,\ Ted,\ country,\ DE) \\ p_3 \circ p_1 &= \emptyset \end{aligned}$$

□

With this notation, we can next introduce the complete formal semantics for RDFPath. For clarity of the presentation, we split the definitions into multiple parts. We start with all basic expressions, including traversing expressions for querying the graph structure of the form $/rp$. After that, we continue with all traversing expressions for querying the topology of the graph with the form $\backslash rp$. In the last step, we define the evaluation of filter expressions in RDFPath and conclude with a table which provides a mapping between syntax and semantics.

Let a, b be RDF terms, $\mathcal{A}$, $\mathcal{B}$ be sets of RDF terms $\{a_1, ..., a_n\}$ and $\{b_1, ..., b_n\}$ with $n, m \in \mathbb{N}$, respectively. The evaluation of an RDFPath expression rp over an RDFP graph $\mathcal{D}$, denoted as $[\![rp]\!]_\mathcal{D}$ is defined recursively as:

$$\begin{aligned} [\![a : /rp]\!]_\mathcal{D} &:= \ \{p \mid p \in [\![rp]\!]_\mathcal{D} \ \wedge \ first(p) = a\}, \\ [\![\mathcal{A} : /rp]\!]_\mathcal{D} &:= \ \{p \mid p \in [\![rp]\!]_\mathcal{D} \ \wedge \ first(p) \in \mathcal{A}\}, \\ [\![* : /rp]\!]_\mathcal{D} &:= \ \{p \mid p \in [\![rp]\!]_\mathcal{D}\}, \\ [\![b]\!]_\mathcal{D} &:= \ \{p \in \mathcal{D} \mid prop(p) = b\}, \\ [\![\mathcal{B}]\!]_\mathcal{D} &:= \ \{p \in \mathcal{D} \mid prop(p) \in \mathcal{B}\}, \\ [\![*]\!]_\mathcal{D} &:= \ \{p \in \mathcal{D}\}, \end{aligned}$$

$$\begin{aligned}
[\![rp_1/rp_2]\!]_{\mathcal{D}} &:= \ \{p_1 \circ p_2 \mid p_1 \in [\![rp_1]\!]_{\mathcal{D}} \ \wedge \ p_2 \in [\![rp_2]\!]_{\mathcal{D}} \ \wedge \ last(p_1) = first(p_2)\}\,, \\
[\![\hat{}b]\!]_{\mathcal{D}} &:= \ \{(o,p,s) \mid (s,p,o) \in \mathcal{D} \wedge p = b\}\,, \\
[\![(rp)]\!]_{\mathcal{D}} &:= ([\![rp]\!]_{\mathcal{D}}), \\
[\![(rp_1 \mid\mid rp_2)]\!]_{\mathcal{D}} &:= [\![rp_1]\!]_{\mathcal{D}} \ \cup \ [\![rp_2]\!]_{\mathcal{D}}, \\
[\![rp_1 \ ?(/rp_2)]\!]_{\mathcal{D}} &:= \ \{p_1 \circ p_2 \mid p_1 \in [\![rp_1]\!]_{\mathcal{D}} \ \wedge \ p_2 \in [\![rp_2]\!]_{\mathcal{D}} \ \wedge \ last(p_1) = first(p_2)\} \\
&\qquad \cup \ \{p_1 \mid p_1 \in [\![rp_1]\!]_{\mathcal{D}} \ \wedge \ \forall \ p_2 \in [\![rp_2]\!]_{\mathcal{D}} : last(p_1) \neq first(p_2)\}\,, \\
[\![(/rp)^*]\!]_{\mathcal{D}} &:= \emptyset \ \cup \ [\![rp]\!]_{\mathcal{D}} \ \cup \ [\![rp/rp]\!]_{\mathcal{D}} \ \cup \ [\![rp/rp/rp]\!]_{\mathcal{D}} \ \cup \ ..., \\
[\![(/rp)^+]\!]_{\mathcal{D}} &:= [\![rp]\!]_{\mathcal{D}} \ \cup \ [\![rp/rp]\!]_{\mathcal{D}} \ \cup \ [\![rp/rp/rp]\!]_{\mathcal{D}} \ \cup \ ..., \\
[\![(/rp)^{n,m}]\!]_{\mathcal{D}} &:= \bigcup_{i=n}^{m} [\![\underbrace{rp/.../rp}_{i}]\!]_{\mathcal{D}}, \\
[\![(/rp)^{n,*}]\!]_{\mathcal{D}} &:= \bigcup_{i=n}^{\infty} [\![\underbrace{rp/.../rp}_{i}]\!]_{\mathcal{D}}, \\
[\![(/rp)^{*,m}]\!]_{\mathcal{D}} &:= \emptyset \ \cup \ \bigcup_{i=1}^{m} [\![\underbrace{rp/.../rp}_{i}]\!]_{\mathcal{D}}, \\
[\![rp \ [filter] \ ]\!]_{\mathcal{D}} &:= \ \{p \in [\![rp]\!]_{\mathcal{D}} \wedge \mathbb{C}[\![cond]\!]^{p}_{\mathcal{D}}\}
\end{aligned}$$

The evaluation of an RDFPath $filter$ over an RDFP graph $\mathcal{D}$ for a path p is denoted as $\mathbb{C}[\![filter]\!]^{p}_{\mathcal{D}}$ and defines a boolean function which evaluates true iff the path p satisfies the test expressed in $filter$. Let $\star$ be an arithmetic function such that $\star \in \{=, >, <, != \}$ and x is an RDF term. The semantics of $\mathbb{C}[\![filter]\!]^{p}_{\mathcal{D}}$ are then defined as:

$$\begin{aligned}
\mathbb{C}[\![filter_1 \ \&\& \ filter_2]\!]^{p}_{\mathcal{D}} &:= \mathbb{C}[\![filter_1]\!]^{p}_{\mathcal{D}} \wedge \mathbb{C}[\![filter_2]\!]^{p}_{\mathcal{D}}, \\
\mathbb{C}[\![filter_1 \ \mid\mid \ filter_2]\!]^{p}_{\mathcal{D}} &:= \mathbb{C}[\![filter_1]\!]^{p}_{\mathcal{D}} \vee \mathbb{C}[\![filter_2]\!]^{p}_{\mathcal{D}}, \\
\mathbb{C}[\![/rp]\!]^{p}_{\mathcal{D}} &:= \exists q \in [\![/rp]\!]_{\mathcal{D}} : first(q) = last(p), \\
\mathbb{C}[\![/rp \ [filter] \ ]\!]^{p}_{\mathcal{D}} &:= \exists q \in [\![/rp]\!]_{\mathcal{D}} : first(q) = last(p) \wedge \mathbb{C}[\![filter]\!]^{q}_{\mathcal{D}}, \\
\mathbb{C}[\![\star \ x]\!]^{p}_{\mathcal{D}} &:= last(p) \ \star \ x, \\
\mathbb{C}[\![\star \ f(x)]\!]^{p}_{\mathcal{D}} &:= \texttt{EvalFunction(f,p,x,}\star\texttt{)}, \\
\mathbb{C}[\![f()]\!]^{p}_{\mathcal{D}} &:= \texttt{EvalFunction(f,p)}, \\
\mathbb{C}[\![rp_1 \star rp_2]\!]^{p}_{\mathcal{D}} &:= \exists p_1, p_2 : \quad p_1 \in [\![rp_1]\!]_{\mathcal{D}} \ \wedge \ p_2 \in [\![rp_2]\!]_{\mathcal{D}} \\
&\qquad \wedge \, last(p) = first(p_1) \wedge last(p) = first(p_2) \\
&\qquad \wedge \, last(p_1) \ \star \ last(p_2)
\end{aligned}$$

The RDFPath expressions for querying the topology of the graph ($\backslash rp$), denoted by $[\![rp]\!]_{\mathcal{D}}$, are defined analogous to the aforementioned semantics. Unchanged expressions are omitted in the following.

$$\begin{aligned}
[\![a : \backslash rp]\!]_{\mathcal{D}} &:= \{p \mid p \in [\![rp]\!]_{\mathcal{D}} \ \wedge \ first(p) = a\}\,, \\
[\![\mathcal{A} : \backslash rp]\!]_{\mathcal{D}} &:= \{p \mid p \in [\![rp]\!]_{\mathcal{D}} \ \wedge \ first(p) \in \mathcal{A}\}\,, \\
[\![* : \backslash rp]\!]_{\mathcal{D}} &:= \{p \mid p \in [\![rp]\!]_{\mathcal{D}}\}\,, \\
[\![rp_1 \ \backslash rp_2]\!]_{\mathcal{D}} &:= \{\delta_1(p_1) \circ p_2 \mid p_1 \in [\![rp_1]\!]_{\mathcal{D}} \ \wedge \ p_2 \in [\![rp_2]\!]_{\mathcal{D}} \ \wedge \ last(p_1) = first(p_2)\}\,, \\
[\![rp_1 \ ?(\backslash rp_2)]\!]_{\mathcal{D}} &:= \{\delta_1(p_1) \circ p_2 \mid p_1 \in [\![rp_1]\!]_{\mathcal{D}} \ \wedge \ p_2 \in [\![rp_2]\!]_{\mathcal{D}} \ \wedge \ last(p_1) = first(p_2)\} \\
&\quad\ \cup \ \{\delta_1(p_1) \mid p_1 \in [\![rp_1]\!]_{\mathcal{D}} \ \wedge \ \forall \ p_2 \in [\![rp_2]\!]_{\mathcal{D}} : last(p_1) \neq first(p_2)\}\,, \\
[\![(\backslash rp)^*]\!]_{\mathcal{D}} &:= \emptyset \ \cup \ [\![rp]\!]_{\mathcal{D}} \ \cup \ [\![rp\backslash rp]\!]_{\mathcal{D}} \ \cup \ [\![rp\backslash rp\backslash rp]\!]_{\mathcal{D}} \ \cup \ ..., \\
[\![(\backslash rp)^+]\!]_{\mathcal{D}} &:= [\![rp]\!]_{\mathcal{D}} \ \cup \ [\![rp\backslash rp]\!]_{\mathcal{D}} \ \cup \ [\![rp\backslash rp\backslash rp]\!]_{\mathcal{D}} \ \cup \ ..., \\
[\![(\backslash rp)^{n,m}]\!]_{\mathcal{D}} &:= \bigcup_{i=n}^{m} [\![\underbrace{rp\backslash ...\backslash rp}_{i}]\!]_{\mathcal{D}}, \\
[\![(\backslash rp)^{n,*}]\!]_{\mathcal{D}} &:= \bigcup_{i=n}^{\infty} [\![\underbrace{rp\backslash ...\backslash rp}_{i}]\!]_{\mathcal{D}}, \\
[\![(\backslash rp)^{*,m}]\!]_{\mathcal{D}} &:= \emptyset \ \cup \ \bigcup_{i=1}^{m} [\![\underbrace{rp\backslash ...\backslash rp}_{i}]\!]_{\mathcal{D}}
\end{aligned}$$

To sum up, Table 4.3 provides an overview on how syntactical RDFPath expressions for querying the graph structure are defined by their algebraic notation using the previously-defined semantics. Syntactical expressions for querying the topology are mapped analogously, and are therefore omitted.

Definition 4.9 (Graph Composition). The most crucial operation in the semantics is the traversing step, computed by the expression $[\![rp_1/rp_2]\!]_{\mathcal{D}}$. An intuitive evaluation strategy that describes this operator more precisely is shown in Algorithm 4.4.1. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two RDFP graphs, i.e. sets of paths. Analogous to a join in relational algebra [Cod70, Lau05], we apply the traversing operator on sets of paths (denoted by $(. \bowtie .)$) by computing the Cartesian product between $\mathcal{P}_1$ and $\mathcal{P}_2$ while applying the composition on each individual pair of paths. □

Table 4.3.: Mapping between RDFPath syntax and semantics for expressions traversing the graph structure and applying filters. Syntactical expressions for querying the topology are mapped analogously. In case of the aggregate functions expressed by result-modifiers, they are introduced only in an example-driven manner. An algebraic notation of them is not shown in this work.

RDFPath Syntax			**RDFPath Semantics**
$query$	$:=$	$\langle ctx \rangle : \langle path \rangle \ . \ \langle result \rangle$	$\llbracket rp \rrbracket_{\mathcal{D}}$
ctx	$:=$	iri	$\llbracket a : /rp \rrbracket_{\mathcal{D}}$
	$\mid$	$\{\mathcal{I}\}$	$\llbracket \mathcal{A} : /rp \rrbracket_{\mathcal{D}}$
	$\mid$	$*$	$\llbracket * : /rp \rrbracket_{\mathcal{D}}$
rp	$:=$	iri	$\llbracket a \rrbracket_{\mathcal{D}}$
	$\mid$	$\{\mathcal{I}\}$	$\llbracket \mathcal{A} \rrbracket_{\mathcal{D}}$
	$\mid$	$(*)$	$\llbracket * \rrbracket_{\mathcal{D}}$
	$\mid$	(rp)	$\llbracket (rp) \rrbracket_{\mathcal{D}}$
	$\mid$	$^\wedge rp$	$\llbracket ^\wedge rp \rrbracket_{\mathcal{D}}$
	$\mid$	rp_1/rp_2	$\llbracket rp_1/rp_2 \rrbracket_{\mathcal{D}}$
	$\mid$	$(rp_1 \ \Vert \ rp_2)$	$\llbracket (rp_1 \ \Vert \ rp_2) \rrbracket_{\mathcal{D}}$
	$\mid$	$/rp_1 \ ?(rp_2)$	$\llbracket rp_1?(/rp_2) \rrbracket_{\mathcal{D}}$
	$\mid$	$/rp(*)$	$\llbracket (/rp)^* \rrbracket_{\mathcal{D}}$
	$\mid$	$/rp(+)$	$\llbracket (/rp)^+ \rrbracket_{\mathcal{D}}$
	$\mid$	$/rp(n, m)$	$\llbracket (/rp)^{n,m} \rrbracket_{\mathcal{D}}$
	$\mid$	$/rp(n, *)$	$\llbracket (/rp)^{n,*} \rrbracket_{\mathcal{D}}$
	$\mid$	$/rp(*, m)$	$\llbracket (/rp)^{*,m} \rrbracket_{\mathcal{D}}$
	$\mid$	$rp[\ filter\ ]$	$\llbracket rp\ [filter]\ \rrbracket_{\mathcal{D}}$
$filter$	$:=$	rp	$\mathbb{C}\llbracket rp \rrbracket^p_{\mathcal{D}}$
	$\mid$	$filter\ \&\&\ filter$	$\mathbb{C}\llbracket filter_1\ \&\&\ filter_2 \rrbracket^p_{\mathcal{D}}$
	$\mid$	$filter \ \Vert \ filter$	$\mathbb{C}\llbracket filter_1 \ \Vert \ filter_2 \rrbracket^p_{\mathcal{D}}$
	$\mid$	$\star$ x	$\mathbb{C}\llbracket \star\ x \rrbracket^p_{\mathcal{D}}$, where $\star \in \{=, >, <, !=\}$
	$\mid$	$\star\ f$(x)	$\mathbb{C}\llbracket \star\ f(x) \rrbracket^p_{\mathcal{D}}$
	$\mid$	$f()$	$\mathbb{C}\llbracket f() \rrbracket^p_{\mathcal{D}}$
	$\mid$	$path\ \star\ path$	$\mathbb{C}\llbracket rp_1 \star rp_2 \rrbracket^p_{\mathcal{D}}$

Furthermore we define, analogously to the traversing step, the optional traversing step, computed by the expression $\llbracket rp_1?/rp_2 \rrbracket_{\mathcal{D}}$. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two RDFP graphs, i.e. sets of paths. We denote the optional traversal of two sets $\mathcal{P}_1$ and $\mathcal{P}_2$ by ⟕. An algorithmic description of this operator is shown in Algorithm 4.4.2.

We captured paths with arbitrary lengths by introducing different types of recursion. The basic operator was expressed by $\llbracket (/rp)^* \rrbracket_{\mathcal{D}}$ and described a recursive traversing with zero or more matches. Assume again two sets of paths $\mathcal{P}_1$ and $\mathcal{P}_2$. An algorithmic description of this operation that describes the evaluation and its termination conditions is denoted by $(\mathcal{P}_1 \bowtie \mathcal{P}_2)^*$ and shown in Algorithm 4.4.3. We will refer to this algorithm once more in Chapter 7, where we discuss its properties and implementation for a subsequent work presented later on in Chapter 5.

Algorithm 4.4.1 : Traversing steps with sets of paths ($\bowtie$)

```
input  : Two sets of paths $\mathcal{P}_1, \mathcal{P}_2$
output : Set of paths $\mathcal{P}_{res}$ after applying $\mathcal{P}_1 \bowtie \mathcal{P}_2$
$\mathcal{P}_{res} \leftarrow \emptyset$
foreach $p_i \in \mathcal{P}_1$ do
    foreach $q_j \in \mathcal{P}_2$ do
        if $last(p_i) = first(q_j)$ then
            $\mathcal{P}_{res} \leftarrow \mathcal{P}_{res} \cup (p_i \circ q_j)$
        end
    end
end
```

Algorithm 4.4.2 : Optional traversing steps with sets of paths (⟕)

```
input  : Two sets of paths $\mathcal{P}_1, \mathcal{P}_2$
output : Set of paths $\mathcal{P}_{res}$ after applying $\mathcal{P}_1$⟕$\mathcal{P}_2$
$\mathcal{P}_{res} \leftarrow \emptyset$
foreach $p_i \in \mathcal{P}_1$ do
    foreach $q_j \in \mathcal{P}_2$ do
        if $last(p_i) = first(q_j)$ then
            $\mathcal{P}_{res} \leftarrow \mathcal{P}_{res} \cup (p_i \circ q_j)$
            $\mathcal{P}_{tmp} \leftarrow \mathcal{P}_{tmp} \cup p_i$
        end
    end
end
$\mathcal{P}_{res} \leftarrow \mathcal{P}_{res} \cup (\mathcal{P}_1 - \mathcal{P}_{tmp})$
```

Algorithm 4.4.3 : Recursive traversing steps on sets of paths $(\bowtie)^*$

```
input  : Two sets of paths $\mathcal{P}_1, \mathcal{P}_2$
output : Set of paths $\mathcal{P}_{res}$ after applying $(\mathcal{P}_1 \bowtie \mathcal{P}_2)^*$
$i \leftarrow 0$, $\Delta P^0 \leftarrow \mathcal{P}_1$
while $\Delta P^i \neq \emptyset$ do
    $i \leftarrow i + 1$
    $tmp = \Delta P^{i-1} \bowtie \mathcal{P}_2$
    $\Delta P^i = tmp - (\Delta P^0 \cup ... \cup \Delta P^{i-1})$
end
return $P_{res} = \Delta P^0 \cup ... \cup \Delta P^i$
```

4.5. Properties of RDFPath

4.5.1. Termination of Recursive Expressions

One crucial aspect we haven't discussed yet but which is indispensable in the case of recursive expressions is how to guarantee their termination. For instance, consider Algorithm 4.4.3 line 4 where we join all compatible paths without any further restrictions on them. Since triples in RDF may also describe a cyclic structure as, e.g., (*Bob*, *knows*, *Alice*), (*Alice*, *knows*, *Ted*), (*Ted*, *knows*, *Bob*) does, infinite loops with repetitive sub-paths can occur. In order to ensure that our algorithm terminates at some point, we need at first to introduce a concept that prohibits infinite loops which, in turn, can then be used to ensures that after a *finite* number of iterations no more new paths are derived, and thus that at some point $\Delta P^i \neq \emptyset$. Accordingly, we start in the section with introducing a new notation of cycles in RDF graphs. After that, we will show how this notation guarantees the termination of recursive expression.

In graph databases, a cycle is determined by the occurrence of two equal nodes on a path. Applying this concept on RDFp would mean that we prohibit resources from appearing multiple times along a path. However, a resource in RDFp that appeared once, e.g. `"knows"`, should be allowed to appear multiple times, regardless of whether it is used in the subject, predicate, or object position. A resources appearing more than once on a path then do not necessarily indicate a cycle in RDF. This makes well-known cycle definitions not applicable for RDFPath. We therefore next define a new notion of cycles for RDFPath that prevents infinite loops and comes along with many fewer restrictions than are used in graph databases.

Definition 4.10 (Cycles in RDFPath). Assume $p = (a_1, a_2, ..., a_n)$ is an RDFp path. We call then a path p *cycle-free* if for any two subsequent resources a_i, a_{i+1} and a_j, a_{j+1} in p it holds that:

$$a_i = a_j \quad \Rightarrow \quad a_{i+1} \neq a_{j+1},\ 0 < i < j \leq n$$

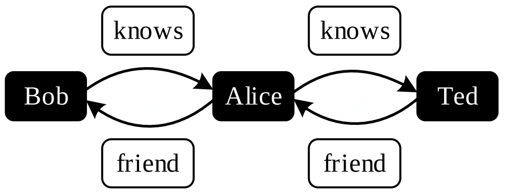
□

According to this definition, we allow on the one hand the usage of repetitive properties like `"knows"`, which are meant to be important for navigating the graph structure. On the other hand, we prohibit repetitive patterns in the form of *two subsequent resources* a_i, a_{i+1} from appearing for a second time on a path, in the same order. To keep this conditions in RDFPath, it is sufficient to check whether there exists a cycle or not each time two RDFp paths p_1, p_2 are composed (denoted by $p_1 \circ p_2$). An naive algorithm that checks if Definition 4.10 holds for the composition of two arbitrary RDFp paths p_1, p_2 is shown in Algorithm 4.5.1.

Algorithm 4.5.1 : Checks cycles in $p_1 \circ p_2$, for two RDFP paths p_1, p_2

```
input  : Two RDFP paths $p_1, p_2$
output : $True$ iff there is a cycle in $p_1 \circ p_2$, else $false$
1  $cycle \leftarrow false$,
2  $i \leftarrow 1$, $j \leftarrow 1$,
3  while ($cycle = false \ \wedge \ i < length(p_1)$) do
4      while ($cycle = false \ \wedge \ j < length(p_2)$) do
5          if ($\pi_i(p_1) = \pi_j(p_2) \ \wedge \ \pi_{i+1}(p_1) = \pi_{j+1}(p_2)$) then
6              $cycle \leftarrow true$
7          end
8          $j \leftarrow j + 1$
9      end
10     $i \leftarrow i + 1$
11 end
12 return $cycle$
```

Example 4.5. Consider the following RDF Graph R with cycles and two types of friendships and the RDFPath query that asks for all friends of `Bob` which are reachable by traversing `knows` or `friend` arbitrary times:



```
Bob :/ (knows | friend)*
```

Applying the query on R, while permitting only *cycle-free* paths, we obtain the following five RDFP paths as result.

$$\begin{gathered}
p_1: \ (Bob,\ knows,\ Alice),\\
p_2: \ (Bob,\ knows,\ Alice,\ knows,\ Ted),\\
p_3: \ (Bob,\ knows,\ Alice,\ friend,\ Bob),\\
p_4: \ (Bob,\ knows,\ Alice,\ knows,\ Ted,\ friend,\ Alice),\\
p_5: \ (Bob,\ knows,\ Alice,\ knows,\ Ted,\ friend,\ Alice,\ friend,\ Bob)
\end{gathered}$$

□

We can see, for instance, that in path p_4 the resource *Alice* is allowed to appear multiple times on the path, since there exist two distinct properties ($knows$, *Alice*) and ($friend$, *Alice*) which lead to *Alice* but do not violate Definition 4.10. For the remainder of this dissertation, we assume all RDFPath results to be *cycle-free*, thus the composition of two RDFP paths is only applicable if the resulting path does not contain any cycles in accordance to Definition 4.10.

Further, we need to note that by implying all paths are *cycle-free*, we might also exclude paths that are relevant for some use cases. One example are queries with a fixed number of traversing steps which we demonstrate next.

Example 4.6. Consider the RDF Graph R shown in the above Example 4.5 and the following query, which has a fixed number of traversing steps.

```
* :/knows /friend /knows
```

In RDFPath, we would *exclude* the following paths since they violate our *cycle-free* property.

$$\begin{aligned} p_1 &: (\textbf{Bob, knows}, \textit{Alice}, \textit{friend}, \textbf{Bob, knows}, \textit{Alice}),\\ p_2 &: (\textbf{Alice, knows}, \textit{Ted}, \textit{friend}, \textbf{Alice, knows}, \textit{Ted}),\\ p_3 &: (\textbf{Ted, friend}, \textit{Alice}, \textit{knows}, \textbf{Ted, friend}, \textit{Alice}) \end{aligned}$$

There might exist cases in which one is interested in those discarded paths as well, although they exhibit cycles. However, for these types of queries where at first, cycles do not play a role and secondly, they have a fixed number of traversing steps, one can use SPARQL Basic Graph Pattern (not Property Paths) instead which enable the retrieval of *all* matchings in such cases.

□

To continue our discussion on the termination of RDFPath queries, we next show how concept of *cycle-free* paths ensure an upper bound for the path length.

Definition 4.11 (Maximum Path Length). In accordance with Definition 4.8 where we introduced the composition between two paths, we have for two RDF triples $p = (a_1, a_2, a_3)$ and $q = (b_1, b_2, b_3)$:

$$p \circ q = (a_1, a_2, a_3) \circ (b_1, b_2, b_3) = (a_1, a_2, b_1, b_2, b_3)$$

iff it holds that $last(p) = first(q)$, thus a_3 and b_1 refer to the same IRI. Accordingly, a newly composed path obtained by $p \circ q$ contains from each of *both* triples p and q at least *two subsequent* resources. Since a path in RDFPath has to be cycle-free, and thus two subsequent resources are not allowed to appear more than once in a path, we can conclude that a triple $t \in \mathcal{G}$ can contribute also at most once to an RDFP path $p = (a_1, a_2, ..., a_n)$ obtained by an RDFPath query evaluated on $\mathcal{G}$. As a result, the maximal possible length of an RDFPath path corresponds to the number of triples in $\mathcal{G}$, denoted by $|\mathcal{G}|$. □

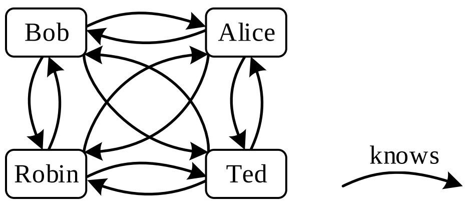
Since each traversing step in RDFPath increases the length of a path, and there exists an upper bound for the length of a path which is determined by the number of triples in the input graph, we conclude that after a finite number of iterations all possible paths are derived. This in turn guarantees the termination of recursions in RDFPath, which requires that an iterations does not derive any new paths.

4.5.2. Upper Bound for Paths in RDFPath

We have illustrated the usefulness of working with paths in multiple examples but omitted discussing the drawbacks up to now. One crucial aspect is clearly the evaluation problem, which can become rather expensive in cases of non-selective queries. However it is not only the maximum length of a path, as discussed in the previous section, which has a significant impact on the total result size, but rather the number of paths which is derivable by an RDFPath query. To investigate this aspect in more detail, we first illustrate the impact of path-based results in a small example. After that, we provide an estimation for a general upper bound of derivable results by an RDFPath query.

A commonly used example to investigate critical cases in graph databases is a so-called *clique* or known as complete graph. That is a fully-connected graph, which demonstrates in our small example a social network with four users, where each person knows everybody.

Example 4.7. Consider the following fully-connected RDF Graph R and an RDF-Path query that asks for all friends of `Bob` which are reachable by following the `knows` property arbitrary times.



```
Bob :/ knows(*)
```

Applying the query on R we obtain the following set of RDFp paths as result.

$$\begin{gathered}
\{\ (Bob,\ knows,\ Alice),\\
(Bob,\ knows,\ Ted),\\
(Bob,\ knows,\ Robin),\\
(Bob,\ knows,\ Alice,\ knows,\ Ted),\\
(Bob,\ knows,\ Alice,\ knows,\ Robin),\\
(Bob,\ knows,\ Alice,\ knows,\ Bob),\\
(Bob,\ knows,\ Robin,\ knows,\ Alice),\\
(Bob,\ knows,\ Robin,\ knows,\ Ted),
\end{gathered}$$

$$
\begin{gathered}
(Bob,\ knows,\ Robin,\ knows,\ Bob),\\
(Bob,\ knows,\ Ted,\ knows,\ Alice),\\
(Bob,\ knows,\ Ted,\ knows,\ Robin),\\
(Bob,\ knows,\ Ted,\ knows,\ Bob),\\
(Bob,\ knows,\ Alice,\ knows,\ Ted,\ knows,\ Robin),\\
(Bob,\ knows,\ Alice,\ knows,\ Ted,\ knows,\ Bob),\\
(Bob,\ knows,\ Alice,\ knows,\ Robin,\ knows,\ Ted),\\
(Bob,\ knows,\ Alice,\ knows,\ Robin,\ knows,\ Bob),\\
(Bob,\ knows,\ Ted,\ knows,\ Alice,\ knows,\ Robin),\\
(Bob,\ knows,\ Ted,\ knows,\ Alice,\ knows,\ Bob),\\
(Bob,\ knows,\ Ted,\ knows,\ Robin,\ knows,\ Alice),\\
(Bob,\ knows,\ Ted,\ knows,\ Robin,\ knows,\ Bob),\\
(Bob,\ knows,\ Robin,\ knows,\ Alice,\ knows,\ Ted),\\
(Bob,\ knows,\ Robin,\ knows,\ Alice,\ knows,\ Bob),\\
(Bob,\ knows,\ Robin,\ knows,\ Ted,\ knows,\ Alice),\\
(Bob,\ knows,\ Robin,\ knows,\ Ted,\ knows,\ Bob)\ \}
\end{gathered}
$$

□

We can clearly see, that although cycles are forbidden and we have a clique of just four resources with one edge, we obtain a sizable amount of resulting paths with a total number of 24 for this query. This blow up affects not only the final but also all intermediate results.

Next we want to estimate a general upper bound for the number of paths that can be derived by an RDFPath query evaluated against an RDF graph $\mathcal{G}$. Consider then the following RDFPath query which computes all possible paths between all resources in an RDF graph $\mathcal{G}$.

```
* :/ *(*)
```

Let the size of an RDF graph $\mathcal{G}$ be determined by the number of triples it contains, denoted by $|\mathcal{G}| = n$. We start with the trivial case which is the amount of derivable paths with length *one*. Since paths of length *one* are the triples themselves, we can note that there exists n derivable paths of length n. To estimate the number of derivable paths of length *two*, we recall Definition 4.9, where we introduced the composition of two RDFp Graphs, denoted by $(.\ \bowtie\ .)$. By computing the Cartesian product between $\mathcal{G}$ with itself, thus $\mathcal{G} \bowtie \mathcal{G}$, we obtain at most $n \cdot n = n^2$ paths. More precisely, we can assume at most $n \cdot (n-1)$ paths, since a triple can contribute according to our *cycle-free* property at most once to a path. We can carry on that way and add for *each* increase of the path length *one* further Cartesian product to the previous result, i.e. $\mathcal{G} \bowtie \mathcal{G} \bowtie \mathcal{G}$ for paths of length three and so on. Since the longest possible path in RDFPath is of length n, we need at most $n-1$ Cartesian products. Consequently, we can estimate for each path length the following number of paths:

$$\begin{array}{rl}
\textbf{Path Length} & \textbf{Number of Derivable Paths} \\
\mathbf{1}: & n \\
\mathbf{2}: & n \cdot (n-1) \\
\mathbf{3}: & n \cdot (n-1) \cdot (n-2) \\
\mathbf{4}: & n \cdot (n-1) \cdot (n-2) \cdot (n-3) \\
\vdots & \\
\boldsymbol{n-2}: & n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ ... \ (n-(n-3)) \\
\boldsymbol{n-1}: & n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ ... \ (n-(n-3)) \cdot (n-(n-2)) \\
\boldsymbol{n}: & n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ ... \ (n-(n-3)) \cdot (n-(n-2)) \cdot (n-(n-1))
\end{array}$$

Considering the longest path of length n we can state that:

$$\underbrace{\underbrace{(n)}_{n} \cdot \underbrace{(n-1)}_{n-1} \cdot \underbrace{(n-2)}_{n-2} \cdot \underbrace{(n-3)}_{n-3} \ ... \ \underbrace{(n-(n-3))}_{3} \cdot \underbrace{(n-(n-2))}_{2} \cdot \underbrace{(n-(n-1))}_{1}}_{n!}$$

We can see that there exists at most $n!$ paths of length n. Further, we can also observe that the number of derivable paths of length $n-1$ can be then determined by dividing the number of paths of length n by $n-(n-1)$, which is then:

$$\frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ ... \ (n-(n-2)) \cdot (n-(n-1))}{n-(n-1)} = \frac{n!}{n-(n-1)} \tag{4.1}$$

According to that, we can rewrite the above estimated number of paths for each path length as follows:

$$\begin{array}{rlll}
\textbf{Path Length} & \textbf{Number of Derivable Paths} & & \\
\boldsymbol{n}: & \dfrac{n!}{n-(n-0)!} & = \dfrac{n!}{0!} = n! & \\
\boldsymbol{n-1}: & \dfrac{n!}{n-(n-1)!} & = \dfrac{n!}{1!} = n! & \\
\boldsymbol{n-2}: & \dfrac{n!}{n-(n-2)!} & = \dfrac{n!}{(n-(n-1)) \cdot (n-(n-2))} & \\
\vdots & & & \\
\mathbf{3}: & \dfrac{n!}{(n-3)!} & = \dfrac{(n-3)! \cdot (n-2) \cdot (n-1) \cdot n}{(n-3)!} & = n \cdot (n-1) \cdot (n-2) \\
\mathbf{2}: & \dfrac{n!}{(n-2)!} & = \dfrac{(n-2)! \cdot (n-1) \cdot n}{(n-2)!} & = n \cdot (n-1) \\
\mathbf{1}: & \dfrac{n!}{(n-1)!} & = \dfrac{(n-1)! \cdot n}{(n-1)!} & = n
\end{array}$$

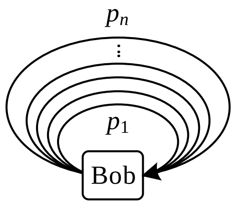
From this representation we can conclude that, for each path length i, the number of derivable paths in RDFPath is determined by

$$\frac{n!}{(n-i)!} \tag{4.2}$$

With this equation we can finally summarize all individual numbers of paths obtained for each path length resulting in:

$$\sum_{i=1}^{n} \left(\frac{n!}{(n-i)!} \right) \tag{4.3}$$

We can now conclude that the upper bound for paths that can be obtained by an RDFPath query evaluated against an RDF graph $\mathcal{G}$ with n triples can be estimated by $\sum_{i=1}^{n} \left(\frac{n!}{(n-i)!} \right)$. One exemplary RDF graph, for which that upper bound can be actually reached has the following structure:



It describes just one resource connected by n distinct predicates $p_1, ..., p_n$.

In Chapter 8, we will compare the performance of our implemented RDFPath Engine with competitive RDF management systems and also existential query languages. We will demonstrate that, in practice and with more realistic RDF graphs and queries, the amount of derived paths and consequently their impact on query evaluation, is significantly smaller than discussed in this section.

4.6. Mapping RDFp to RDF

In this chapter we introduced RDFPath, an expressive navigational query languages that uses RDFP as an underlying data model. We have seen that having paths rather than triples or pairs of nodes enable more meaningful results for, e.g. reachability queries. We achieved this by means of RDFP, which was used as the data model for the input and output of each RDFPath expression. However, if we recall our initial goal for RDFPath, which was to become a part of the ecosystem of RDF data management tools, we need our results to be compatible with other Semantic Web languages and tools. Since we cannot assume to have the support of RDFP, it is crucial to investigate how to again produce triple-based RDF graphs out of RDFP paths but without losing valuable information about the traversed path. That way, we facilitate the integration of RDFPath into existing workflows and environments, where for instance, RDFPath can be used to preprocess complex paths once in advance, and SPARQL is used afterwards to query interactively precomputed RDF data that contains paths. This problem has been also investigated for SPARQLeR in [KJ07], an SPARQL extension with support for regular path queries. SPARQLeR allows to output a matched path as a concatenated list of resources or alternatively in a triple-based representation encoded with new RDFS vocabulary. We propose as next two solutions for a mapping that ensures on the one hand the compatibility to other RDF data management systems, and preserves on the other hand the information about traversed resources.

Quadruples for Representing Paths. One common approach to enhance the syntax and semantics of RDF are so-called quadruples which extend the ternary structure of RDF by a fourth element. The general form is an 4-*ary* tuple

```
<subject> <property> <object> <context>,
```

which expresses that for a quadruple (s,p,o,c) "a subject s has the property p with the object (value) o under the context c". As this is a rather small addition on top of RDF, it is easy to implement it in existing Semantic Web tools [CBHS05]. Meanwhile, most RDF management systems support such a structure. But in most cases, the fourth element is used to handle multiple RDF graphs and is thus introduced as a unique identifier to differentiate between graphs. We can now adopt this idea for RDFPath. However, rather than adding an identifier to a triple that refers to a graph, we use the fourth element to inject the complete information about the original path in a compact representation that does not violate the 4-*ary* structure of quadruples but preserves all its information. The context c is then seen as a description for a triple *(s,p,o)* that contains information about the resources of which a path is composed. Next, we will investigate how to define such a mapping more formally.

Definition 4.12 (RDF graph with Quadruples). Analogous to an RDF graph as defined in Chapter 4.2, assume $\mathcal{I}, \mathcal{B}, \mathcal{L}$ to be pairwise disjoint, countably infinite sets of international resource identifiers (IRIs), blank nodes and literals, respectively. Then an RDF graph $\mathcal{Q}$ with quadruple (*N-Quads*) is defined as a set of RDF quadruples, such that:

$$\mathcal{Q} \subseteq (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L}) \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L}),$$

where $q \in \mathcal{Q}$ is a single quadruple (s,p,o,c). However, in order to model paths as a fourth element, rather then referring to it as a graph label, we abused the original notation by also allowing literals in the fourth position. We denote such graphs as RDF^Q graphs in the following. □

Definition 4.13 (Mapping to Quadruples). Given an RDFp graph $\mathcal{D}$, we define the RDFPath result modifier *quad()* as a mapping function that produces the following RDF^Q graph Q as result:

$$\begin{aligned} Q := \{(s,p,o,c) \mid t \in \mathcal{D} & \\ & \wedge s = first(t) \\ & \wedge p = prop(t) \\ & \wedge o = last(t) \\ & \wedge c = t.toString()\} \end{aligned}$$

The function *toString()* returns for a path $t \in \mathcal{D}$ a string concatenation of all resources traversed in t, where the symbol "/" is used as seperator. Its basic idea is similar to the list operator introduced in SPARQLeR [AMS07]. □

Example 4.8. Consider as an example the followed RDFp graph $\mathcal{D}$, that is mapped into the corresponding RDF^Q graph Q by applying the mapping function *quad()* on $\mathcal{D}$.

$$\mathcal{D} = \left\{ \begin{array}{c} (Bob,\ knows,\ Alice,\ knows,\ Ted), \\ (Bob,\ knows,\ Robin,\ friend,\ Ted), \\ (Alice,\ knows,\ Ted,\ country,\ DE), \\ (Alice,\ country,\ DE) \end{array} \right\}$$

$$\Big\downarrow\ \mathcal{D}.quad()$$

$$\mathcal{Q} = \left\{ \begin{array}{c} (Bob,\ knows,\ Ted,\ \texttt{"Bob/knows/Alice/knows/Ted"}), \\ (Bob,\ knows,\ Ted,\ \texttt{"Bob/knows/Robin/friend/Ted"}), \\ (Alice,\ knows,\ DE,\ \texttt{"Alice/knows/Ted/country/DE"}), \\ (Alice,\ country,\ DE,\ \texttt{"Alice/country/DE"}) \end{array} \right\}$$

□

One drawback of this approach is the loss of semantics in mapping to a text-based literal. On the one hand, we obtain a *human*-readable explanation for a path, but on the other, as we compose everything into a string, it is not *machine*-readable any more thus cannot be processed *automatically* in another step or system. A further problem might arise due to semantic conflicts with RDF data management systems with regard to interpreting the fourth element correctly. If such systems do not provide enough flexibility, the context element might be simply interpreted as the name for a graph rather than an explanation for its origin. To cope with this issue, we next propose a more advanced approach which preserves the information contained in a path while remaining in the triple-based RDF model.

An Ontology to Represent Paths. In order to preserve information in terms of *machine*-readability and enhance the compatibility with other RDF management systems, we need to find a representation that allows the encoding of all the information contained in RDFp paths using RDF triples. In [AMS07], the authors propose to use RDF Schema [BG14] for that. Thus, instead of mapping the path into a concatenated string as suggested previously, we can utilize an ontology to model the structure of RDFp paths. However, for this we need first to introduce a small ontology that captures the knowledge represented in RDFp paths. For an RDFp graph $\mathcal{P}$, with $p \in \mathcal{P}$ we have to model the following information:

- **All individual resources** a path p is composed of are retrieved by $\pi_1(p),\ \pi_2(p),\ \pi_3(p), ...,\ \pi_{length(p)}(p)$
- **First** element of a path, defined by $first(p)$
- **Last** element of a path, defined by $last(p)$
- **Length** of a path, defined by $length(p)$

Without implying any further restrictions on the data representation, RDF Schema is sufficient to model this information. We refer to [BG14] for a detailed introduction into RDF Schema specification. Figure 4.4 shows the complete ontology for RDFp in Notation 3 [BLC11]. Its visualization is illustrated in Figure 4.5. The ontology represents instances of RDFp paths with the newly introduced class `rdfp:Path`, which is defined as a subclass of RDF Sequence Container (`rdf:Seq`). This in turn contains an ordered bag of resources in order to model all individual resources of which a path is composed. The remaining information is captured by addition properties, namely `rdfp:first`, `rdfp:last` and `rdfp:length`.

Definition 4.14 (Mapping to RDF Schema Triples). Let $\mathcal{D}$ denote an RDFp graph. Algorithm 4.6.1 defines the decomposition of an RDFp path $t \in \mathcal{D}$ into a set of RDF triples (named `mapToRdfs()`) using the previously-introduced ontology. We define then the result modifier *rdfs()* as a mapping function that produces the following RDF graph R as result:

$$R := \{(s, p, o) \mid t \in \mathcal{D} \wedge (s, p, o) \in \texttt{mapToRdfs}(t)\}$$

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfp: <http://dbis.informatik.uni-freiburg.de/rdfpath#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

rdfp:Path rdfs:subClassOf rdf:Seq ;
    rdfs:label "RDFp path" ;
    rdfs:comment "Class of RDFp paths" .

rdfp:first a rdf:Property ;
    rdfs:comment "First resource of path" ;
    rdfs:domain rdfp:Path ;
    rdfs:range rdfs:Resource .

rdfp:last a rdf:Property ;
    rdfs:comment "Last resource of path" ;
    rdfs:domain rdfp:Path ;
    rdfs:range rdfs:Resource .

rdfp:length a rdf:Property ;
    rdfs:comment "Number of resources in path" ;
    rdfs:domain rdfp:Path ;
    rdfs:range xsd:integer .
```

Figure 4.4.: Ontology for representing RDFp paths in N3 [BLC11]

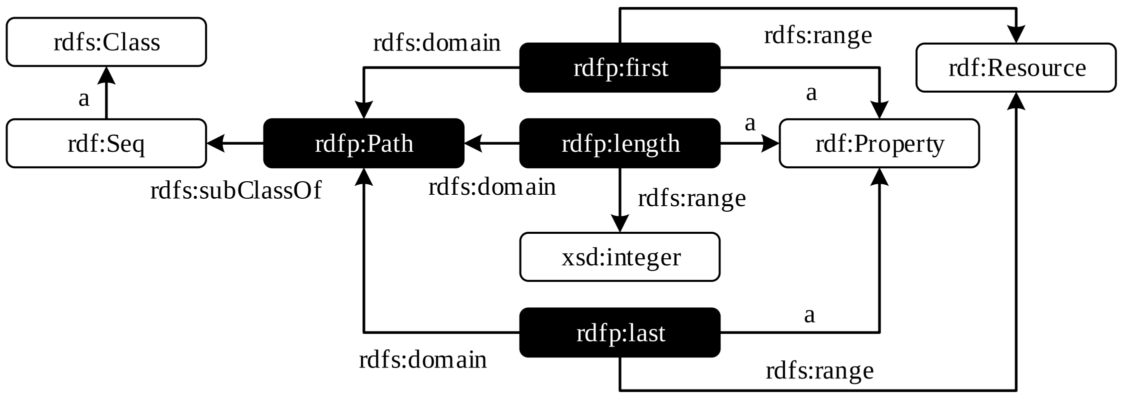


Figure 4.5.: RDF Schema Ontology for representing RDFp paths

□

One can easily see that modeling further information derived from, e.g., the actual query that was used to obtain the path, requires just minor extensions to the ontology. However, as the main intuition behind this section is to illustrate the general feasibility of mapping RDFp to RDF, we keep the ontology small and simple rather then extending it with all possible properties.

Algorithm 4.6.1 : `mapToRdfs()`: mappping a single RDFP path to RDF Schema

```
input  : An RDFp path $p = (a_1, ..., a_n)$
output : Set of RDF triples $\mathcal{R}$
1  $\mathcal{R} \leftarrow \emptyset$
2  $pathname$ = "_:path" + $hash(p)$
3  $\mathcal{R} \leftarrow (first(p),\ pathname,\ last(p))$
4  $\mathcal{R} \leftarrow \mathcal{R} \cup (pathname,$ a, rdfp:Path)
5  foreach $a_i \in p$ do
6      $\mathcal{R} \leftarrow \mathcal{R} \cup (pathname,$ "rdf:_" + $i,\ a_i)$
7  end
8  $\mathcal{R} \leftarrow \mathcal{R} \cup (pathname,$ rdfp:first, $first(p))$
9          $\cup\ (pathname,$ rdfp:last, $last(p))$
10         $\cup\ (pathname,$ rdfp:length, $length(p))$
```

Example 4.9. Consider as an example the following RDFP graph $\mathcal{D}$, that is mapped into the corresponding RDF Schema graph R by applying the mapping *rdfs()* on $\mathcal{D}$.

$$\mathcal{D} = \Big\{ (Bob,\ knows,\ Alice,\ knows,\ Ted) \Big\}$$

$$\mathcal{Q} = \left\{ \begin{array}{c} \rightsquigarrow \mathcal{D}.rdfs() \\ (Bob,\ \_{:}path1960,\ Ted), \\ (\_{:}path1960,\ a,\ rdfp{:}Path), \\ (\_{:}path1960,\ rdf{:}\_1,\ Bob), \\ (\_{:}path1960,\ rdf{:}\_2,\ knows), \\ (\_{:}path1960,\ rdf{:}\_3,\ Alice), \\ (\_{:}path1960,\ rdf{:}\_4,\ knows), \\ (\_{:}path1960,\ rdf{:}\_5,\ Ted), \\ (\_{:}path1960,\ rdfp{:}first,\ Bob), \\ (\_{:}path1960,\ rdfp{:}last,\ Ted), \\ (\_{:}path1960,\ rdfp{:}length,\ 5) \end{array} \right\}$$

□

In the resulting RDF graph of the above example, we can observe that our mapping introduces redundant information about the *first* and *last* node of a path, which can be derived from the structured path as well. We could also define a *lite* version of the mapping that does not imply such redundancy. However, having clearly named properties provides multiple advantages. First of all, having properties with a predefined meaning is valuable if we want to reason more about the results. In addition it simplifies subsequent query writing with other RDF languages. Secondly, it reduces the complexity of evaluation, for e.g., SPARQL, since we can benefit from the resulting star-shaped query patterns, which can be highly optimized as shown for SPARQL in [SPNL14].

4.7. Discussion

With RDFPath, we have introduced an expressive navigational querying language for RDF which, in contrast to many other approaches, provides access to the full path between two resources, rather than working on pairs of nodes as is often done by traditional existential semantics. We have illustrated the usefulness of such results via manifold examples, and discussed the problem of evaluation that can become rather costly for non-selective queries. This might make our approach unattractive for certain practical purposes, where we need to guarantee a low degree of complexity with regard to query evaluation. However, an expressive yet intuitive RDF query language that produces such comprehensiveness results is exactly what we were aiming for with RDFPath, despite its complexity in evaluation. By design, we did not apply any kind of widely accepted restrictions on RDFPath, such as implicit projections on fixed variables or binary relations on recursion which are meant to reduce complexity at the cost of either expressiveness or simplicity in querying.

Furthermore, if we understand querying with RDFPath more as some sort of analytical or offline ETL-like processing that is executed on large portions of the graph, we are willing to accept also long running queries. For such use cases, the mappings from RDFp to RDF is of particular importance. It produces RDF graphs that include all information derived by our navigational queries but in such a way that it can be processed by any other RDF Data Management System. Accordingly, we understand RDFPath as a complementary language in the rich ecosystem of languages and technologies developed for the Semantic Web. We will come back to the idea of using RDFPath in offline ETL-like scenarios in Chapter 6, where we will propose a MapReduce-based engine for RDFPath and investigate optimized join techniques for such kinds of workloads. Moreover, in a subsequent work shown in Chapter 7.4, we will investigate the usage of RDFPath also in interactive querying scenarios. Thereby we will demonstrate that an implementation of RDFPath on SQL-On-Hadoop solutions exhibits, despite its path-based semantics, better performance for linear-shaped queries than various competitors.

With TriAL-QL, we will discuss in the next chapter a follow-up work which has a stronger focus on a low degree of complexity in evaluation rather than obtaining comprehensive path-based results. This work will reuse the concept of RDFp as we have introduced it in the context of RDFPath, to produce results with an explanation on how the triples were derived.

5. TriAL Query Language

Contents

5.1. Motivation

Due to the underlying graphical data model of RDF, graph databases and particularly their respective graph query languages have been commonly used as a basis for the specification of new RDF querying languages [AGH04, AG05, AG08, ABE09a, PAG10]. However, although the standard model of graph databases and the triple-based model of RDF are very similar, there is a crucial difference between both. In RDF, an edge label (*predicate*) does not come from a finite alphabet like in directed edge-labeled graphs and may also appear as a source or destination (*subject* and *object*, respectively) of another edge. $\{(s, p, o), (p, s, o')\}$ for instance, is not a valid edge-labeled graph [LRV13] but a allowed in RDF. Consequently, RDF query languages based on typical graph query languages like *regular path queries* (RPQs) and *nested regular expressions* (NREs) are not capable of certain constructs and lack important querying features, e.g. reasoning over predicates within a query [Ang12, AGP14]. Accordingly, most existing RDF query languages fail to cover *all* the varieties inherent to its triple-based model, including SPARQL 1.1 and its derivatives [HSP13, RSV15]. As a result, the development of more expressive, navigational RDF languages is of general interest and has lead to a wide range of proposals ranging from application-specific concepts to expressive graph data processing languages [ABE09a, PAG10, PSHL11, PSHL12, LRV13, AGP14, PSL15a, PSL15b, HSP13, RSV15, PSL17].

With RDFPath in Chapter 4, we have proposed an expressive navigational query language, which has its roots in nested regular expressions. Nonetheless, it is equipped with functionalities which allow us to capture many aforementioned querying features. This includes especially some sort of reasoning over predicates, by providing the necessary operators to query RDF data along with its ontology. However, there exists still further properties in RDF data which need more expressive query languages that are not related to the graph database model [LRV13, AGP14].

To the best of our knowledge, there are only two RDF query languages that enable expressive navigational capabilities with full support of reasoning and can be evaluated in combined (low-degree) polynomial time, namely *Triple Query Language Lite* (TriQ-Lite) [AGP14] and *Triple Algebra with Recursion* (TriAL*) [LRV13]. TriQ-Lite is defined as a general Datalog extension that captures SPARQL queries enriched with the OWL 2 QL profile, whereas TriAL* is a closed language that works directly with triples including recursion over *triple joins*. While the steady growth of Semantic Web data, with its high degree of diversity in both structure and vocabulary, justifies particularly expressive RDF query languages, it also raises the need for solutions that scale with the data size. With that discordant goal of having an expressive query language that can be evaluated on web-scale data, we found TriAL* to have a higher applicability and better met our requirements. This is mainly due to the following reasons: TriAL* and its closeness to relational algebra captures the recent trend in scalable, interactive SQL-on-Hadoop solutions, better meeting our requirements regarding a scalable solution that works also on larger graphs. We will discuss this aspect in more detail, when we introduce our distributed implementation for TriAL* in Chapter 7. Another advantageous property of TriAL* in comparison to TriQ-Lite is its inherent compositionality, which simplifies querying since we can intuitively split complex problems into multiple nested expressions.

With TriAL* [LRV13] we benefit from an expressive algebra which subsumes many previous approaches, while adding novel features that are not expressible in most other languages based on the standard graph model. However, while TriAL* is a neat approach to query complex expressions in RDF, its algebraic notation is inappropriate for writing queries in practice. Thus, first of all we propose a new syntax called TriAL* Query Language (TriAL-QL), which is an easy to write and grasp representation of TriAL*. It preservers its compositional algebraic structure by representing each algebra operation with a SQL-like statement. This way, even complex navigational queries become easily expressible. Moreover, it allows us to implement a few handy tools, which prove to be useful in SQL and have an respective equivalent in TriAL-QL, such as storing graphs for later usage. We further introduce the Extended Triple Algebra with Recursion, or short E-TriAL*, which aims to mitigate some shortcomings of TriAL* which were hampering its usage in practical scenarios. We adopt the idea of provenance for TriAL*, which allows us to track the origin of triples and provide more meaningful results. To do so, we extend the concept of triples to quadruples where a fourth element is introduced in order to compose provenance while query writing.

Most of the results of this Chapter were published in [PSL17, PSL15a] and in [PSL15b], where it was honored with the *Best Paper Runner-Up Award* sponsored by Google[1]. Some of the results, such as provenance for TriAL* which were not presented in these publications, were discussed in conferences [CV15, SS15, SDI15] with the authors of TriAL* [LRV13] and are shown here for the first time in a written form. Overall, we can summarize the contributions of this chapter as follows:

1. In Section 5.2, we introduce the basic notation of Triple Algebra with Recursion (in short TriAL*) following the formalisms presented in [LRV13, Vrg14]. We will present a formal definition of *all* TriAL* operators and supplement them with examples.
2. Our extension of TriAL* (named in short E-TriAL*) is discussed in Section 5.3. It describes the concept of provenance, which allows us to gain some sort of insight into how triples are derived while querying.
3. The syntax of TriAL* Query Language (TriAL-QL) with its mapping to corresponding TriAL expressions is introduced in Section 5.4. We will illustrate its usage on two important patterns using exemplary queries.
4. We conclude this Chapter with Section 5.5, where we summarize the advantages but also the limitations of TriAL-QL and refer to its technical implementation in Chapter 7, where evaluation strategies and optimizations are discussed.

[1]http://dbweb.enst.fr/events/webdb2015/

5.2. Triple Algebra with Recursion

The *Triple Algebra with Recursion* (short TriAL*) [LRV13, Vrg14] is an expressive RDF query language based on relational algebra. In contrast to most other approaches it is a *closed* and hence *compositional* RDF language, where the output is a set of triples rather than graphs, mappings or RDFp paths. The result is again a typical RDF graph, as it is done in SPARQL via the construct operator [HSP13], which can be processed by any other RDF query engine. Furthermore TriAL* enables to write queries that are not expressible using languages based on the standard graph model (e.g. *Regular Path Queries* [CMW87, CM90, CDLV03] and *Nested Regular Expressions* [PAG10, BPR12]), where edge labels come from a predefined alphabet and cannot appear as subject or objects in another triple [AG08].

Data Representation. The core idea of TriAL* is to work directly on triples rather than transforming the RDF model into another representation. This way not only the input but also each intermediate and final result is always a set of triples. Accordingly, we can define the underlying data model for TriAL* similarly to an RDF graph.

Definition 5.1 (Triplestore). Assume $\mathcal{I}$ to be a pairwise disjoint, countably infinite set of international resource identifiers (IRIs), $\mathcal{B}$ the set of blank nodes, and $\mathcal{L}$ the set of literals. We refer to the underlying data model of TriAL* as a *triplestore* defined as a ternary relation

$$E \subseteq (\mathcal{I} \times \mathcal{I} \times \mathcal{I}),$$

where $t \in E$ is a triple (s, p, o). For clarity of presentation, we follow the original notation from [LRV13] and do not consider literals and blank nodes in RDF graphs, thus focusing again on *ground* RDF graphs [PAG10]. □

Triple Joins. TriAL* takes the relational algebra as its basis, but implies certain restrictions to guarantee the closure property with regard to triples. As in each navigational query language, the most crucial expression is the traversing of the graph-structure. Intuitively, if we want to traverse two triples like (*Bob, knows, Alice*) and (*Alice, knows, Ted*), we need to find a concept how to represent a composition of both. In RDFPath, we could use an RDFp path, which would result in the followed representation:

$$(Bob,\ knows,\ Alice,\ knows,\ Ted)$$

However, the most important property of TriAL* is its closeness with regard to triples. It is therefore essential to stick to ternary relations. A valid result might be for instance a new triple that describes just the transitivity between both persons and leaves out all intermediate resources, such as:

$$(Bob,\ knows,\ Ted)$$

The authors in [LRV13] proposed a so-called *triple join* for that purpose, which is meant to compose two ternary relations but keep only three positions in the results. To indicate which of six total possible resources to keep, six identifier are introduced, namely s_1, p_1, o_1 and s_2, p_2, o_2 referring to the positions of the left and right triple, respectively.

Definition 5.2 (Triple join). More formally, let E_1 and E_2 be two triplestores representing sets of triples, and $\mathcal{I}$ is a pairwise disjoint, countably infinite set of international resource identifiers (IRIs). Formally, a *triple join* (denoted by $\bowtie$) between E_1 and E_2 is defined in accordance to [LRV13] as:

$$E_1 \overset{i,j,k}{\underset{\theta,\eta}{\bowtie}} E_2,$$

where,

- $i, j, k \in \{s_1, p_1, o_1, s_2, p_2, o_2\}$ indicate the implicit projection on three fields to keep the operation closed with s_1 referring to the subject of E_1, o_2 referring to the object of E_2, etc.,
- θ represents the join conditions, i.e., comparisons of elements in $\{s_1, p_1, o_1, s_2, p_2, o_2\}$ among each other or with resources $i \in \mathcal{I}$ that evaluate to true or false,
- η is a set of conditions using arithmetic functions over elements in $\{s_1, p_1, o_1, s_2, p_2, o_2\}$ and data values that evaluate to true or false, i.e., typically filters conditions.

The result of $E_1 \overset{i,j,k}{\underset{\theta,\eta}{\bowtie}} E_2$ is then a ternary triplestore E_3, where $(r_i, r_j, r_k) \in E_3$ iff

- $r_i, r_j, r_k \in \{s_1, p_1, o_1, s_2, p_2, o_2\}$,
- $t_1 = (s_1, p_1, o_1) \in E_1$ and $t_2 = (s_2, p_2, o_2) \in E_2$,
- each condition in θ and η holds w.r.t. t_1 and t_2.

$\square$

Example 5.1. To illustrate the basic principles of a triple join, consider the following triplestore E,

$$E = \{(Bob,\ knows,\ Alice),\ (Alice,\ knows,\ Ted),\ (Ted,\ knows,\ Robin)\}$$

and the exemplary triple join expression:

$$E \overset{s_1,p_1,o_2}{\underset{o_1=s_2}{\bowtie}} E$$

Consider for instance the triples $(Bob,\ knows,\ Alice) \in E$ on the left hand side and $(Bob,\ knows,\ Ted) \in E$ on the right hand side. The implicit projection s_1, p_1, o_2

refers to the subject and predicate from the left triple and the object from the right triple. The output is then (*Bob*, *knows*, *Ted*), iff it holds that $o_1 = s_2$, thus *Alice* = *Alice* in case of the currently considered triples. The final result produced by the above triple join is the following triplestore:

$$\{(Bob,\ knows,\ Ted),\ (Alice,\ knows,\ Robin)\}$$

□

The flexibility of the triple join in TriAL allows us to compose two triples in any possible way. The most notable combinations allow us to query, for instance, the topology of a graph with predicate-subject relationships. These are not expressible in languages based on the traditional graph model where, e.g., $\{(s, p, o), (p, s, o')\}$ is not a valid graph [LRV13]. In order to illustrate the full strength of this algebra and show some examples that highlight those aspects, we need to introduce the remaining expressions from TriAL* as originally introduced in [LRV13].

Triple Algebra. Following the notation in [LRV13, Vrg14], we define next the remaining expressions of the Triple Algebra (in short TriAL), where each expression is closed, thus is producing again a triplestore as result. Let us denote in the following each relation name in a triplestore as a TriAL expression.

Definition 5.3 (Selection). Let E be a triplestore, with $t = (s, p, o) \in E$ and θ, η as defined before for the triple join. Then, the *selection* operator in TriAL $\sigma_{\theta,\eta}(E)$ is defined as:

$$\begin{aligned}\sigma_{\theta,\eta}(E) := \{(r_i, r_j, r_k) \mid\ & r_i, r_j, r_k \in \{s, p, o\} \\ & \wedge t = (s, p, o) \in E \\ & \wedge \theta, \eta \text{ holds w.r.t. } t\ \}\end{aligned}$$

□

Definition 5.4 (Union). Let assume E_1 and E_2 are two triplestores. Analogous to relational algebra, we define then the union operation in TriAL $E_1 \cup E_2$ as:

$$E_1 \cup E_2 := \ \{t \mid t \in E_1 \vee t \in E_2\}$$

□

Definition 5.5 (Difference). Let assume E_1 and E_2 are two triplestores. Analogous to relational algebra, we define the difference in TriAL $E_1 - E_2$ as:

$$E_1 - E_2 := \ \{t \mid t \in E_1 \wedge t \notin E_2\}$$

□

Definition 5.6 (Intersection). Given two triplestores E_1 and E_2, we can define the intersection $(E_1 \cap E_2)$ in TriAL in accordance to [LRV13, Vrg14] and analogous to relational algebra as follows:

$$E_1 \cap E_2 := E_1 \underset{s_1=s_2,\ p_1=p_2,\ o_1=o_2}{\overset{s_1,p_1,o_1}{\bowtie}} E_2$$

As shown, the intersection of two triplestores can be easily defined with a join between both triplestores. □

Example 5.2. Consider the following two triplestores E_1 and E_2 with:

$$\begin{aligned}
E_1 &= \{(Bob,\ knows,\ Alice),\ (Alice,\ knows,\ Ted),\ (Ted,\ knows,\ Robin)\} \\
E_2 &= \{(Bob,\ knows,\ Alice),\ (Alice,\ knows,\ Ted),\ (Ted,\ age,\ 32)\}
\end{aligned}$$

Applying previously introduced TriAL operations on E_1 and E_2 we obtain the following results:

$$\begin{aligned}
\sigma_{s=Ted}(E_1) &= \{(Ted,\ knows,\ Robin),\ (Ted,\ age,\ 32)\} \\
E_1 \cup E_2 &= \{(Bob,\ knows,\ Alice),\ (Alice,\ knows,\ Ted), \\
&\qquad (Ted,\ knows,\ Robin),\ (Ted,\ age,\ 32)\} \\
E_1 - E_2 &= \{(Ted,\ knows,\ Robin)\} \\
E_1 \cap E_2 &= \{(Bob,\ knows,\ Alice),\ (Alice,\ knows,\ Ted)\}
\end{aligned}$$

□

Recursions. Up to now, only queries of *fixed* length could be expressed with TriAL. However, in order to capture also different variates of the reachability problem we need functionalities to express paths of *arbitrary* length, which is know to not be expressible in relational algebra. As this is a crucial aspect of navigational querying languages for RDF, the authors in [LRV13] added recursion to TriAL. We refer to this as *Triple Algebra with Recursion* (TriAL*) in the following.

Recursion is added by applying the *Kleene closure* on any triple join. However, in contrast to binary relations, a join in TriAL is not necessarily associative. We need to choose three elements among $\{s_1, p_1, o_1, s_2, p_2, o_2\}$, where two elements come from a triplestore and one element will come from the other one, which means that this operation is actually asymmetric. Therefore, recursion is added by means of two operations, a $left$ and a *right Kleene closure*.

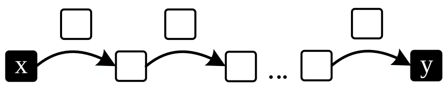
Definition 5.7 (Left and Right Kleene Closure). We denote the *right Kleene closure* as $(e \bowtie^{i,j,k}_{\theta,\eta})^*$ and the *left Kleene closure* as $(\bowtie^{i,j,k}_{\theta,\eta} e)^*$, where e is a TriAL* expression. Formally, they are defined in [LRV13] as:

$$\begin{aligned}
&(e \bowtie)^* := \emptyset\ \cup\ e\ \cup\ e \bowtie e\ \cup\ (\ e \bowtie e\ ) \bowtie e\ \cup\ ((\ e \bowtie e\ ) \bowtie e) \bowtie e\ \cup ..., \\
&(\bowtie e)^* := \emptyset\ \cup\ e\ \cup\ e \bowtie e\ \cup\ e \bowtie (\ e \bowtie e\ )\ \cup\ e \bowtie (e \bowtie (\ e \bowtie e\ ))\ \cup ...
\end{aligned}$$

□

TRIAL* captures a few interesting properties, that are not expressible by most other RDF querying languages. To illustrate them, we will consider in the following the exemplary RDF graph in Figure 5.1 that describes different kinds of relations between people and includes a small topology. This graph is adopted from a transport service example introduced in [PAG10, LRV13], but applied on the social network domain.

Example 5.3. The first reachability problem we are looking at asks for pairs of users (x, y) connected by a path which exhibits a linear shape as shown in the following:



To express this pattern, without considering any further conditions, we need solely object-subject joins. Thus, we could model this query easily as a problem in a graph database. The corresponding TRIAL* for an triplestore E is then:

$$(E \overset{s_1,\, p_2,\, o_2}{\underset{o_1=s_2}{\bowtie}})^*$$

If we assume E to be the RDF graph shown in Figure 5.1, we obtain for the above TRIAL* expression the following triples as a result:

$$\{(Robin,\ knows,\ Alice),\ (Robin,\ coworker,\ Ted),\ (Robin,\ country,\ CH),$$
$$(Robin,\ country,\ DE),\ (Robin,\ age,\ 31),$$
$$(Bob,\ coworker,\ Ted),\ (Bob,\ country,\ CH),$$
$$(Bob,\ age,\ 31),\ (Bob,\ country,\ DE),$$
$$(Alice,\ age,\ 31),\ (Alice,\ country,\ DE),$$
$$(friend,\ type,\ relationship)\}$$

Note that, in accordance to Definition 5.7, also all original triples from E are included in the result. However, for clarity of presentation, we left them out and presented only newly derived triples.

□

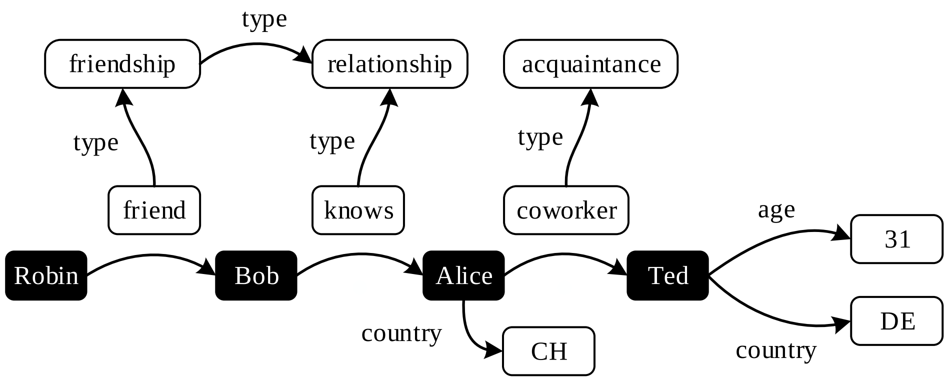
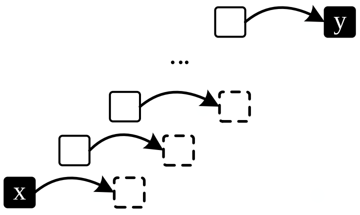


Figure 5.1.: RDF example describing a different kind of relations between people.

Example 5.4. For the second example, we investigate a query type that involves reasoning capabilities, allowing us to query RDF data along with its ontology by the use of property-subject joins. Again, we ask for pairs of nodes (x, y) connected by a path, but this time the corresponding pattern exhibits the following shape:



Due do its triple-based model, there is conceptually no difference between object-subject and property-subject joins in TriAL*. However, in graph databases, this would be not easily expressible and would imply at least a more complex graph representation. In TriAL*, we express such a shape, evaluated over a triplestore E as follows:

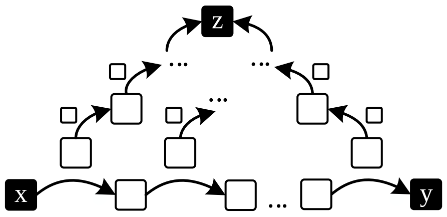
$$(E \overset{s_1,\, p_2,\, o_2}{\underset{p_1=s_2}{\bowtie}})^*$$

If we evaluate this expression on the exemplary RDF graph in Figure 5.1, we obtain the following triplestore as a result. Note that we are again showing newly-derived triples and leave out all original triples from E although they are part of the result.

$$\{(\mathit{Robin},\ \mathit{type},\ \mathit{friendship}),$$
$$(\mathit{Bob},\ \mathit{type},\ \mathit{relationship}),$$
$$(\mathit{Alice},\ \mathit{type},\ \mathit{acquaintance})\}$$

□

Example 5.5. For the last TriAL* example, we investigate a pattern that exhibits some structural properties that cannot be expressed in any query language based on *Regular Path Queries* [CMW87] or *Nested Regular Expressions* [PAG10, BPR12] as shown in [LRV13]. We ask for pairs of nodes (x, y) such that there is a connection of the same type from x to y. More specifically, if we consider our current example, we ask for pairs of users that are connected from x to y using the same type of relation z, e.g. a friendship relation. For this, we combine both previous pattern shapes into one, resulting in the following shape:



$\square$

Such queries are highly relevant, for instance if we want to query the topology of a graph [PAG10, RSV15] or to track provenance [MC13]. In TriAL*, we can capture such a structure for a triplestore E with two nested recursive expressions as follows:

$$E_1 = (E \bowtie^{s_1,\ o_2,\ o_1}_{p_1=s_2})^*$$
$$E_2 = (E_1 \bowtie^{s_1,\ p_1,\ o_2}_{o_1=s_2,\ p_1=p_2})^*$$

The inner expression E_1 computes the transitive closure of the properties from E. In other words, if we consider again our running example, we derive triples that describe connections between pairs of users by the use of *topological* information about their friendship relationships. In the first step, we derive for, e.g., *Robin* and *Bob* the triple (*Robin*, *friendship*, *Bob*) and in a subsequent round (*Robin*, *relationship*, *Bob*). All those newly derived triples are added to the initial triplestore E, such that the expression E_2 is evaluated on both the triples: from the initial triplestore and newly derived ones. For E_2, again the transitive closure based on the TriAL* expression is computed, but we consider only those pairs of users x, y that are connected with the same type of friendship relation. The evaluation of the TriAL* expressions shown above on the triplestore in Figure 5.1 produces the following results:

$$E_1 = E \ \cup \left\{ \begin{array}{c} (\mathit{Robin},\ \mathit{friendship},\ \mathit{Bob}), \\ (\mathit{Robin},\ \mathit{relationship},\ \mathit{Bob}), \\ (\mathit{Bob},\ \mathit{relationship},\ \mathit{Alice}), \\ (\mathit{Alice},\ \mathit{acquaintance},\ \mathit{Ted}) \end{array} \right\}$$
$$E_2 = E_1 \ \cup \ \{\ (\mathit{Robin},\ \mathit{relationship},\ \mathit{Alice})\ \}$$

We can see, that the pair "*Robin*, *Alice*" belongs to the final results, since we derived the triples (*Robin*, *relationship*, *Bob*) and (*Bob*, *relationship*, *Alice*). However, the pair "*Robin*, *Ted*" is not becoming part of the results, as there is no relation between *Robin* and *Ted* that uses the same type of relationship.

5.2.1. Additional Recursion Capabilities

So far, we have discussed the *right* $(e \bowtie^{i,j,k}_{\theta,\eta})^*$ and *left* $(\bowtie^{i,j,k}_{\theta,\eta}\ e)^*$ *Kleene closure*. These expressions denote the complete transitive closure, and thus are evaluated recursively until no new triples are derived. However, in practice it is not necessarily needed to compute always the *complete* transitive closure. In many cases, the majority of results is obtained within the first recursive steps and we observe a so-called *long tail*, where just a few results are added in the remaining rounds. Quite often, we have also some sort of knowledge about the data or its ontology that would enable us to formulate queries in a more efficient way by, e.g., restricting recursion up to a certain depth. Consider for instance the reachability of users in a social network. In such cases, it might be sufficient to consider only those pairs of users, which can be reached by a path with length of at most, e.g., four. Such a functionality becomes even more crucial against the background of processing RDF data at web-scale, where restrictions on the recursion depth become essential for evaluating. In RDFPath (cf. Chapter 4), we have already introduced such kind of expressions by means of different recursive traversing steps inspired by XPath. Next, we will make use of these definitions in order to extend TriAL* in that respect. We will therefore replace the *Kleene Star* with scalars to imply a limitation on the recursion depth.

Definition 5.8 (Shorthand for Iterative Triple Joins). More formally, let us assume e is a TriAL* expression and $n \geq 1$. We denote the expressions $(e \bowtie)^n$ and $(\bowtie e)^n$ as two short forms in E-TriAL* which subsume n subsequent similar triple joins over e. We define them recursively as:

$$
\begin{aligned}
(e \bowtie)^1 &:= e \bowtie e & (\bowtie e)^1 &:= e \bowtie e \\
(e \bowtie)^n &:= (e \bowtie)^{n-1} \bowtie e, \quad n \geq 2 & (\bowtie e)^n &:= e \bowtie (\bowtie e)^{n-1}, \quad n \geq 2
\end{aligned}
$$

□

Example 5.6. Consider a triplestore E describing the RDF graph illustrated in Figure 5.2. We can see here a linear friendship relation between five people. Applying our newly introduced shorthand for iterative triple joins on the triple join used in Example 5.3, we obtain the following results:

$$
\begin{aligned}
(E \overset{s_1,\,p_1,\,o_2}{\underset{o_1=s_2}{\bowtie}})^1 &= \{\ (Robin,\ knows,\ Alice),\ (Bob,\ knows,\ Ted), \\
&\qquad (Alice,\ knows\ ,Dave)\ \} \\
(E \overset{s_1,\,p_1,\,o_2}{\underset{o_1=s_2}{\bowtie}})^2 &= \{\ (Robin,\ knows,\ Ted),\ (Bob,\ knows,\ Dave)\ \} \\
(E \overset{s_1,\,p_1,\,o_2}{\underset{o_1=s_2}{\bowtie}})^3 &= \{\ (Robin,\ knows,\ Dave)\ \} \\
(E \overset{s_1,\,p_1,\,o_2}{\underset{o_1=s_2}{\bowtie}})^4 &= \{\ \emptyset\ \}
\end{aligned}
$$

□

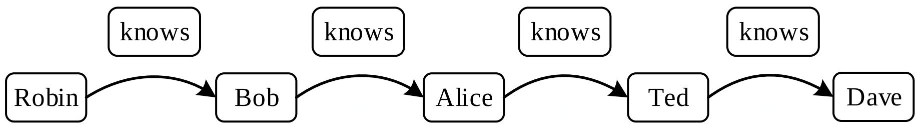


Figure 5.2.: RDF example describing friendship relations between five people.

Definition 5.9 (Limited Left and Right Kleene Closure). Assume e is a TriAL* expression and n, m $(n, m \geq 1)$ refer to the minimal and maximal recursion depth, respectively. We extend the supported recursions by the following expressions:

$$(e \bowtie)^* \mid (e \bowtie)^+ \mid (e \bowtie)^{(n,m)} \mid (e \bowtie)^{(n,*)} \mid (e \bowtie)^{(*,m)}$$
$$(\bowtie e)^* \mid (\bowtie e)^+ \mid (\bowtie e)^{(n,m)} \mid (\bowtie e)^{(n,*)} \mid (\bowtie e)^{(*,m)}$$

where,

$$\begin{aligned}
(e \bowtie)^+ &:= \emptyset \;\cup\; e \bowtie e \;\cup\; (e \bowtie e) \bowtie e \;\cup\; ((e \bowtie e) \bowtie e) \bowtie e \;\cup \ldots \\
(e \bowtie)^{(n,m)} &:= \emptyset \;\cup\; (e \bowtie)^n \;\cup\; (e \bowtie)^n \bowtie e \;\cup\; \ldots \;\cup\; (e \bowtie)^m \\
(e \bowtie)^{(n,*)} &:= \emptyset \;\cup\; (e \bowtie)^n \;\cup\; (e \bowtie)^n \bowtie e \;\cup\; ((e \bowtie)^n \bowtie e) \bowtie e \;\cup\; \ldots \\
(e \bowtie)^{(*,m)} &:= \emptyset \;\cup\; e \;\cup\; e \bowtie e \;\cup\; (e \bowtie e) \bowtie e \;\cup\; \ldots \;\cup\; (e \bowtie)^m
\end{aligned}$$

The corresponding expressions for the *right Kleene closure* follow the same principles and can be therefore defined analogous. □

Example 5.7. Consider as a triplestore E again the RDF graph shown in Figure 5.2. To illustrate the usage of the newly defined recursions, we demonstrate the evaluation for two of them on E.

$$(E \overset{s_1,\, p_1,\, o_2}{\underset{o_1 = s_2}{\bowtie}})^+ = \left\{ \begin{array}{c} (Robin,\ knows,\ Alice), \\ (Robin,\ knows,\ Ted), \\ (Robin,\ knows,\ Dave), \\ (Bob,\ knows,\ Ted), \\ (Bob,\ knows,\ Dave), \\ (Alice,\ knows,\ Dave) \end{array} \right\}$$

$$(E \overset{s_1,\, p_1,\, o_2}{\underset{o_1 = s_2}{\bowtie}})^{2,3} = \left\{ \begin{array}{c} (Robin,\ knows,\ Ted), \\ (Robin,\ knows,\ Dave), \\ (Bob,\ knows,\ Dave) \end{array} \right\}$$

□

5.3. Provenance for TriAL

TriAL* is an expressive algebra with respect to how RDF graphs can be traversed, as it allows us to query all variants inherent to the triple-based model of RDF recursively. However, expressive querying features are just one side of the coin, and meaningful results are the other. In TriAL* we need to stick to a triple-based representation, thus results are constrained to RDF triples. This appear convenient to derive new triples that describe, e.g. the transitive closure, and provide existential answers such as whether a path exists between two resources. Unfortunately, all information about intermediate resources which were traversed in order to derive a new triple are lost in this case. However, with the discussion on RDFPath we have seen how valuable such information can be. Path-based results which contain all the resources that were traversed along a path provide detailed insights into the actual structure of the graph and are therefore considered to be important in many application fields. Examples include, for instance, querying proteins in biology, where results which explain how proteins are interconnected to diseases are much more meaningful than those results which just state that there exists a connection [ABE$^+$09b, CDJ$^+$10]. The same applies for other domains, such as social networks or governmental data, where it is often crucial to understand *how* “things” are connected and *what* data pieces or sources were used to obtain a result.

A widely-used concept to describe the origin of a piece of data is called *data provenance* [BKT01, GKT07, CCT09, KG12, MBC13, GM13]. There exist many differently-grained provenance models in diverse application fields which allow to trace data through all transformations and exchanges. Considerable work as been done, for instance, with the so-called *why-provenance* for relational algebra introduced in [BKT01]. The basic idea is to trace all tuples which contributed *somehow* to an output record. What was missing in that idea with regard to TriAL*, is a more detailed description on *how* the traced tuples contributed actually to the output record, e.g. their order. More relevant for us is therefore the subsequent work on the *how-provenance*, which traces not only the tuples which contributed to an output record but also how they have been combined in order to derive the output record from the input data. The *how-provenance* is based on so-called *provenance semirings* [GKT07], an algebraic structure used by different kinds of provenance models [KG12]. A detailed study which illustrates these different notations can be found in [CCT09]. The provenance which we will introduce in this section can be also seen as a form of *how-provenance* but adopted to RDF, where the notation of *annotated relations* in semirings is extended to triples and only *one* description for each output record is allowed.

Another related concept to our work has been proposed by the World Wide Web Consortium (W3C). They defined a general concept of provenance in its PROV specification as *“information that describe the people, institutions, entities, and activities involved in producing a piece of data or a thing”* [MBC13, GM13]. Adopted to our scenario, where we want to track the provenance of triples, we can say that we trace each step of transformation that was applied on the input data to obtain the result. However, using the W3C PROV specification [GM13] to describe provenance in TriAL* would blow up our data model since the PROV specification implies to model provenance by interactions of agents, entities and activities. Instead, we need just a lightweight solutions that does not require extensive changes to TriAL* similar to the algebraic structure of provenance semirings.

We will introduce as next a particularly flexible provenance model to overcome the implicit loss of information inherent to the triple-based structure of TriAL*. It shares some beforehand mentioned ideas but allows us to track only those information, we are interested in. For clarity of the presentation, we will refer to TriAL* extended with provenance as *Extended Triple Algebra with Recursion*, or short E-TriAL*.

Data Representation. The basic idea of our provenance concept is to extend the triple-based data model of TriAL* by injecting one fourth element into the notation of a triple which will be then used to keep track of traversed resources. With the notation of RDFp in Section 4.2 (page 58), we have already proposed a solution which enables us to describe such traversed resources by means of paths. We simplify this notation for E-TriAL* in such a way, that the provenance of each individual RDF triple will be stored along the respective triple as a concatenated string.

Definition 5.10 (Triplestore with Provenance). More formally, assume $\mathcal{I}$ to be a pairwise disjoint, countably infinite set of international resource identifiers (IRIs), $\mathcal{B}$ the set of blank nodes and $\mathcal{L}$ the set of literals. Furthermore, to represent the provenance let $\mathcal{R}$ denote a set of words over IRIs, i.e. $\mathcal{R} = \mathcal{I}^*$. The underlying data model of E-TriAL* is then a so-called *quadstore* defined as the following quaternary relation:

$$E \subseteq (\mathcal{I} \times \mathcal{I} \times \mathcal{I} \times \mathcal{R}),$$

where $t \in E$ is a quadruple (s, p, o, r).

$\square$

Triple Algebra with Provenance. In TriAL*, a triple join describes the composition between two ternary relations, where only three out of six positions are kept in the results. For E-TriAL*, we extend this notation to eight positions, denoted by (s_1, p_1, o_1, r_1) and (s_2, p_2, o_2, r_2), referring to the left and right quadruple, respectively. The new elements r_1 and r_2 are then *words* [Har78, AHV95] representing the provenance of the respective triple. The composition of provenance elements is hereby becoming part of the query expression itself. This means, that in contrast to most other approaches [CCT09, KG12, GM13], we do not rely on predefined rules which derive the provenance of a triple *automatically* from the query expression and the data. Instead, we allow a query writer to describe for each TriAL* expression how to compose a representative provenance element. Such an approach might appear cumbersome at first glance, however, it proves to be a valuable and expressive mechanism that gives us lot of freedom in expressing exactly that provenance we are interested in and, e.g. leaving out information that might be redundant for a particular use case. In general, given two quadruples (s_1, p_1, o_1, r_1) and (s_2, p_2, o_2, r_2), a provenance element is composed as a word ϕ where

$$\phi \in \{s_1, p_1, o_1, r_1, s_2, p_2, o_2, r_2, \varepsilon\}^8.$$

We can see, that ϕ is a concatenation of eight words[2] in $\{s_1, p_1, o_1, r_1, s_2, p_2, o_2, r_2, \varepsilon\}$. Shorter words are constructed by means of the empty word ε. In order to distinguish IRIs, a blank is inserted between each word, i.e. between each IRI, as illustrated in the following example. Note, that it is allowed to choose from all positions from both quadruples in an arbitrary way, including repetitions. This gives us the desired flexibility to track only those information we are interested in and simply ignore non-informative resources.

Example 5.8. Consider two quadruples:

$$q_1 : (\textit{Robin},\ \textit{knows},\ \textit{Alice},\ \texttt{Robin knows Bob})$$
$$q_2 : (\textit{Alice},\ \textit{knows}\ ,\textit{Dave},\ \texttt{Alice knows Ted})$$

where, (s_1, p_1, o_1, r_1) and (s_2, p_2, o_2, r_2) refer to q_1 and q_2, respectively. For an exemplary ϕ, $\phi = r_1\ p_1\ r_2$ we obtain after replacing the three positions with their respective words the following new word as provenance:

$$\underbrace{\texttt{Robin knows Bob}}_{r_1}\ \underbrace{\texttt{knows}}_{p_1}\ \underbrace{\texttt{Alice knows Ted}}_{r_2}$$

□

We continue this section with a formal description on how the individual expression in TRIAL* need to be adapted in order to become compatible with our provenance model. For that, we start with introducing a modified selection operator that enables to perform some basic modifications on the provenance element. That might be, for instance, an initial step where provenance is added for the first time, thus the mapping of triples into quadruples.

[2]Conceptually, there is no reason for restricting the number of composed words to eight, however having such an upper bound is beneficial for the complexity of evaluation.

Definition 5.11 (Selection with Provenance). Let E be a quadruple store and θ, η as defined before. ϕ is a word which describes the provenance we are interested in. The *selection* operator in E-TriAL* $\sigma^{\phi}_{\theta,\eta}(E)$ is then defined as:

$$\begin{aligned}\sigma^{\phi}_{\theta,\eta}(E) := \{\ (a_i, a_j, a_k, \phi) \mid\ & a_i, a_j, a_k \in \{s, p, o\} \\ & \wedge \phi \in \{s, p, o, r, \varepsilon\}^4 \\ & \wedge t = (s, p, o, r) \in E \\ & \wedge \theta,\ \eta \text{ holds w.r.t. } s, p, o\}\end{aligned}$$

□

Definition 5.12 (Triple Join with Provenance). Let assume E_1 and E_2 to be two quadruple stores representing sets of quadruples. Formally, a *triple join with provenance* (denoted again by $\bowtie$) between E_1 and E_2 is defined as:

$$E_1 \overset{i,j,k\ \mid\ \phi}{\underset{\theta,\eta}{\bowtie}} E_2,$$

where,

- $i, j, k \in \{s_1, p_1, o_1, s_2, p_2, o_2\}$,
- θ, η as defined before,
- $\phi \in \{s_1, p_1, o_1, r_1, s_2, p_2, o_2, r_2, \varepsilon\}^8$

The result of $E_1 \overset{i,j,k\ \mid\ \phi}{\underset{\theta,\eta}{\bowtie}} E_2$ is then a quadruple store E_3, where $(a_i, a_j, a_k, \phi) \in E_3$ iff:

- $a_i, a_j, a_k \in \{s_1, p_1, o_1, s_2, p_2, o_2\}$,
- $t_1 = (s_1, p_1, o_1, r_1) \in E_1$ and $t_2 = (s_2, p_2, o_2, r_2) \in E_2$,
- each condition from θ and η holds w.r.t. s_1, p_1, o_1 and s_2, p_2, o_2.

□

The remaining operators of E-TriAL, namely *union*, *difference* and *intersection*, can be also equipped with the concept of provenance. Their definitions follow the same principles as shown above. We will therefore skip their formal description as it would not provide any new insights for the reader.

Example 5.9. Consider E describing the RDF graph in Figure 5.2 (page 99). The following E-TriAL* expression illustrates an initial step, where provenance is added to E by using the selection operator introduced in Definition 5.11.

$$E_0 = \overset{s\ p\ o}{\sigma}(E)$$

In that example, with $\phi = s\ p\ o$ the complete triple is specified as provenance. We therefore obtain for E_0:

$$\begin{array}{c}\{\ (\mathit{Robin},\ \mathit{knows},\ \mathit{Bob},\ \texttt{Robin knows Bob}),\\ (\mathit{Bob},\ \mathit{knows},\ \mathit{Alice},\ \texttt{Bob knows Alice}),\\ (\mathit{Alice},\ \mathit{knows},\ \mathit{Ted},\ \texttt{Alice knows Ted}),\\ (\mathit{Ted},\ \mathit{knows},\ \mathit{Dave},\ \texttt{Ted knows Dave})\ \}\end{array}$$

In a next step, we extend the transitive query used in Example 5.3 (page 98) that asks for pairs of users (x, y) connected by a path which exhibits a linear shape. Now, we are not only interested in the existence of pairs of users (x, y), but we want to exploit provenance to obtain information on the actual resources that were traversed on the path that connects both. For this, assume the following query:

$$E_1 = \left(E_0 \underset{o_1 = s_2}{\overset{s_1, p_1, o_2\ |\ r_1\ r_2}{\bowtie}}\right)^*$$

This expression uses $\phi = r_1\ r_2$ which means that the provenance of a newly derived triple is composed out of the provenance of both joined triples. The final result E_1 is then as follows:

$$\begin{array}{c}E_0 \cup\\ \{\ (\mathit{Robin},\ \mathit{knows},\ \mathit{Alice},\ \texttt{Robin knows Bob Bob knows Alice}),\\ (\mathit{Bob},\ \mathit{knows},\ \mathit{Ted},\ \texttt{Bob knows Alice Alice knows Ted}),\\ (\mathit{Alice},\ \mathit{knows},\ \mathit{Dave},\ \texttt{Alice knows Ted Ted knows Dave}),\\ (\mathit{Robin},\ \mathit{knows},\ \mathit{Ted},\ \texttt{Robin knows Bob Bob knows Alice Alice knows Ted}),\\ (\mathit{Bob},\ \mathit{knows},\ \mathit{Dave},\ \texttt{Bob knows Alice Alice knows Ted Ted knows Dave})\ \}\end{array}$$

□

The above examples illustrates how to use provenance in E-TriAL* to keep track of *all* triples which were used in order to derive a triple. We can notice that certain information, e.g. the resources which were part of the join condition appear multiple times. In case we want a more compact representation of provenance for that particular query, i.e without duplicate resources, we can modify the initial expression where provenances is added to E which we show in the next example.

Example 5.10. Consider E that describes again the RDF graph in Figure 5.2. The following E-TriAL* expressions adds in a first step provenance to E, where due to $\phi = s$ only the subject s is specified as provenance. The second expression which computes E_1 is the same as in the above example:

$$E_0 = \overset{s}{\sigma}(E)$$

$$E_1 = \left(E_0 \overset{s_1,p_1,o_2 \mid r_1\ r_2}{\underset{o_1=s_2}{\bowtie}}\right)^*$$

We obtain for E_0 as an intermediate result:

{ (*Robin*, *knows*, *Bob*, `Robin`),
(*Bob*, *knows*, *Alice*, `Bob`),
(*Alice*, *knows*, *Ted*, `Alice`),
(*Ted*, *knows*, *Dave*, `Ted`) }

The output E_1 is then:

E_0 ∪ { (*Robin*, *knows*, *Alice*, `Robin Bob`),
(*Bob*, *knows*, *Ted*, `Bob Alice`),
(*Alice*, *knows*, *Dave*, `Alice Ted`,
(*Robin*, *knows*, *Ted*, `Robin Bob Alice`),
(*Bob*, *knows*, *Dave*, `Bob Alice Ted`),
(*Robin*, *knows*, *Dave*, `Robin Bob Alice Ted`) }

We can see that each provenance entry in E_1 contains, due to the recursively derived provenance, all traversed users except the last one. In order to obtain a more complete provenance description that include also the last user, we can complement this example with a last simple selection operation applied at the end:

$$E_2 = \overset{r\ o}{\sigma}(E_1)$$

The above selection operation uses $\phi = r\ o$ to extend the provenance word r by adding the word o to the provenance. The final output E_2 is then:

{ (*Robin*, *knows*, *Bob*, `Robin Bob`),
(*Bob*, *knows*, *Alice*, `Bob Alice`),
(*Alice*, *knows*, *Ted*, `Alice Ted`),
(*Ted*, *knows*, *Dave*, `Ted Dave`),
(*Robin*, *knows*, *Alice*, `Robin Bob Alice`),
(*Bob*, *knows*, *Ted*, `Bob Alice Ted`),
(*Alice*, *knows*, *Dave*, `Alice Ted Dave`,
(*Robin*, *knows*, *Ted*, `Robin Bob Alice Ted`),
(*Bob*, *knows*, *Dave*, `Bob Alice Ted Dave`),
(*Robin*, *knows*, *Dave*, `Robin Bob Alice Ted Dave`) }

□

These examples highlighted some benefits of our flexible provenance model that enables to track exactly those information as provenance that one is interested in. We have seen that, on the one hand, it is possible to produce a granular description that contains *all* resources that were traversed in order to derive a triple. On the other hand, in cases where, a more compact view might be more helpful, one can select just a small fragment of all traversed resourced to become part of the provenance. That way, for instance, redundant information are left out and only those pieces of information are collected that describe the provenance one is interested in.

Termination of Kleene Closure. Recursive expressions in TriAL* involve the *Kleene star* as introduced in Definition 5.7. The authors in [LRV13, Vrg14] have shown that the semantics of these expression is the standard least-fixpoint semantics [AHV95] which can be evaluated with iterative algorithms. An recursive expression terminates then if a new iteration does not derive any *new* triples, i.e. all derived triples have been already discovered in previous iterations. With respect to E-TriAL*, we need to note that the quadruple representation which is used to keep track of provenance does not have an impact on that. The actual termination condition for recursive TriAL* expressions uses again only the triple, i.e. s, p, o, as input. The fourth provenance element is not taken into account. More information on the evaluation of recursive expressions will be discussed in Section 7.2, where we present algorithms and technical details of our TriAL-QL Engine.

5.3.1. Mapping Provenance to RDF

With provenance in E-TriAL*, we have introduced a valuable extensions that enables us to trace the origin of a triple. That way, we derive triples together with their description, which provides much more meaningful results that include, for instance, all resources which have been traversed along a path. However, if we recall that E-TriAL*s intention is to be part for the ecosystem of RDF management tools, where the results have to stay compatible with other Semantic Web languages, we need to find a way to transform the quadruple representation again into RDF triples. For that, we can benefit from the work which we have done for RDFPath in mapping RDFp to RDF. In Chapter 4.6 (page 85), we have introduced two mapping strategies. In case of the first one, we used again a quadruple representation, where the fourth element is a string concatenation of all resources. Since this does not solve our issue, we jump directly to the second mapping strategy where we introduced an ontology to represent paths as proposed by [KJ07]. That way, we were able to describe paths using RDF triples. The proposed definitions and algorithms in Section 4.6 can be – with just a few minor changes – directly applied on E-TriAL* results providing us the desired triple-based RDF representation. The changes include only three new properties to represent the *subject*, *predicate* and *object* of each triple.

Example 5.11. Consider as an example the following E-TriAL* result E, that is mapped into the corresponding triple-based representation Q by applying a slightly-adapted mapping function *rdfs()* from Definition 4.14 (page 87) on E.

$$E = \Big\{(Bob,\ knows,\ Ted,\ \texttt{Bob knows Alice knows Ted})\Big\}$$

$$\rightsquigarrow E.rdfs()$$

$$Q = \left\{\begin{array}{c}
(Bob,\ \_{:}p1,\ Ted),\\
(\_{:}p1,\ a,\ rdfp{:}Path),\\
(\_{:}p1,\ rdfp{:}subject,\ Bob),\\
(\_{:}p1,\ rdfp{:}property,\ knows),\\
(\_{:}p1,\ rdfp{:}object,\ Ted),\\
(\_{:}p1,\ rdf{:}\_1,\ Bob),\\
(\_{:}p1,\ rdf{:}\_2,\ knows),\\
(\_{:}p1,\ rdf{:}\_3,\ Alice),\\
(\_{:}p1,\ rdf{:}\_4,\ knows),\\
(\_{:}p1,\ rdf{:}\_5,\ Ted),\\
(\_{:}p1,\ rdfp{:}length,\ 5)
\end{array}\right\}$$

□

5.4. TriAL Query Language

While TriAL* is a neat and expressive approach for querying RDF, its algebraic notation is inappropriate for practical usage, where we need a more compact and writable language. Thus, we propose the TriAL* Query Language (TriAL-QL) that keeps the compositional structure of TriAL* by representing each algebra operation with a SQL-like statement. This way, even complex navigational queries are easy to grasp and write and, in addition, also all extensions introduced in Section 5.3 are supported. The basic idea is to flatten the algebraic expressions of TriAL* to a sequence of interrelated statements. A complete grammar can be found in Appendix C (page 225). Table 5.1 shows the algebra of TriAL* and E-TriAL* with the corresponding syntax in TriAL-QL. Each algebra operation is represented by exactly one SQL-like statement.

Table 5.1.: TriAL-QL Algebra & Syntax, where E, E_1 and E_2 correspond to a triplestore and $i, j, k, \phi, \theta, \eta$ as previously defined.

	Algebra	Syntax (TriAL-QL)
TriAL*	$\sigma_{\theta,\eta}(E)$	SELECT i, j, k FROM E FILTER θ, η
	$E_1 \bowtie^{i,j,k}_{\theta,\eta} E_2$	SELECT i, j, k FROM E_1 JOIN E_2 ON θ FILTER η
	$E_1 \cup E_2$	E_1 UNION E_2
	$E_1 - E_2$	E_1 MINUS E_2
	$E_1 \cap E_2$	E_1 INTERSECT E_2
	$(E \bowtie^{i,j,k}_{\theta,\eta})^*$	SELECT i, j, k FROM E ON θ FILTER η USING *right*
	$(\bowtie^{i,j,k}_{\theta,\eta} E)^*$	SELECT i, j, k FROM E ON θ FILTER η USING *left*
E-TriAL*	$\sigma^{\phi}_{\theta,\eta}(E)$	SELECT i, j, k WITH ϕ FROM E FILTER θ, η
	$E_1 \bowtie^{i,j,k \mid \phi}_{\theta,\eta} E_2$	SELECT i, j, k WITH ϕ FROM E_1 JOIN E_2 ON θ FILTER η
	$(E \bowtie^{i,j,k \mid \phi}_{\theta,\eta})^*$	SELECT i, j, k WITH ϕ FROM E ON θ FILTER η USING *right*
	$(\bowtie^{i,j,k \mid \phi}_{\theta,\eta} E)^*$	SELECT i, j, k WITH ϕ FROM E ON θ FILTER η USING *left*
	$(E \bowtie^{i,j,k \mid \phi}_{\theta,\eta})^{n,m}$	SELECT i, j, k WITH ϕ FROM E ON θ FILTER η USING $right(n, m)$
	$(\bowtie^{i,j,k \mid \phi}_{\theta,\eta} E)^{n,m}$	SELECT i, j, k WITH ϕ FROM E ON θ FILTER η USING $left(n, m)$

We can see that the syntax TriAL-QL stays close to its TriAL* and E-TriAL* expression illustrating the strength of a compositional language where the result of the first statement can be used as input for the second. This makes TriAL-QL easy to write, understand and modify. A further advantage is its extensibility, where new operators that extend the capabilities of our language can be added smoothly with new keywords. We will show this with operations that enable storing and retrieving of final but also intermediate results. Such features are well known from languages equipped with data manipulation as SQL. For TriAL-QL, we will introduce just a few basic expressions that allow us the reuse of results and gives more control over provenance in quadruple stores.

Data Manipulation. First, we extend the syntax of TriAL-QL with a STORE operation that enables us to materialize results E of an E-TriAL* expression e as a new relation in a quadruple store. This way, not only the output but also intermediate results can be stored for later processing, if desired. One further aspect which need to be considered is provenance. To facilitate dealing with it, we added an optional keyword PROVENANCE to TriAL-QL, which indicates whether provenance is kept or dropped while storing results. The syntactical rules are defined as follows:

STORE (PROVENANCE) E AS $identifier$

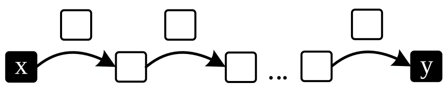
The corresponding counterpart of storing results is deleting them again. For this, a drop command is added, where one has the choice of removing the complete quadruple store or just its provenance using the following syntax:

DROP (PROVENANCE) $identifier$

Furthermore, an additional keyword, RDFS, is required to indicate the mapping of provenance into RDF triples. The corresponding expression for storing provenance as RDF is then:

STORE (RDFS) E AS $identifier$

Example 5.12. For an introductory example we consider again the reachability query from Example 5.3 (98), where we asked for pairs of users (x, y) connected by a path which exhibits linear shape visualized as follows:



A query in TriAL-QL that captures such a pattern together with its algebraic representation in TriAL* is then:

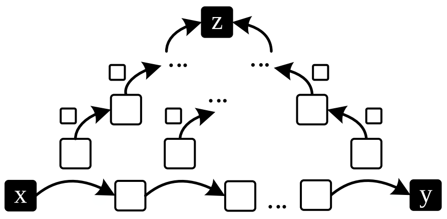
$$E_1 = (E \bowtie^{s_1,\ p_1,\ o_2}_{o_1=s_2})^*$$

$$\Big\downarrow$$

$$E_1 = \text{SELECT } s_1,\ p_1,\ o_2 \text{ FROM } E \text{ ON } o_1 = s_2 \text{ USING } \mathit{right}$$

□

Example 5.13. Next, we take the pattern from Example 5.5 (page 99) that asks for pairs of nodes (x, y) such that a connection exists from x to y where only one type of connection is allowed on the whole path. The type is again a recursive query which derives its property from the use of e.g. RDFS rules.



A query in TriAL-QL that captures such a pattern together with its algebraic representation in TriAL* is then:

$$E_1 = (E \bowtie^{s_1,\ o_2,\ o_1}_{p_1=s_2})^*$$
$$E_2 = (E_1 \bowtie^{s_1,\ p_1,\ o_2}_{o_1=s_2,\ p_1=p_2})^*$$

$$\Big\downarrow$$

$$E_1 = \text{SELECT } s_1,\ o_2,\ o_1 \text{ FROM } E \text{ ON } p_1 = s_2 \text{ USING } \mathit{right}$$
$$E_2 = \text{SELECT } s_1,\ p_1,\ o_2 \text{ FROM } E_1 \text{ ON } o_1 = s_2,\ p_1 = p_2 \text{ USING } \mathit{right}$$

The result is then a set of triples (x, z, y), such that

- there exists a connection from x to y, where
- each intermediate predicate has, for instance, the same superclass z.

□

Example 5.14. Next, let us consider the case where we are interested in provenance. More precisely, we want to keep information about the original property that connected two resources on the path between x and y. In the above queries, this property is replaced recursively in accordance with the expression e_1. This was required for the way e_2 is determining connections, since it needs to access recursively-derived properties. Moreover, as the original properties between x and y might have been different (otherwise we would not need nesting at this point), we additionally want to preserve the complete list of (original) properties between two resources. To do so, we can modify the expressions from the previous Example 5.13 as follows:

$$
\begin{aligned}
E_0 &= \sigma^{p}(E) \\
E_1 &= (E_0 \bowtie^{s_1,o_2,o_1 \ \mid \ r_1}_{p_1=s_2})^* \\
E_2 &= (E_1 \bowtie^{s_1,p_1,o_2 \ \mid \ r_1 \ r_2}_{o_1=s_2,\ p_1=p_2})^*
\end{aligned}
$$

$$\rotatebox{-90}{$\rightsquigarrow$}$$

$$
\begin{aligned}
E_0 &= \text{SELECT } s,\ p,\ o \text{ WITH } p \text{ FROM } E \\
E_1 &= \text{SELECT } s_1,\ o_2,\ o_1 \text{ WITH } r_1 \text{ FROM } E_0 \text{ ON } p_1 = s_2 \text{ USING } \textit{right} \\
E_2 &= \text{SELECT } s_1,\ p_1,\ o_2 \text{ WITH } r_1\ r_2 \text{ FROM } E_1 \text{ ON } o_1 = s_2,\ p_1 = p_2 \\
&\quad\ \text{USING } \textit{right}
\end{aligned}
$$

With E_0, we can see how provenance is added by the use of a selection operator, where just the current property ($\phi = p$) is specified as provenance. The subsequent expression which computes E_1 does not extend the provenance entry, but simply propagates it to all newly derived triples ($\phi = r_1$). The outer expression for E_2, which defines the actual traversal between two resources x and y creates for each new recursive step a new provenance entry by composing the previous ones ($\phi = r_1\ r_2$). The resulting provenance is then a description of all properties that have been traversed in order to derive a triple. □

Example 5.15. We can further increase the *informativeness* of the provenance in the above example and add each individual triple to the provenance which contributes in deriving a new triple. For that, we modify the expression as follows:

$$
\begin{aligned}
E_0 &= \sigma^{s\ p\ o}(E) \\
E_1 &= (E_0 \bowtie^{s_1,o_2,o_1 \ \mid \ r_1 \ r_2}_{p_1=s_2})^* \\
E_2 &= (E_1 \bowtie^{s_1,p_1,o_2 \ \mid \ r_1 \ r_2}_{o_1=s_2,\ p_1=p_2})^*
\end{aligned}
$$

$$\rotatebox{-90}{$\rightsquigarrow$}$$

$$
\begin{aligned}
E_0 &= \text{SELECT } s,\ p,\ o \text{ WITH } s\ p\ o \text{ FROM } E \\
E_1 &= \text{SELECT } s_1,\ o_2,\ o_1 \text{ WITH } r_1\ r_2 \text{ FROM } E_0 \text{ ON } p_1 = s_2 \text{ USING } \textit{right} \\
E_2 &= \text{SELECT } s_1,\ p_1,\ o_2 \text{ WITH } r_1\ r_2 \text{ FROM } E_1 \text{ ON } o_1 = s_2,\ p_1 = p_2 \\
&\quad\ \text{USING } \textit{right}
\end{aligned}
$$

In comparison to the previous example, the provenance entry for E_0 contains now not only the property but the complete triple ($\phi = s\ p\ o$). Furthermore, also each recursive step in the inner expression E_1 is now tracing its provenance, where each contributing triple is added completely to the provenance ($\phi = r_1\ r_2$). The final provenance in E_2 is then a description of *all* triples that have been used in order to derive a triple, including those which where traversed by E_1. □

5.5. Limitations of TriAL-QL

The high expressiveness complemented with a good evaluation complexity of E-TriAL* comes with a few more shortcomings, which we haven't discussed yet but that might hamper its practical usage. One crucial limitation of E-TriAL* is at the same time its most important property, the triple-based model. Having just three positions to work with is a substantial restriction for query writing, leading to complex nested queries in cases where more variables are required to express a desired pattern. This is especially present when multiple tests on data values are involved. For instance, consider a simple query that asks for pairs of users, which worked at the same department in the same year. Both constraints have to be expressed in separate expressions which then again need to be combined with a further expression. But in the end, this is mainly a syntactical hurdle which does not affect the expressiveness and hence, by using *union*, *minus* and *intersection*, even complex pattern can be captured as was shown in [LRV13]. Certainly, we could also think of a higher level language than TriAL-QL, which would aim to facilitate querying by having greater capabilities of expressing complex pattern in a short form. However, such a language has to be designed carefully in order to capture on one side the expressiveness of TriAL* while on the other side not inducing any extensions that would increase the complexity in evaluation. With TriAL-QL, we have chosen to design a language which stays close to the triple algebra. Accordingly, we could easily see that both properties remain preserved but unfortunately there are no short forms for these sort of multiple tests on data values.

Apart from that, it is also the relatedness of E-TriAL* to relational algebra itself which can be seen as a grave weakness. A relational view on data has, in contrast to a graph based one, some inherent implication of how certain pattern are expressed. If we consider for instance long-chained navigational queries with distinct traversing steps, each step needs to be described by an individual triple join. Whereas more graph-based languages like RDFPath provide querying capabilities geared towards a compact and intuitive way of expressing such patterns. However, this is solely a syntactical issue, since we could argue again with a more high level language on top of E-TriAL*.

Nonetheless, as we will see in Chapters 7, the advantages of having a language which is close to relational algebra outweighs its drawbacks with regard to implementation and optimization. For that, we highly benefit from well-studied algorithms and widely adapted techniques in the area of relational databases.

With RDFPath and TriAL-QL we have seen two orthogonal approaches capturing navigational queries. Both are equipped with expressive querying features, which cannot be expressed in most other RDF querying languages so far. Both exhibit also some fundamental issues that might hamper its practical usage for certain use cases. This provides the basis for further research questions which might investigate intersections between intuitive querying with meaningful results as shown in RDFPath and efficient yet simple algebra operations as introduced with E-TriAL* which benefit from their closeness to relational algebra. However, in the remaining chapters we will focus on the crucial point of this dissertation, the distributed processing of such expressive, navigational query languages, where we utilize a cluster environment to process them against RDF graphs at web-scale.

Part IV.

Distributed RDF Processing Techniques

6. RDF Processing with MapReduce

Contents

6.1. Motivation

With the emergence of semantic knowledge bases in diverse areas which include, for instance, biology, chemistry or the annotation of sports events, we can observe a steady growth of semantically-annotated data at a scale where querying the data becomes a challenging task. Consequently, distributed execution frameworks have been investigated for various query processing task that enable the work on web-scale RDF data. Recent examples include distributed SPARQL engines [HAR11, HMM+11, SPZL11] or OWL reasoning at large-scale [UKM+12]. With that in mind, we consider in particular the evaluation of complex, navigational RDFPath queries to be a data-intensive task that require solutions which allow the scale with the data size [HAR11]. Consequently, processing RDFPath queries against large RDF graphs can be also seen as an exhausting “Big Data” task.

In recent years, it has been widely recognized that the Hadoop ecosystem has become indispensable for many “Big Data” applications. System architectures for processing large amounts of data typically follow a layered approach: the front tier is responsible for answering simple queries in real-time as low latencies are essential. More complex analyses are performed *offline* in batches and results are pushed to the front tier in intervals (cf. e.g. [SKS13]). Typical representatives of such long-running queries are also navigational “friend-of-friend” RDFPath queries, if computed for all users in parallel. Due to its inherent high degree of parallelism and good scalability, MapReduce [DG08] is one of the predominant frameworks used for dealing with such large data. Although it might not be the most efficient solution wrt. node utilization, it gracefully handles load-balancing on top of commodity hardware, especially when it comes to rapidly-growing datasets where its built-in fault tolerance becomes another advantage. Thus, it is a natural candidate for processing our long-running RDFPath queries. This concept fits well with our initial motivation behind RDFPath to complement the existing ecosystem of tools and languages developed for the Semantic Web. RDFPath queries are then seen as a preprocessing task, where resulting RDFp paths are mapped again into a valid RDF graph as introduced in Section 4.6, such that they can serve as input for other RDF management systems. Moreover, if we consider HDFS as a common data pool that is shared across various Semantic Web applications, other systems are able to directly access the resulting RDF graph from our RDFPath MapReduce Processor. Hadoop-based SPARQL engine like Sempala [SPNL14] or S2RDF [SPSL15, SPSL16] can then be used as a low latency interface, where data does not even need to be duplicated or moved in order to be further processed.

Following this line of research, we proceed to investigate evaluation strategies for RDFPath queries with MapReduce on large RDF graphs in this section. After an initial introduction to Hadoop and the overall architecture of our RDFPath MapReduce Processor, we will start first with the underlying storage schema for RDFp graphs, named RDFp Store. RDFp Store is essentially our own serializable format for RDFp paths stored in HDFS. It integrates multiple optimizations such as binary competitors and dictionary encoding and provides efficient access to all individual resources of RDFp paths.

In addition, partitioning strategies which are applied in an initial preprocessing step reduce the overall amount of data that needs to be retrieved. Individual operations are explained by means of our RDFPath semantics, where we include a mapping from technical operations to the corresponding algebra expression. After that, we will continue with a strong focus on the actual join strategy used to compute traversing steps. Hereby, we will start with the commonly-used reduce-side join technique, followed by a short overview of alternative approaches. After identifying a promising candidate for improving the join performance of RDFPath queries, we introduce the concept of our Map-Side-Merge join. In order to provide more comprehensive comparisons and allow for some sort of comparison with other approaches, we base the experiments on SPARQL basic graph pattern (BGP). Although it does not provide the same expressiveness regarding navigational queries, it allows us to compare the crucial part of the evaluation, the actual join processing. Both RDFPath and SPARQL queries essentially break down into a sequence of joins. We can therefore assume that an improvement of BGPs evaluation performance in comparison to other techniques implies also an improvement in the evaluation of RDFPath queries. This is underpinned by the fact, that an RDFPath query of *fixed* length, thus without recursion, can be expressed as a BGP. To that end, we will present a distributed (n-way) sort-merge join on top of MapReduce, where the join is computed completely in the map phase. It addresses the problem of cascaded executions by using the reduce phase of MapReduce to assure that the “left-hand” side of the join is sorted wrt. the join attribute(s). Our data model assures that the “right-hand” side of the join is always presorted on the required attributes. For the reduction of intermediate results, bloom filters [Blo70, GWCL06] are used to remove dangling tuples. A comparison of other MapReduce-based join techniques showed that our approach exhibits a performance benefit of 15% to 48% on average in LUBM [GPH05], a benchmark which is used to compare the execution times of SPARQL BGP engines.

To the best of our knowledge, as of 2010 our RDFPath MapReduce Processor was the first navigational RDF query language implemented on top of MapReduce. Most of the results from this Chapter were published in [PSHL11] and in [PSHL12]. The map-side merge join optimization, which in contrast to related work [YDHJ07, HB10] does not require changes to the underlying MapReduce code, was published in [PSS$^+$13], as a general join technique for MapReduce. We can summarize the contributions of this chapter as follows:

- We provide an implementation for RDFPath that is based on MapReduce. The overall architecture of our RDFPath MapReduce Processor consists of three modules discussed in Section 6.2.
- In Section 6.3, we introduce RDFp Store, a data storage schema for RDFp paths. It comes with its own serializable format for efficient comparison and retrieval of RDFp resources and a partitioning strategy that reduces the amount of data that needs to be read from disk.
- The translation of RDFPath queries into MapReduce operations is discussed in Section 6.4, where we focus especially on join processing techniques.
- Two Map-Side merge joins, a 2-way and n-way sort-merge join, will be presented as an optimization for RDFPath in Section 6.5. Both joins are computed completely in the map phase and support a cascaded execution while using the reduce phase only for sorting. Further, bloom filters improve the overall performance by removing dangling intermediate results. Comprehensive experiments confirm in Section 6.5.4 the good performance of our join technique in comparison with other MapReduce joins and systems.

6.2. Distributed MapReduce Processor for RDFPath

We implemented a processor for RDFPath queries on *Apache Hadoop*[1], which is a collection of many open-source frameworks for distributed storage and processing on a cluster of machines. Before discussing the architecture of our implementation, we will give a briefly introduction into the two core Hadoop technologies HDFS and MapReduce.

Hadoop Distributed File System. The *Hadoop Distributed File System* (HDFS) is a distributed file system which follows conceptually the Google File System (GFS) introduced in [GGL03]. As one of the core components of Hadoop, it provides a distributed, fault-tolerant and common *data-pool* for the Hadoop ecosystem. Its basic idea is to split data into blocks, with a typical block size between 64 and 512 MB. Each block is then replicated across multiple machines, typically three. However, the replication is not only meant for fault-tolerance but has also a significant impact on the performance. This is due to one of the key concept for data processing on Hadoop, the usage of data locality. Instead of moving data to a calculation that probably runs on another machine, it is rather the calculation itself which is moved to the data by means of programming paradigms like MapReduce. Having multiple replications distributed across several machines increases the chances that data does not need to be moved but instead, can be locally processed. The architecture of HDFS is organized as a master/slave model. The master is the so-called *namenode* which stores metadata for all files and knows the locations of all blocks. The slaves, which are called *datanodes*, are then used to store the blocks of data.

MapReduce. The MapReduce programming model [DG08] enables scalable, fault tolerant and massively parallel computations on a cluster of machines. The essential idea is to provide a simple programming abstraction, where the task of parallelization is handled automatically. Developers express their algorithms in this model by specifying so-called `map()` and `reduce()` functions which are executed on the data in parallel. The underlying system makes use of data locality and avoids transfers of large datasets through the network if possible.

A conceptual view on the workflow of MapReduce is shown in Figure 6.1, where we can see that a single MapReduce iteration consists of a staged *map* and a *reduce* phase separated by a so-called *Shuffle* & *Sort* phase.

The main concept is that for each input record, which is usually represented as a key-value pair, a `map()` function gets invoked. Generally, this function is meant to apply filtering and extraction tasks on the input record. The output is then a list of key-value pairs derived from the input record. In the followed intermediate *Shuffle* & *Sort* phase, all intermediate key-value pairs are sorted in accordance to

[1]http://hadoop.apache.org/

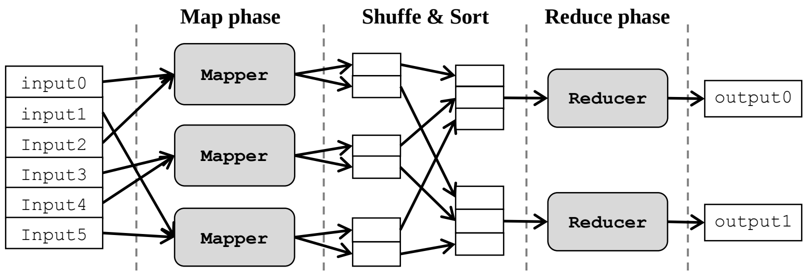


Figure 6.1.: MapReduce Dataflow

their *key*. This operation can be also understood as a `Group By (key)` operation in SQL, where all key-value pairs that share a key are assigned to the same so-called partition.

Next, a `reduce()` function gets invoked for each distinct key and thus individual partition, where the key and a list of all respective values is the input. This processing step closes one MapReduce iteration, by producing again a list of key-value pairs. In cases where a certain task needs to be expressed by multiple subsequent MapReduce iterations, the output of the `reduce()` functions is used again as input for the subsequent `map()`. The signatures of `map()` and `reduce()` functions can be therefore summarized as follows:

```
map():    (inKey, inValue)          -> list(outKey, tmpValue)
reduce(): (outKey, list(tmpValue)) -> list(outValue)
```

For a more detailed introduction into MapReduce we recommend [LD10, Whi15].

RDFPath Architecture. We can continue with an overview of complete architecture which is shown in Figure 6.2. The MapReduce-based implementation of RDFPath is mainly composed of three components: RDFp Store, RDFPath Loader, and RDFPath Processor.

- **RDFp Store:** The basis for evaluating RDFPath queries with MapReduce is a distributed storage for RDFp graphs. With HDFS there exists a distributed and scalable file system that allows us to build the RDFp Store on top of it. Main design goals were not only the overall reduction of data size that needs to be processed during query execution but also a unified representation that does not restrict the expressiveness and flexibility of RDFp. For a path-based language, which traverses the graphs by following predicates, usually just a small fraction of the data needs to be considered at once. With this in mind, we included a partitioning schema, dictionary encoding and further compression techniques using our own serializable data structures that capture the full strength of RDFp while enabling further optimization strategies. A detailed introduction is presented in Section 6.3.

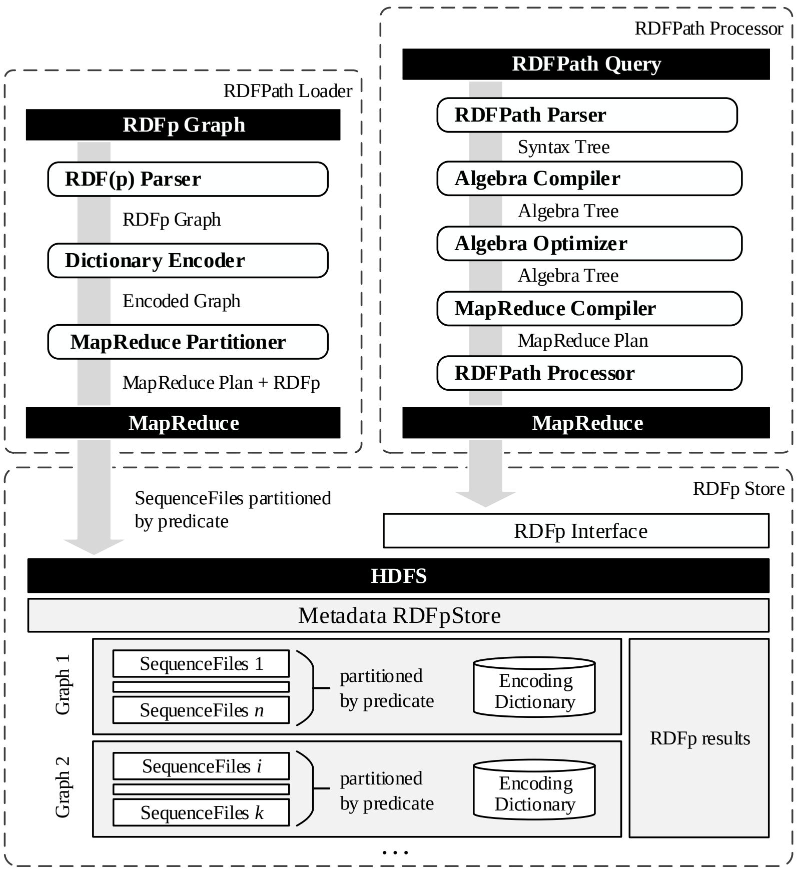


Figure 6.2.: RDFPath System Architecture

- **RDFPath Loader:** Loading new graphs in RDFp Store requires multiple tasks that need to be applied on the data once in advance. In a first step, the *RDF(p) Parser* translates RDFp graphs and diverse RDF serializations into a unified RDFp representation. After that, the (optional) *Dictionary Encoder* is used to create encoded RDFp graphs, where text-based IRIs are replaced by short numerical values. In a last step, our internal data structure based on a partitioning schema and compressed containers (so-called SequenceFiles) is created and stored in HDFS using MapReduce.

- **RDFPath Processor:** The core component of our implementation is the *RDFPath Processor* which (1) parses RDFPath queries, (2) translates them into their algebraic representation, and (3) creates a MapReduce plan that computes the corresponding expressions. The final MapReduce plan is then (iteratively) executed, where the results are stored again in HDFS as RDFp paths in SequenceFiles. Section 6.4 provides detailed insights on how the most crucial operations are expressed in the MapReduce model.

6.3. RDFp Store

RDFp was introduced in Section 4.2 as a flexible yet lightweight data model to represent paths in RDF graphs. It forms the basis for RDFPath (cf. Section 4.4), an expressive navigational language for RDF geared towards more meaningful, path-based results. In order to process next RDFPath queries with MapReduce, we first need to find a proper data representation for RDFp graphs in the Hadoop ecosystem which meets our requirements. First of all, data needs to be stored in a distributed way. Secondly, not only whole RDFp paths but also individual resources need to be accessible efficiently and allow for some sort of optimization. And lastly, intermediate but also final results should share the same unified representation, thus each iteration produces at any time a valid RDFp graph.

With the Hadoop Distributed File System (HDFS), distributed storage exists which can be used by any Hadoop application. It is well-known for its scalability and applicability for large data sets while providing enough flexibility to develop new data structures on top of it. This provides a simple yet elegant way to integrate our own application-specific data structures by means of so-called containers, which are automatically maintained in HDFS.

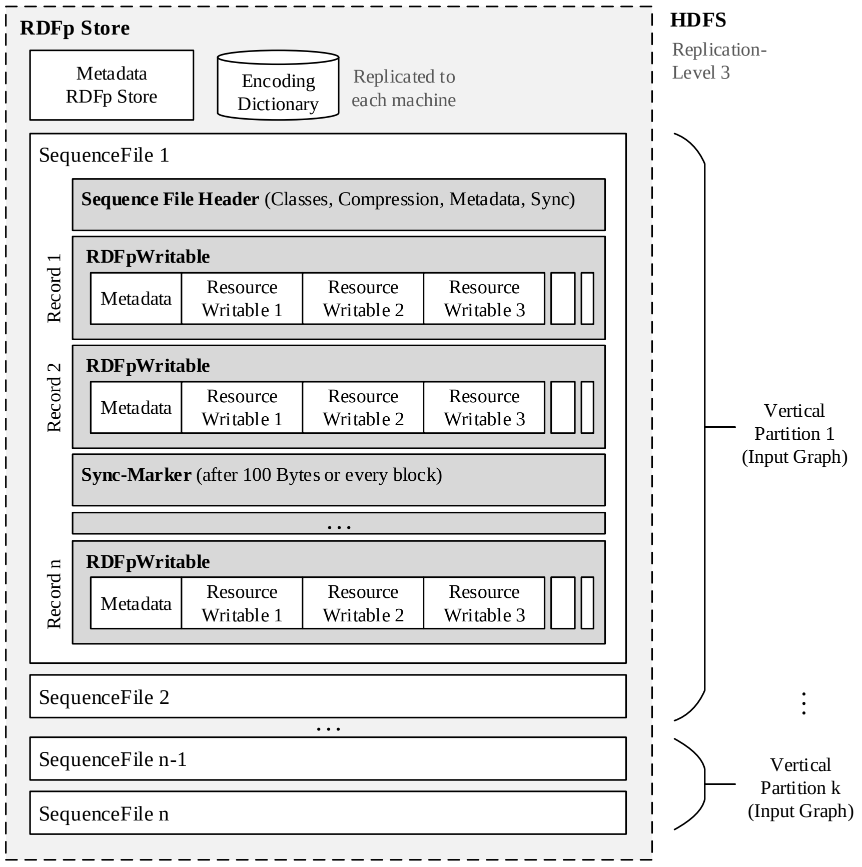


Figure 6.3.: Overview RDFp Store

Figure 6.3 illustrates an overview on how RDFP is represented in the Hadoop ecosystem. We can see that all the data resides in HDFS, but in a structured format, which we will introduce bottom-up in the following. We will start with a few technical low-level optimization, for instance the design of a single RDFP resource and its tight integration into HDFS, and proceed then with more high-level aspects like partitioning and compressing strategies that were built on top.

Modeling RDFp in Hadoop. Following the notation in Section 4.2, the basic element of an RDFP path is a resource r representing an RDF term with $r \in (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$. Such a resource is modeled by its own class named `ResourceWritable` which is shown in Figure 6.5. We can see that it implements the `WritableComparable` interface of Hadoop, a precondition if we want to (1) serialize (store and retrieve) objects in HDFS and (2) compare them with each other. Further, there are a few handy functions added to enable support for, e.g. different encoding strategies and data types. With a binary comparator, resources can be sorted in according to their natural order using their binary representation and thus there is no need to materialize resources in certain cases. This will be of particular interest for MapReduce, where we benefit strongly from fast comparison that avoids materializing objects.

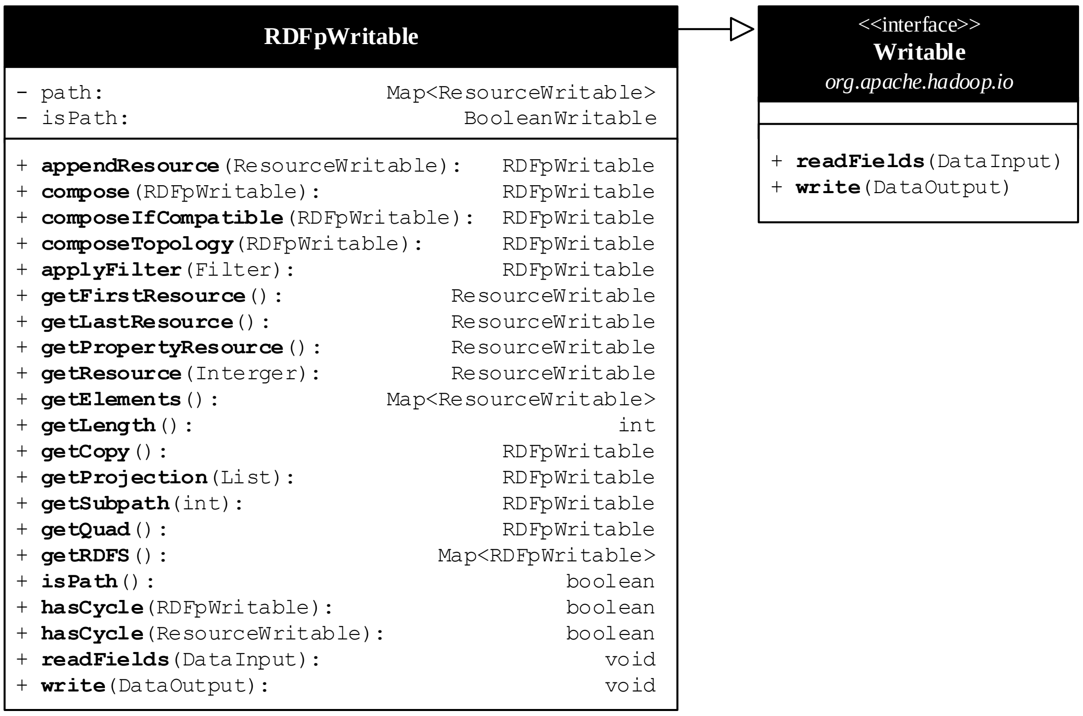

RDFpWritable
- path: Map<ResourceWritable>
- isPath: BooleanWritable
+ appendResource(ResourceWritable): RDFpWritable
+ compose(RDFpWritable): RDFpWritable
+ composeIfCompatible(RDFpWritable): RDFpWritable
+ composeTopology(RDFpWritable): RDFpWritable
+ applyFilter(Filter): RDFpWritable
+ getFirstResource(): ResourceWritable
+ getLastResource(): ResourceWritable
+ getPropertyResource(): ResourceWritable
+ getResource(Interger): ResourceWritable
+ getElements(): Map<ResourceWritable>
+ getLength(): int
+ getCopy(): RDFpWritable
+ getProjection(List): RDFpWritable
+ getSubpath(int): RDFpWritable
+ getQuad(): RDFpWritable
+ getRDFS(): Map<RDFpWritable>
+ isPath(): boolean
+ hasCycle(RDFpWritable): boolean
+ hasCycle(ResourceWritable): boolean
+ readFields(DataInput): void
+ write(DataOutput): void
<<interface>>
Writable
org.apache.hadoop.io
+ readFields(DataInput)
+ write(DataOutput)

Figure 6.4.: Class diagram of RDFpWritable

The general structure of the entire RDFP path is a n-ary tuple $p = (r_1, r_2, r_3, ..., r_n)$ composed of n resources r_i, with $r_i \in (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$. In RDFP STORE this is modeled by the class `RDFpWritable` which is meant to compose multiple `ResourceWritable` classes as shown in Figure 6.3. A more detailed description is shown in Figure 6.4, where we can see that it implements again the `Writable` interface of Hadoop and is composed solely out of serializable objects. But in addition, in order to optimize the execution performance, most crucial functionalities were added to this class to work directly on the data. Referring to the terminology introduced in Section 4.4 (page 71), we present in Table 6.1 a mapping of implemented functions to our RDFPath semantics.

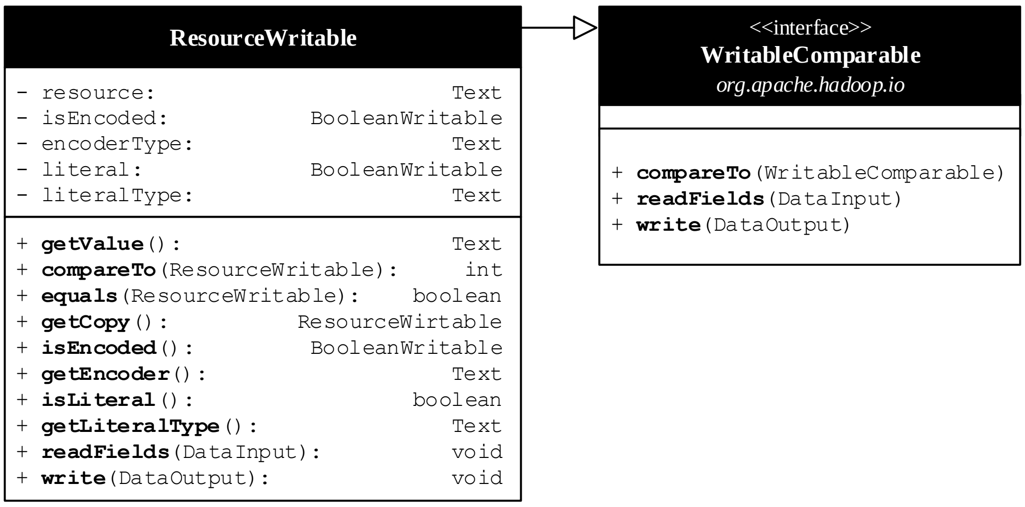

ResourceWritable
- resource: Text
- isEncoded: BooleanWritable
- encoderType: Text
- literal: BooleanWritable
- literalType: Text
+ getValue(): Text
+ compareTo(ResourceWritable): int
+ equals(ResourceWritable): boolean
+ getCopy(): ResourceWirtable
+ isEncoded(): BooleanWritable
+ getEncoder(): Text
+ isLiteral(): boolean
+ getLiteralType(): Text
+ readFields(DataInput): void
+ write(DataOutput): void
<<interface>>
WritableComparable
org.apache.hadoop.io
+ compareTo(WritableComparable)
+ readFields(DataInput)
+ write(DataOutput)

Figure 6.5.: Class diagram of ResourceWritable

Table 6.1.: Mapping of functions to RDFPath semantics, with `this.path` being the current path and reference refers to the formal definition.

Function	RDFPath Semantics	Reference
`composeIfCompatible(RDFpWritable nPath)`	`this.path` $\circ$ `nPath.path`	Def. 4.8, p. 72
`composeTopology(RDFpWritable nPath)`	δ_1(`this.path` $\circ$ `nPath.path`)	Def. 4.7, p. 72
`getFirstResource()`	π_1(`this.path`)	Def. 4.4, p. 71
`getLastResource()`	π_n(`this.path`)	Def. 4.4, p. 71
`getPropertyResource()`	π_2(`this.path`)	Def. 4.4, p. 71
`getResource(`i`)`	π_i(`this.path`)	Def. 4.4, p. 71
`getLength(`i`)`	$length$(`this.path`)	Def. 4.5, p. 71
`getProjection(`$list\ l$`)`	$\pi_{list\ l}$(`this.path`)	Def. 4.6, p. 72
`getSubpath(`i`)`	δ_i(`this.path`)	Def. 4.7, p. 72
`getQuad()`	`this.path` $.quad()$	Def. 4.13, p. 86
`getRDFS()`	$mapToRdfs$(`this.path`)	Alg. 4.6.1, p. 89
`hasCycle(RDFpWritable nPath)`	$cylce$(`this.path`, `nPath.path`)	Alg. 4.5.1, p. 79

An RDFp graph $\mathcal{P}$ is a set of RDFp paths p, such that $p \in \mathcal{P}$. In RDFp Store, such a set of `RDFpWritable` paths are serialized in a container called `SequenceFile`, which is also the base data structure for different types of files like `ArrayFile` or `MapFile`. These files not only contain data but might also have indexes or even bloom-filter integrated. In order to be used as input and output format in MapReduce, a proper `RecordReader` and `RecordWriter` need to be provided which enable reading and writing of records, in our case objects of type `RDFpWritable`. We decided to use a basic `SequenceFile`, without any indexes as it is supposed to be the fastest file format for sequential reading and writing. Unlike a B-tree, for example, a `SequenceFile` does not support seeking a specific key or modifying entries. However it is optimized for a high write-and-read throughput of objects and for this task it is a suitable choice that implements the aforementioned mentioned `Writable` interface. Moreover, it supports three storage strategies which can further improve the overall performance, specified in the `SequenceFileHeader` as illustrated in Figure 6.3. The following strategies were supported in the used version of Hadoop [Whi15]:

- **Uncompressed:** `RDFpWritable` objects are stored as illustrated in Figure 6.3, where the value is used to represent `RDFpWritable`
- **Record-compressed:** `RDFpWritable` objects are first compressed and then stored, where each value entry that keeps a `RDFpWritable` is compressed individually
- **Block-compressed:** `RDFpWritable` objects are first collected in blocks of configurable size and then compressed as a set.

To that end, we decided to use record-compressed `SequenceFiles` as they proved to provide a good tradeoff between compression and performance. For dictionary encoding, which we will introduce next, the actual storing strategy does not show a significant impact on data sizes but comes along with a higher computational effort. As a result, only uncompressed `SequenceFiles` were used for experiments with dictionary encoding.

Dictionary Encoding. A commonly-used optimization to reduce the overall size of data that needs to be managed is based on creating a dictionary in order to encode and decode individual words. Long text-based values are then replaced with a shorter representation. There is a wide range of approaches and technologies that can be used for that purpose [EGK95, UKOvH09, CMK+14, MWV+15]. However, as it was not a main focus of our work but just intended as a proof-of-concept experiment for MapReduce, we stuck to technologies that can be easily integrated into the Hadoop ecosystem. With this in mind, we decided to use Berkeley DB [OBS99, Ora16], an embedded key-value database library which is available in a Java Edition and does not require a dedicated infrastructure running in parallel to Hadoop. It can therefore be easily used within the MapReduce environment, where database files are simply shared via HDFS. A schematic overview is shown in Figure 6.6. There are two so-called *data stores* which maintain two *read*-optimized dictionaries. The first one is used to store the mapping of words to encoded values. This is required, since we want to keep the encoded values as small as possible and are therefore using auto-increasing integers. The second data store is used in order to decode the translated values back into their initial text-based form. Both dictionaries are created while loading the data into RDFp Store. For performance issues, both dictionaries are first replicated to each machine by changing the Hadoop replication factor to the number of machines available in the Hadoop Cluster. Next, a local copy of the *decoding* dictionary is created on each machine using the data replicated through HDFS. As those dictionaries are solely meant to decode values, transaction control and locking were disabled and the data store is configured for read-only access. With each RDFPath query that is evaluated over the data using dictionaries, a mechanism is triggered that checks whether the local copies are still up to date and replaces them if required.

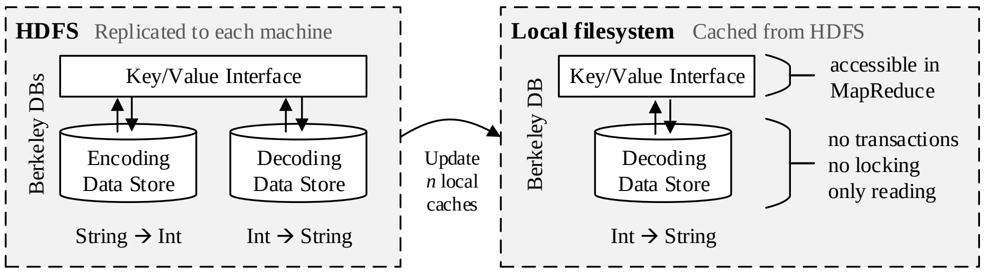


Figure 6.6.: Using Berkeley DB for dictionary encoding with MapReduce

In a subsequent work, we also investigated the usability of Cassandra [Hew10] and HBase [Geo11] for storing the dictionaries in a dedicated infrastructure running in parallel to Hadoop MapReduce. However, with an overall performance in orders of a few ten thousand record reads per second on a cluster with ten machines, both were by orders of magnitude not fast enough for decoding long-chained RDFp paths at web-scale, where each resource causes a new operation. This observation is in line with other research on benchmarking NoSQL systems, as they are optimized for low latency rather than high throughput in general [CST$^+$10, Cat10]. In comparison, using Berkeley DB we achieved a read performance of almost one million records per second on each machine [OB06].

Partitioning Strategies. Typically, RDF stores represent RDF datasets using some sort of partitioning and indexing strategies in order to improve the performance of evaluation. In our case, we basically need to consider two aspects for our layout. First, even though RDFPath supports diverse patterns, including querying the topology, for a navigational query language we tend to follow predicates more frequently than other patterns, as they appear closer to the natural interpretation of graphs. Second, the fact that the data needs to be distributed on a cluster of machines hampers the use of indexes. As a result, we use for the initial RDF dataset a representation that is similar to a so-called *triple table*, but partitioned by predicates. This principle – also known as *vertical partitioning* [AMMH07] – forms the ground base for our data layout and is often in use by RDF triplestores with DBMS back-end. In Figure 6.7, we can see how vertical partitioning is applied on our running example. For partitions which exhibit a certain size – as is often the case for partitions storing the commonly used predicate `type` – more than one `SequenceFile` is created. Please note that vertical partitioning is only applied once to the initial RDFp graph while loading the graph into our RDFp Store. Due to the fact that intermediate RDFp paths do not have a clear predicate, vertical partitioning is not applicable anymore.

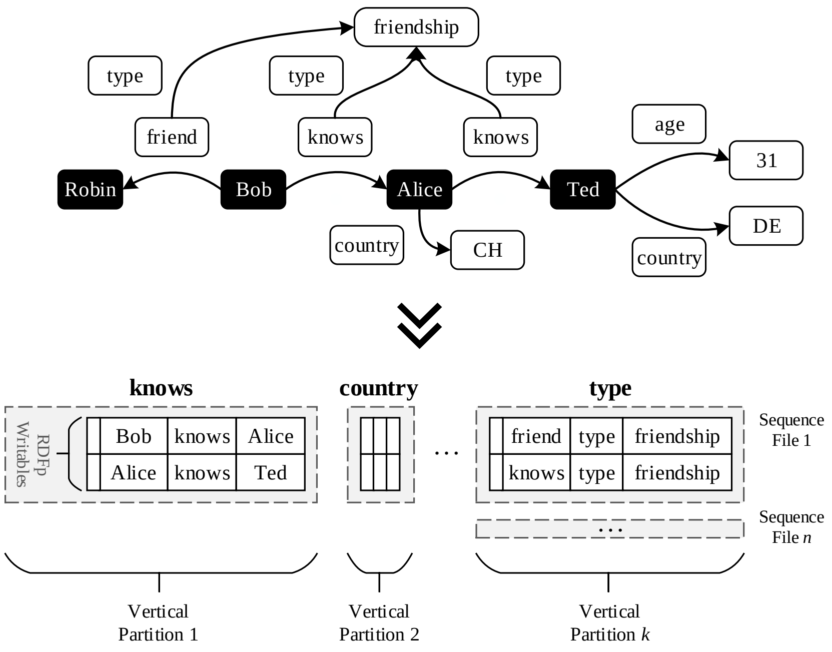


Figure 6.7.: Example for vertical partitioning in SequenceFiles

6.4. RDFPath MapReduce Processor

The translation of RDFPath queries follows a standard approach illustrated exemplary in Figure 6.8. The query asks for the Friends-of-Friend of Bob which are above 21 years old. In a first step, the RDFPath query ① gets parsed and its abstract syntax tree ② is generated by means of the EBNF grammar presented in Appendix B (p. 223). Next the abstract syntax tree is translated into an algebraic representation ③ using the RDFPath semantics discussed in Section 4.4. Hereby we can see that for our query, we need to access the RDFPath Store three times, where just two partitions are used in the end. Before continuing with the next translation step, we apply some widely-used optimization strategies discussed in [HH07, SSB$^+$08, SML10]. In the current example, the so-called *filter-pushing* approach is applicable. Rather than loading the complete partition for the predicate `age` and filtering afterwards, we can move the filter operation directly to the data access operation in the RDFPath Store. Note that, as we are encoding only text-based values, an efficient numerical comparison remains feasible. In cases where we have a filter on a text-value, we first need to use the dictionary in order to translate the corresponding value(s) into its encoded numerical value. The algebra is then evaluated bottom-up, where for each algebra operation a corresponding MapReduce job is added to the MapReduce execution plan ④.

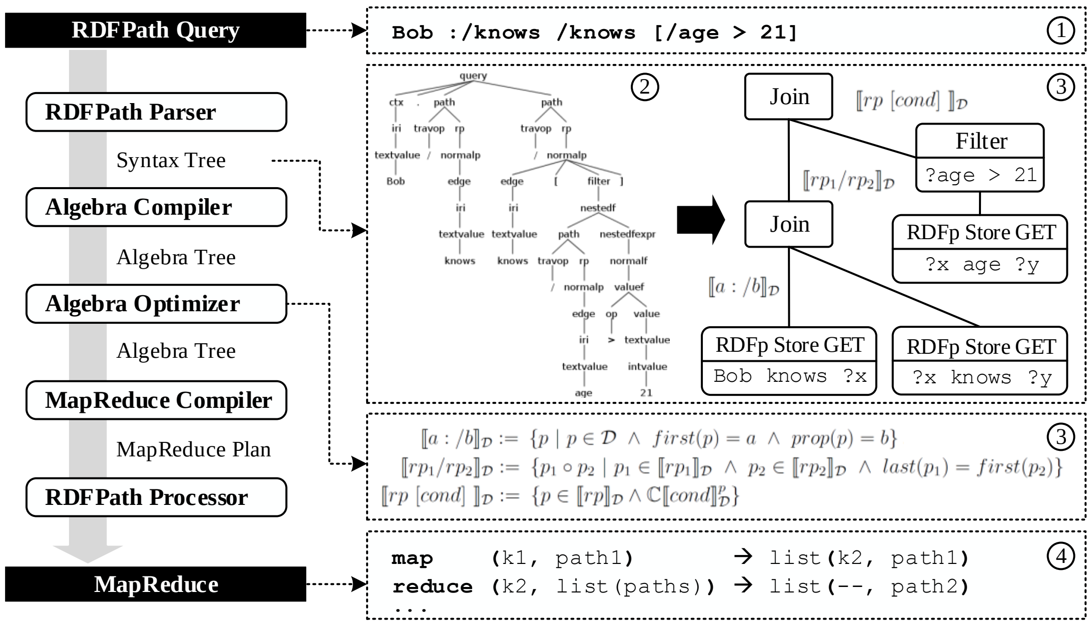

RDFPath Query
RDFPath Parser
Syntax Tree
Algebra Compiler
Algebra Tree
Algebra Optimizer
Algebra Tree
MapReduce Compiler
MapReduce Plan
RDFPath Processor
MapReduce
Bob :/knows /knows [/age > 21]
①
②
③
④
Join
Filter
?age > 21
RDFp Store GET
?x age ?y
Bob knows ?x
?x knows ?y
$[\![rp\ [cond]\ ]\!]_\mathcal{D}$
$[\![rp_1/rp_2]\!]_\mathcal{D}$
$[\![a : /b]\!]_\mathcal{D}$
$[\![a : /b]\!]_\mathcal{D} := \{p \mid p \in \mathcal{D} \wedge first(p) = a \wedge prop(p) = b\}$
$[\![rp_1/rp_2]\!]_\mathcal{D} := \{p_1 \circ p_2 \mid p_1 \in [\![rp_1]\!]_\mathcal{D} \wedge p_2 \in [\![rp_2]\!]_\mathcal{D} \wedge last(p_1) = first(p_2)\}$
$[\![rp\ [cond]\ ]\!]_\mathcal{D} := \{p \in [\![rp]\!]_\mathcal{D} \wedge \mathbb{C}[\![cond]\!]^p_\mathcal{D}\}$
map (k1, path1) → list(k2, path1)
reduce (k2, list(paths)) → list(--, path2)
...

Figure 6.8.: Translating an RDFPath query into MapReduce

6.4.1. Mapping to MapReduce

Up to now, we have seen the overall architecture behind RDFPath, discussed how RDFp paths are represented in the RDFp Store and presented the translation process of an RDFPath query into MapReduce jobs. Next we will address the most important part, which is the evaluation strategy that we use with MapReduce. In Figure 6.8 step ③, we have seen that the computation of traversing steps in RDFPath, which are clearly the most important operations, essentially breaks down to the evaluation of a series of joins. Indeed, processing joins proves to be at the core of evaluating such data-intensive tasks, on which we will focus next.

Generally we can state that processing joins on large datasets is, even with MapReduce, a challenging and costly task [BPE+10, LLC+11]. One has to consider many parameters. For instance, a proper data partitioning and distribution strategy, which have a significant impact on the overall performance while having at the same time just two simple functions – `map()` and `reduce()` – and a staged workflow available to express a join. In order to understand the problem setting more precisely, we introduce some general terminology in the following. Given two datasets with RDFp paths L and R stored in HDFS, split into blocks and spread across a cluster of machines, if we want to join both datasets on their join keys $L.k$ and $R.k$ (k can be assumed a partial data entry extracted from an RDFp path), we have to ensure that a subset of L with the join-key $k1$ is brought together with the corresponding subset of R, such that all records with $L.k1 = R.k1$ can be processed on the same machine as a partial result of the whole join between L and R. Unfortunately, this is rarely the case in MapReduce. Data blocks are distributed and replicated randomly across all machines in the cluster. Thus, if we want to join arbitrary datasets on arbitrary keys, data needs to be shuffled over the network or alternatively appropriate pre-partition and replication strategies need to be developed.

Reduce-Side Joins. The simplest but most versatile join technique in MapReduce is called a *Reduce-Side Join* [LD10, Whi15, LLC+11]. Some literature refer to it as *Repartition Join* [BPE+10] as the basic idea is based on reading both datasets completely and repartition them according to the join key. An initial preprocessing step like pre-partitioning can increase the efficiency but is not required. The basic idea is based on `map()` functions which emit extracted join keys together with a dataset ID (i.e. L or R) as a composite intermediate key. The record itself (an RDFp path) is emitted as the corresponding intermediate value. The shuffle phase groups the key-value pairs according to the intermediate join key such that all values with the same join key are sent to the same reducer, i.e. to the same machine. An appropriate sorting comparator makes sure that records from dataset L occur previous to records from dataset R or vice versa. Accordingly, `reduce()` functions get a sorted list of values as input, which have the join key in common. And that is exactly what is required to process a join in the reduce phase. The first part of the list has to be held in memory whereas the second one is processed as a stream of data for joining both. The main drawback of this approach is that both datasets are completely transferred over the network regardless of the join output. This is especially inefficient for selective queries and consumes a lot of network bandwidth. Furthermore, we have to hold one of both datasets in memory while the other one is streamed. In some cases this may exceed the capacities of individual machines [BPE+10], particularity if the data appears to be skewed. Nonetheless, we based our first prototype of RDFPath on Reduce-Side joins, due to their inherent flexibility and the absence of additional pre-computation costs which would have been required by more efficient join algorithms like *Map-Side joins*, discussed later.

Next, in order provide more technical insights on the actual implementation, we present a few details of our adapted Reduce-Side join, which is used to compute traversing steps in RDFPath. We start with the `map()` function, which reads RDFp paths stored in RDFp Store (cf. Figure 6.3). Its pseudo-code is shown in Algorithm 6.4.1, where we can see that it basically follows the previously mentioned Reduce-Side join strategy. For each RDFp path p_i, a composite key with two entries (r_i, id_i) is extracted. Both the composite key (r_i, id_i) and the corresponding path p_i are then emitted as key-value intermediate results. The first entry r_i contains the actual join key which is used as an identifier for *partitions* in the subsequent shuffle and sort phase. It ensures that all records which share this key r_i end up in the same partition p_i and are forwarded within a sorted list to the same reducer. However, the *default partitioner* which is used to assign keys to partitions does not support a composite key structure. For that, a new partitioner needs to be provided, which enables MapReduce to understand the composite key structure and partition the records according to the desired value. In this case, the partitioner needs to consider simply the first entry r of the composite key and ignore the second component. Furthermore, in order to improve the overall performance a small optimization can be applied here. The *binary comparator*, which was mentioned in Figure 6.4, can be adopted to avoid materializing the composite key for comparisons which are required for sorting and partitioning in the shuffle and sort phase of MapReduce.

The order of records within each partition, or more precisely the order within individual lists which are passed as input to reducers, needs then to be sorted on the second entry of our composite key, the *id*. Recall that with this information we distinguish between both datasets we want to join. Thus, to avoid unnecessary I/O we need to ensure reading the input data for each reducer just once. Actually, this can be achieved by sorting all RDFp paths, which were emitted with $id = 0$, ahead of all those which were emitted with $id = 1$. To do so, again a small modification of a class is required, which specifies the MapReduce workflow. With a new so-called *grouping comparator*, we can supplement MapReduce such that composite keys are supported, where just the second element *id* is considered and the previously used join key r is ignored. Again, also here we can enhance the overall performance of comparisons by providing a binary comparator which avoids materializing composite keys.

Algorithm 6.4.1 : map() function of reduce-side join

```
input  : NullWritable $key$, RDFpWritable $path$
output : list (RDFpPairWritable, RDFpWritable)
1  ResourceWritable $r \leftarrow \emptyset$              // identifier for partitioner
2  int $id \leftarrow 0$                       // identifier for grouping comperator
3  RDFpPairWritable $pair \leftarrow \emptyset$          // composition of both identifiers
4  if $path$.isLeftJoinSet() then
5      $r \leftarrow path$.getLast()
6      $id \leftarrow 0$
7  else
8      $r \leftarrow path$.getFirst()
9      $id \leftarrow 1$
10 end
11 $rid \leftarrow (id, r)$
12 emit ($pair$, $path$)
```

Algorithm 6.4.2 : reduce() function of reduce-side join

```
input  : RDFpPairWritable $pair$, list(RDFpWritable) $\mathcal{P}$
output : NullWritable, RDFpWritable
1  $\mathcal{P}$              // sorted list of all RDFpWritables that share a key
2  $\mathcal{L} \leftarrow \emptyset$              // set of RDFpWritables (paths of left join set)
3  foreach $p_1 \in \mathcal{P}$ do
       // paths from the left join set are ahead of others
4      if $p_1$.isLeftJoinSet() then
5          $\mathcal{L} \leftarrow \mathcal{L} \cup p_1$
6      else
           // only paths from right join set appear from now in $\mathcal{P}$
7          foreach $p_2 \in \mathcal{L}$ do
8              $p_{new} \leftarrow p_2$.composeIfCompatible($p_1$)
9              emit (null, $p_{new}$)
10         end
11     end
12 end
```

Algorithm 6.4.2 shows the pseudo-code for the `reduce()` function, which is called for each individual join key r, where all entries that share a join key are grouped in the same ordered list. As it is ensured that all values with the dataset $id = 0$ (*right hand side* of join) are sorted ahead of values with dataset $id = 1$ (*left hand side* of join) the whole list needs to be streamed once through the reducer. Therefore we first store all right-hand side values of the join in main memory (line 5), and compute then for each further record from the left-hand side the respective join (line 8) and emit the result in form of a newly composed RDFP path.

Example 6.1. To illustrate a reduce-side join in RDFPath, we consider once more the running example from Figure 6.7 (page 130), which models the friendship between four persons including some demographic information. We will refer to this RDF Graph as R. The following RDFPath query asks for all friends of `Bob` which are reachable by traversing `knows` or `friend` an arbitrary number of times. After that, with a subsquent traversing step that includes a filter constraint we determine their country and ensure that we consider only those friends in germany.

```
Bob :/ (knows | friend)* /country[=DE]
```

Applying the exemplary query on RDF Graph R and assuming that we already processed the query so far that just the last traversal is left to evaluate, the following join needs to be computed:

$$\left\{\begin{array}{c}(Bob,\ knows,\ \boldsymbol{Robin})\\(Bob,\ knows,\ \boldsymbol{Alice})\\(Bob,\ knows,\ Alice,\ knows,\ \boldsymbol{Ted})\end{array}\right\}\underset{join}{\overset{reduce-side}{\bowtie}}\left\{\begin{array}{c}(\boldsymbol{Alice},\ country,\ CH)\\(\boldsymbol{Ted},\ country,\ DE)\end{array}\right\}$$

In a first step, the `map()` function is applied on each RDFP path from both input datasets, where we obtain the following intermediate results[2]. The composite key for each RDFP path is composed of the respective join key, in this case the names of friends, together with a dataset id, that is used to distinguish the left and the right hand side of the join.

$$\begin{array}{lcl}p_1 : (Bob,\ knows,\ \boldsymbol{Robin}) & \overset{\texttt{map()}}{\leadsto} & ((\boldsymbol{Robin},\ 0),\ p_1)\\ p_2 : (Bob,\ knows,\ \boldsymbol{Alice}) & \overset{\texttt{map()}}{\leadsto} & ((\boldsymbol{Alice},\ 0),\ p_2)\\ p_3 : (Bob,\ knows,\ Alice,\ knows,\ \boldsymbol{Ted}) & \overset{\texttt{map()}}{\leadsto} & ((\boldsymbol{Ted},\ 0),\ p_3)\\ q_1 : (\boldsymbol{Alice},\ country,\ CH) & \overset{\texttt{map()}}{\leadsto} & ((\boldsymbol{Alice},\ 1),\ q_1)\\ q_2 : (\boldsymbol{Ted},\ country,\ DE) & \overset{\texttt{map()}}{\leadsto} & ((\boldsymbol{Ted},\ 1),\ q_2)\end{array}$$

Next in the shuffle and sort phase, data is partitioned in accordance with the first part of the composite key, thus on names of friends. This way, we bring together people and their country as both pieces of information will end up in the same partition. Within a partition, the second part of the composite key is used to maintain

[2]Please note, that $(Alice,\ country,\ CH)$ will be actually filtered out much earlier, directly after reading it for the first time. Yet, for clarity of presentation we kept it.

an order, where the list of friends appears ahead of their country information. The first part has to be kept in main memory. After that, a join is applied between all values from the first part which is held in memory and with each value from the second part of the list. In the continued example shown next, we can observe that in total three partitions were created, one for each individual friend. The partition for *Robin* and *Alice* provides no results, since we do not have sufficient information for *Robin*, and those for *Alice* do not satisfy our constraint $country = DE$. At the end, just one result is emitted, as in our example *Ted* is the only person who had the desired country information. The resulting RDFp path is then a composition of the friends of a friend chain that starts with Bob and their respective country.

$$Robin: \Big\{(Bob,\ knows,\ \boldsymbol{Robin})\Big\} \quad \overset{\texttt{reduce()}}{\rightsquigarrow} \quad \emptyset$$

$$Alice: \left\{\begin{array}{c}(Bob,\ knows,\ \boldsymbol{Alice}),\\ (\boldsymbol{Alice},\ country,\ CH)\end{array}\right\} \quad \overset{\texttt{reduce()}}{\rightsquigarrow} \quad \emptyset$$

$$Ted: \left\{\begin{array}{c}(Bob,\ knows,\ Alice,\ knows,\ \boldsymbol{Ted}),\\ (\boldsymbol{Ted},\ country,\ DE)\end{array}\right\} \quad \overset{\texttt{reduce()}}{\rightsquigarrow}$$

$$(Bob,\ knows,\ Alice,\ knows,\ Ted,\ country\ ,\ DE)$$

□

To wrap up, we have seen that computing an RDFPath query essentially breaks down into the evaluation of joins, where each individual traversal step is expressed by a reduce-side join. The development of more efficient joins is therefore a fundamental cornerstone for improving the overall evaluation performance of RDFPath queries. Conveniently, this is also the case for the W3c recommended RDF query language SPARQL, where the processing of so-called basic graph patterns, which are at the core of SPARQL, also breaks down to the evaluation of joins. This intersection enables a much more comprehensive comparisons due to manifold join optimization and evaluation strategies available for SPARQL on top of Hadoop. We will continue introducing further join techniques which aim to improve the performance of RDFPath queries. However, for evaluation purposes we will come back to SPARQL's basic graph pattern, which allows us to provide more comparisons with other related work.

Further Join Techniques. In addition to these widely used *Reduce-side Joins*, there are several other join techniques focusing on certain join types like *Theta-Joins* [OR11] or optimizing existing join techniques for particular application fields [AU11]. In [YDHJ07] the authors proposed a modified MapReduce workflow by adding a merge phase after the reduce phase, whereas the authors in [JTC11] propose a join phase in between the map and reduce phase. Both approaches attempt to improve the support for joins in MapReduce but require profound modification to the MapReduce framework. There are also further join techniques that exploit the idea behind *Semi-Joins* to reduce the amount of data transferred through the network. Unfortunately these approaches require an additional MapReduce phase to identify unnecessary records [HB10, BPE+10]. Jens Dittrich et al. proposed in [DQRJ+10] a new index and join technique called *Trojan Indexes* and *Trojan Join* that do not require modifications to the MapReduce framework. These techniques avoid shuffling data during the join execution at the cost of a preprocessing stage containing a co-partitioning and index-building step at load time.

Broadcast Join. Another group of joins is based on getting rid of the shuffle and reduce phase to avoid transferring both datasets over the network. This kind of join technique is called a *Map-Side Join* since the actual join processing takes place in the map phase [Whi15]. The most common one is the *Broadcast Join* [BPE+10] or *Memory-Backed Join* [LD10] respectively. This join can be applied if one of the two datasets, let us assume R, is small enough to be transferred to each machine in advance. This can be achieved either by copying R to the Distributed Cache or by increasing the HDFS replication factor of R [Whi15].

For joining L with R the map phase becomes initialized with L as input such that each Mapper gets a subset of L assigned. Each Mapper needs to retrieve R and create a main-memory hash table [BPE+10]. Accordingly the actual join can be processed as a relational hash join without transferring L through the network, as each mapper can compute a partial join for his subset of L independently.

Although it can be stated that *Broadcast Joins* are an efficient way for joining larger datasets with smaller ones, this approach is only applicable if there is information available about the dataset sizes in advance, such that it can be ensured that the smaller dataset fits into main-memory. Furthermore it increases the starting costs for the MapReduce job as the smaller dataset have to be redistributed before each join execution if no caches or sufficient high replication factors are used.

Map-Side Join. A further join technique that belongs to the group of *Map-Side Joins* we will optimize in the next section is called a *Map-Side Merge Join* [LD10, LLC+11]. However, such a join cannot be applied on arbitrary datasets. A preprocessing step is necessary to fulfill several requirements: First of all, the datasets have to be sorted according to the join key. Next, both have to be equally partitioned, such that a partial merge join can be applied on two sorted partitions independently.

This sounds very strict and at least costly to achieve with MapReduce and HDFS. Indeed, the output of a MapReduce job (including shuffle and reduce phase) matches exactly those requirements [Whi15]. Accordingly, a slightly-modified so-called *identity* mapper and reducer is sufficient to produce the desired results. If different join keys are required, this preprocessing step has to be repeated for each additional join key such that we have a pre-sorted and pre-partitioned data replica for each individual join key stored in HDFS. Finally, the map phase can process an efficient merge join between pre-sorted partitions independently on a cluster of machines. Shuffling data is reduced, but with the constraint that only one sequential join is possible. For a sequence of joins, the shuffle and sort phase is needed again to preprocess intermediate results of previously computed joins.

Summarized, *Map-Side Merge Joins* are more efficient but less flexible than *Reduce-Side-Joins* as they require a time and space-consuming preprocessing step for each join key and thus are not applicable in general for a sequence of joins. In the next section, we will focus on this aspect by providing an optimized *Map-Side Merge Join* technique, which enables us to compute subsequent joins while reducing the overall amount of data that needs to be pre-processed for each join.

6.5. Map-Side Merge Joins for MapReduce

Contents

To further improve the performance and scalability of our RDFPath MapReduce Processor on web-scale RDF data, we proceed to investigate the efficient evaluation of joins with MapReduce. Since both the computation of RDFPath queries and SPARQL BGPs essentially translates to the evaluation of joins on the operator level, we base this section on SPARQL basic graph patterns (BGPs) which enables us to provide more comprehensive comparisons with other commonly-used approaches. Processing joins is an area which has been extensively studied by the database community in the past [Gra93], where merge joins have emerged as a widely-adopted solution used in many databases. It would also be promising to investigate their applicability in a distributed environment based on MapReduce. However, this poses multiple challenges. First of all, merge joins are only applicable on datasets which are sorted in accordance with the join key. Although not problematic in the case of just one join, where a preprocessing step can sort the data in accordance to all possible join keys, computing a sequence of cascaded merge join is a non-trivial task. One has to consider that, in order to apply a subsequent merge join, all intermediate results ("left-hand" side of the join) need to be resorted again, which typically involves an additional and costly MapReduce phase.

To overcome these issues, we will present a distributed (*n-way*) *sort-merge join* on top of MapReduce, where the join is computed completely in the map phase. It addresses the problem of cascaded executions by using the reduce phase of MapReduce to assure that the "left-hand" side of the join is sorted wrt. the join attribute(s). Our data model assures that the "right-hand" side of the join is always pre-sorted by the required attributes. For the reduction of intermediate results, bloom filters [Blo70, GWCL06] are used to remove dangling tuples. A comparison of our approach to other MapReduce-based join techniques showed that our approach exhibits a performance benefit of 15% to 48% on average over all LUBM [GPH05] queries.

The remainder of this section is structured as follows: We start with a conceptual overview of our approach in Section 6.5. Section 6.5.1 introduces the data store layout for our merge join implementation and the actual join algorithms are discussed afterwards in Section 6.5.3. Section 6.5.4 reports the results of our experimental comparison by means of different SPARQL BGP implementations, which provide more comprehensive comparisons than one would get by a comparison that is based on RDFPath queries.

Our Approach in a Nutshell. In order to perform a join between two datasets L and R with MapReduce, i.e. $L \bowtie R$, we need to ensure that the subsets of L and R with the same join key values can be processed on the same machine. The approach which we present in this section is an adaptation of the classical sort-merge join, a well known technique in database systems. Our goal is to perform the join completely in the map phase which reduces network I/O. The key idea is to first sort datasets L and R by the join key such that identifying equal values in both datasets can be done using interleaved linear scans. Consequently, the first thing we have to guarantee is that L and R are always sorted by join key and also that the join output has to be sorted according to the join key of the next join iteration in a sequence of joins. Furthermore, for an efficient parallel execution in a cluster of N machines we have to divide the join task into N independent subtasks where each subtask can be processed by exactly one machine. Therefore, we split both (sorted) datasets in N non-overlapping subsets of continuous key ranges such that

$$L = \bigcup_{1 \leq i \leq N} L_i \quad \text{and,} \qquad\qquad R = \bigcup_{1 \leq i \leq N} R_i$$

If we use the same key ranges for both datasets (cf. Figure 6.9), it holds that

$$L \bowtie R = \bigcup_{1 \leq i \leq N} L_i \bowtie R_i$$

Our data preprocessing and store layout is described in detail in Section 6.5.1. Driven by this data partitioning, the map phase can process an efficient parallel merge join between pre-sorted dataset splits. We use the reduce phase only to guarantee that the join output fulfills the preconditions for the next iteration, i.e. it must be sorted according to the next join key and be split into N subsets such that key ranges match with the next join partition. Furthermore, we use *dynamic bloom filters* to discard *dangling tuples* in intermediate join results, i.e. tuples where the bloom filter guarantees that they will not find a join partner in the next iteration and hence do not contribute to the final query result.

6.5.1. RDF Data Store Layout

Since the input datasets have to be sorted by join key to apply a merge join and basic graph patterns operate on a single input RDF graph, it is reasonable to perform a data preprocessing that reduces the sorting effort during query execution. Furthermore, it is a common practice to partition the RDF graph into smaller subsets such that triple pattern matching can be done more efficiently [AMMH07]. In

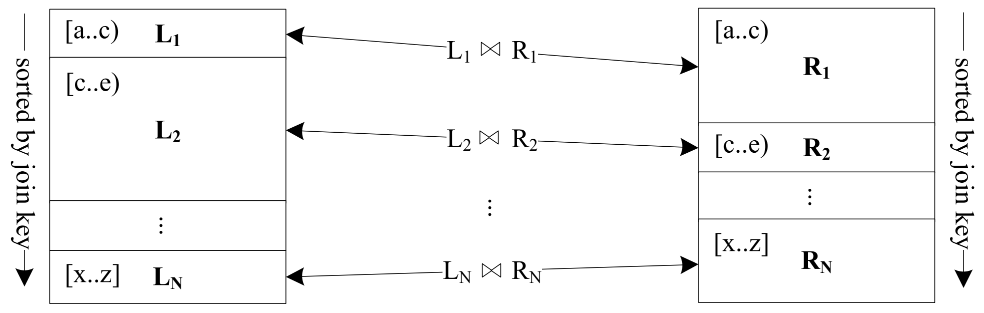


Figure 6.9.: Distributed merge join as a union of N independent subtasks

this section we describe our data store layout for RDF that (1) partitions the data to efficiently support the most common triple pattern types and (2) ensures that we only have to sort the output of the previous join while the second input is always pre-sorted.

Based on the ideas in [AMMH07] we split the data using a vertical partitioning schema where all triples with the same predicate are stored in the same first level partition. We call such a partition a *P-partition*. Similar to [HMM+11], we complement these partitioning schema by also looking at the objects such that all triples with the same predicate and object are also stored in the same second level partition, denoted as *PO-partition* (cf. Figure 6.10). That is, for every triple (s p o), there exists a tuple (s o) in P-partition p and an entry (s) in PO-partition $p|o$. The original RDF graph and all partitions (P and PO) are stored in the distributed filesystem (HDFS). Technically, they are stored using the `SequenceFile` format of Hadoop that allows comparisons on the byte level. We do not consider the combination of predicate and subject since this would result in many small partitions which is undesired in a MapReduce framework.

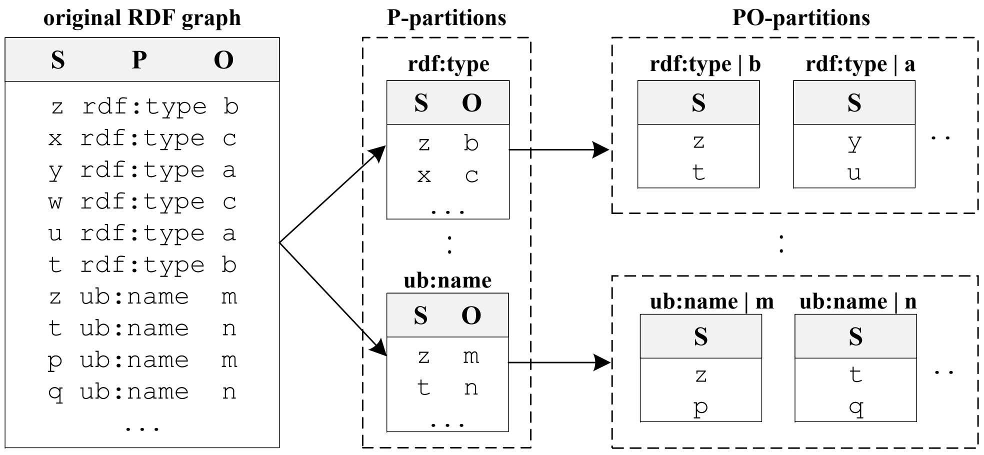


Figure 6.10.: General store layout with P and PO-partitions

To reduce the sorting effort during query execution we perform a pre-sorting of the input dataset and the partitions by all possible attributes, i.e. the overall input dataset (RDF graph) is sorted and stored three times by subject, predicate and object, respectively. P-partitions are sorted and stored twice, once by subject and once by object. PO-partitions can be sorted only by subject.

As already mentioned in Section 6.5, we have to split every sorted partition into N non-overlapping subsets of continuous key ranges. This is achieved during the initial sorting of the partition by assigning continuous and non-overlapping key ranges to the N reducers. The key ranges are derived by a *sampling* of the partition values, i.e. a representative sample of the values is extracted and key ranges are assigned such that there is a uniform and ordered distribution of partition values to the resulting N subsets. To get a uniform distribution, the sample is taken from the subject values if the partition is sorted by subject or from the object values if it is sorted by object.

This layout works fine for joins between triple patterns where the join variable is on the same position, e.g. subject-subject joins (cf. first two triple patterns in Figure 6.14 on page 145). For these joins, both sides must be sorted by the same attribute (e.g. by subject) and therefore they also have the same key ranges. However, when it comes to mixed join variable positions, e.g. the rather common subject-object join (cf. last two triple patterns in Figure 6.14 on page 145), both sides must be sorted by different attributes (one side by subject, the other side by object) and hence the key ranges of both sides will not match in general. But if the key ranges do not match, the join result is not guaranteed to be complete. To overcome this problem, we use two different samplings when sorting a partition. For example, when sorting a P-partition by subject, we do not only pick a sample of the subject values but also a sample of the object values and derive two different key ranges from these samples. We then split the sorted partition in two ways according to subject key ranges and object key ranges, respectively. Hence, a P-partition is actually stored four times in our store layout (cf. Figure 6.11 for P-partition *rdf:type*). To reduce the storage overhead introduced by this layout, we compress every partition using the snappy compression library[3] that is already shipped with Hadoop such that the final stored size is actually smaller than the original uncompressed input (cf. Table 6.2 in Section 6.5.4).

A drawback of this approach is that data distribution can be skewed in general when key ranges are not derived from the sort attribute (e.g. key ranges derived from object sampling while sorting by subject). For joins on the same variable position, e.g. subject-subject joins, this is not a problem as we can use the key ranges that give a uniform distribution for both sides. But in cases of mixed join variable positions, one side will be equally balanced (where the sampling fits to the sorting) while the other side can be more or less unbalanced. This is best illustrated by an example. Consider the BGP ($a\ b\ ?x\ .\ ?x\ c\ d$) with two triple patterns that translates to an object-subject join on variable $?x$. Consequently, for the first triple pattern we use the P-partition b sorted by object (and filtered by subject a) and for

[3] `https://code.google.com/p/snappy/`

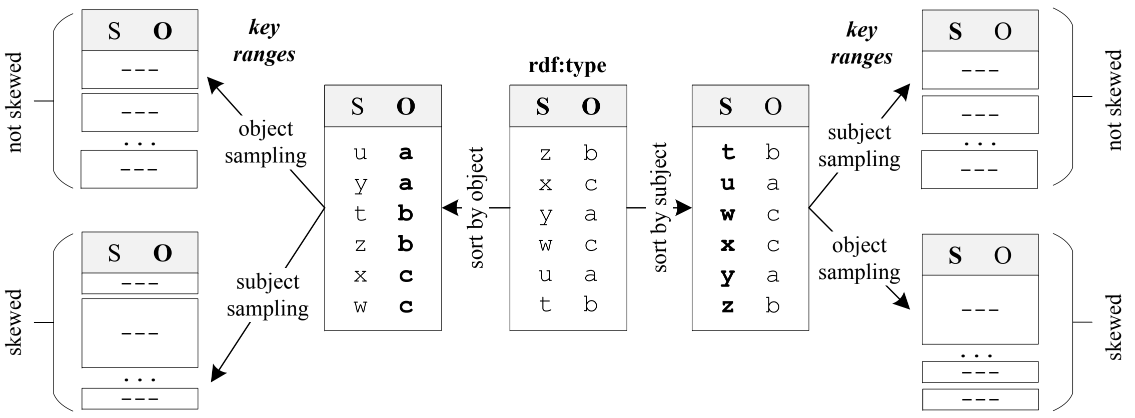


Figure 6.11.: Detailed view of sorting and key ranges for P-partition *rdf:type*

the second pattern we use the PO-partition $c|d$ sorted by subject. But in addition, both sides must have the same key ranges and hence use the same sampling. In this case, we could either use key ranges derived by object or subject sampling for both sides. The former gives a uniform distribution for the partition of the first triple pattern but a potentially skewed one for the second pattern and vice versa.

Handling skewed data in parallel joins has already been researched (e.g. [XKZC08]). Our solution follows a greedy approach, i.e. we always use the sampling that is optimal for the larger of both sides. Though this is not an optimal solution in theory, our experiments confirm that it works fine in practice for most queries. Nonetheless, this is a crucial point for future optimizations of our approach.

6.5.2. Dynamic Bloom Filter Optimization

A *bloom filter* [Blo70] is a space-efficient data structure used to check whether a given element is contained in a set. It consists of a bit vector of size m and k different uniform hash functions that map an element to one of the m bit-vector positions. The filter is constructed by applying the hash functions to each element of the set and setting all corresponding positions to 1. To check whether an element is contained in the set, all k hash functions are applied. This is illustrated in Figure 6.12 for $m = 15$ and $k = 3$. If any of these positions are 0, the element is definitely not in the set, i.e. there are no *false negatives*. However, *false positives* can occur, i.e. a positive membership test does not guarantee that an element is really contained due to possible collisions of the hash functions.

We use bloom filters to remove dangling intermediate results, i.e. results that do not contribute to the final query result. To this end, we build up a bloom filter for each of the N subsets of a partition during the initial sorting (cf. Figure 6.13) and store them on every machine in the cluster by setting the number of replications to N. We can then access these filters locally during join execution to discard those intermediate results where the filter guarantees that they will not find a join

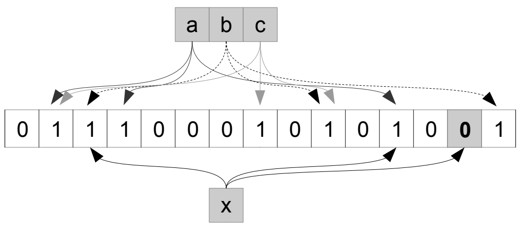


Figure 6.12.: Bloom filter membership test (is x included in the set $\{a, b, c\}$)

partner in the partition of the next join iteration. This is done in the map function for every intermediate join output. The efficiency of this approach strongly relies on the false positive probability but for static bloom filters this can only be estimated if the number of elements to be inserted is known a-priori, so that the bloom filter size can be determined in advance. For that reason, we use dynamic bloom filters [GWCL06] which are essentially a collection of standard bloom filters that increase dynamically with the number of inserted elements while guaranteeing a pre-defined false positive probability.

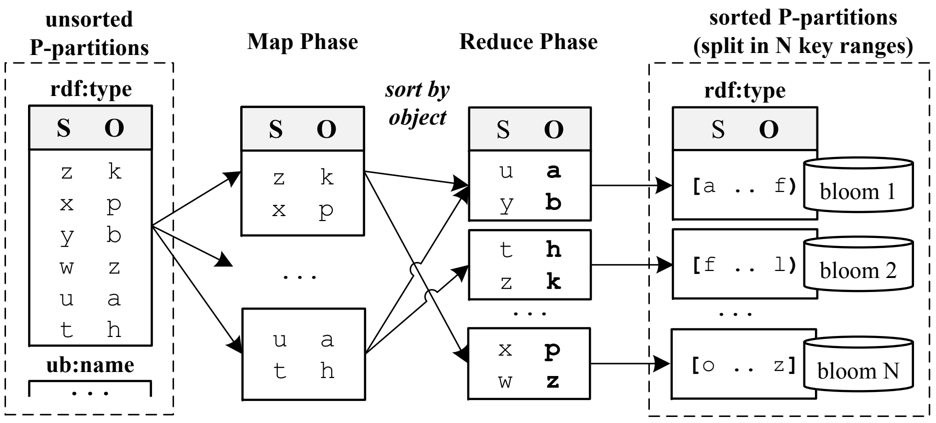


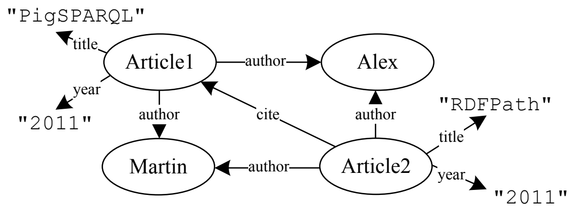
Figure 6.13.: Initial sorting (object) and bloom filter creation for P-parition *rdf:type*

6.5.3. Map-Side Merge Join with MapReduce

After the initial data store generation, the actual BGP query processing can be devided into three subtasks: (1) First, we have to select the input partitions of the data store that match the triple patterns of the BGP. (2) The results for every triple pattern are iteratively joined in the map phase. (3) If there is more than one join iteration the join output has to be post-processed in the reduce phase, i.e. it must be sorted and split into N subsets.

We start with a briefly repetition of the SPARQL terminology, which we use to describe our strategies and algorithms. A SPARQL query defines a graph pattern

P that is matched against an RDF graph G. This is done by replacing the variables in P with elements of G such that the resulting graph is contained in G (pattern matching). The most basic construct in a SPARQL query is a *triple pattern*, i.e. an RDF triple where subject, predicate and object can be variables, e.g. $(?s\ p\ ?o)$. That is, a triple pattern selects a subset of an RDF graph that matches the bound values in the pattern. A set of triple patterns concatenated by AND (.) is then called a *basic graph pattern* (BGP) as illustrated in Figure 6.14. The query asks for all articles published in 2011 that are cited by at least one article. The result of a BGP is defined to be the intersection of all subsets defined by the corresponding triple patterns and can be computed by joining the results of all triple patterns on their shared variables, in this case $?article1$.



SPARQL BGP query

```
SELECT *
WHERE {
  ?article1 title ?title .
  ?article1 year  "2011" .
  ?article2 cite  ?article1
}
```

Figure 6.14.: Example RDF graph and SPARQL BGP query

More formally, let V be the infinite set of query variables and T be the set of valid RDF terms.

Definition 6.1 (Solution Mapping). A (solution) mapping μ is a partial function $\mu : V \rightarrow T$. We call $\mu(?v)$ the variable binding of μ for $?v$. Abusing notation, for a triple pattern p we call $\mu(p)$ the triple that is obtained by substituting the variables in p according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. $\square$

Definition 6.2 (Compatible Mappings). Two mappings μ_1, μ_2 are compatible, $\mu_1 \sim \mu_2$, iff for every variable $?v \in dom(\mu_1) \cap dom(\mu_2)$ it holds that $\mu_1(?v) = \mu_2(?v)$. It follows that mappings with disjoint domains are always compatible and the set-union (merge) of μ_1 and μ_2, $\mu_1 \cup \mu_2$, is also a mapping. $\square$

Definition 6.3 (Joins). The answer to a triple pattern p for an RDF graph G is a list of mappings $\Omega = \{\mu \mid \mu(p) \in G\}$ without a given order. The join of two lists of mappings, $\Omega \bowtie \Omega'$, is defined as the merge of the compatible mappings in Ω and Ω', $\Omega \bowtie \Omega' = \{(\mu_1 \cup \mu_2) \mid \mu_1 \in \Omega, \mu_2 \in \Omega', \mu_1 \sim \mu_2\}$. $\square$

For the following discussion, we consider the example SPARQL BGP from Figure 6.14 which consists of three triple patterns p_1, p_2, p_3:

$$(?art1\ title\ ?title\ .\ ?art1\ year\ 2011\ .\ ?art2\ cite\ ?art1)$$

Let $\Omega^1, \Omega^2, \Omega^3$ denote the mappings for p_1, p_2, p_3, respectively. The query result is then defined as $\Omega^1 \bowtie \Omega^2 \bowtie \Omega^3$. Furthermore, we use the notation Ω^1_i to refer to the

i-th subset of Ω^1 defined by key ranges, i.e. Ω_i^1 and Ω_i^2 have the same key range for the join key. It follows that $\Omega^1 \bowtie \Omega^2 = \bigcup_{1 \leq i \leq N} (\Omega_i^1 \bowtie \Omega_i^2)$. In our example, the join key is $?article1$ for all triple patterns.

Input Selection. For every triple pattern in a BGP we have to (1) identify the corresponding partition, (2) select the appropriate sorting and (3) choose the matching key ranges. The partition selection is derived from the bounded values in a triple pattern. With regard to the above SPARQL example, we would choose P-partitions *title* and *cite* for p_1, p_3, respectively, and PO-partition *year|2011* for p_2. The entries of these partitions directly correspond to $\Omega^1, \Omega^2, \Omega^3$. If there is no partition that directly matches the given pattern, e.g. if the subject is bound, we choose the most appropriate partition and apply a filter in the map phase before feeding the data to the map function. The sorting of the selected partitions is defined by the position of the join variable, i.e. partitions for p_1 and p_2 must be sorted by subject whereas the partition for p_3 must be sorted by object. The selection of the matching key ranges has already been outlined in Section 6.5.1. For the first join between Ω^1 and Ω^2 the choice is clear as it is a subject-subject join, hence we can use key ranges derived by subject sampling which means that both sides are equally balanced. However, the second join between the result of $(\Omega^1 \bowtie \Omega^2)$ and Ω^3 is a subject-object join where we have to decide whether we use key ranges derived by subject or object sampling. Without loss of generality, we assume $|(\Omega^1 \bowtie \Omega^2)| < |\Omega^3|$. Thus, we choose key ranges derived by object sampling such that the partition splits for p_3 are equally balanced.

2-Way Merge Join. After the input selection, the query result is computed by a sequence of cascaded 2-way merge joins as illustrated in Figure 6.15. The input partitions (recall that the entries correspond to the solution mappings for the triple patterns) are pre-sorted by the join key and split into N subsets with matching key ranges, e.g. $\Omega^1 = \bigcup_{1 \leq i \leq N} \Omega_i^1$. Within the map phase, every machine in the cluster computes the partial join between two subsets with matching key ranges, i.e. $(\Omega_i^1 \bowtie \Omega_i^2)$.

Due to the locality principle of MapReduce, one subset is always read locally. However, we cannot guarantee that both subsets with the same key range are stored on the same machine as data placement is done by the distributed filesystem where we store the partitions (for Hadoop this is HDFS). In general, the larger subset is chosen to be processed locally whereas the smaller subset has to be transferred over the network at the beginning of the map phase. This is automatically handled by Hadoop. Thus, in every join iteration only the smaller subset (which is typically the output of the previous join iteration) is transferred, in contrast to reduce-side joins where typically both sides must be transferred. However, it is a topic of our future developments to improve the co-locality of matching key ranges such that both sides can be read locally.

The merge-join algorithm for the map function is illustrated in Algorithm 6.5.1. On every machine the map function is invoked with a composite key consisting of the current join key and the join key of the next iteration, if any. The value is also a composite value consisting of the corresponding subsets of solution mappings (these are essentially the entries of the input partitions) and the bloom filter of the next join partition. Regarding our example, the map invocation for the i-th mapper in the first iteration would be:

$$\texttt{map}(\{?article1, ?article1\}, \{\Omega_i^1, \Omega_i^2, bloom(\Omega_i^3)\})$$

The current join key is $article1$ and this is also the join key of the next iteration. The map function computes $\Omega_i^1 \bowtie \Omega_i^2$ and discards those mappings where the bloom filter membership test fails, i.e. for every merge of compatible mappings it is checked whether the value of the next join key is contained in the bloom filter of the next join partition. If this test fails, it is guaranteed that there is no join partner in the next iteration and the mapping can be discarded.

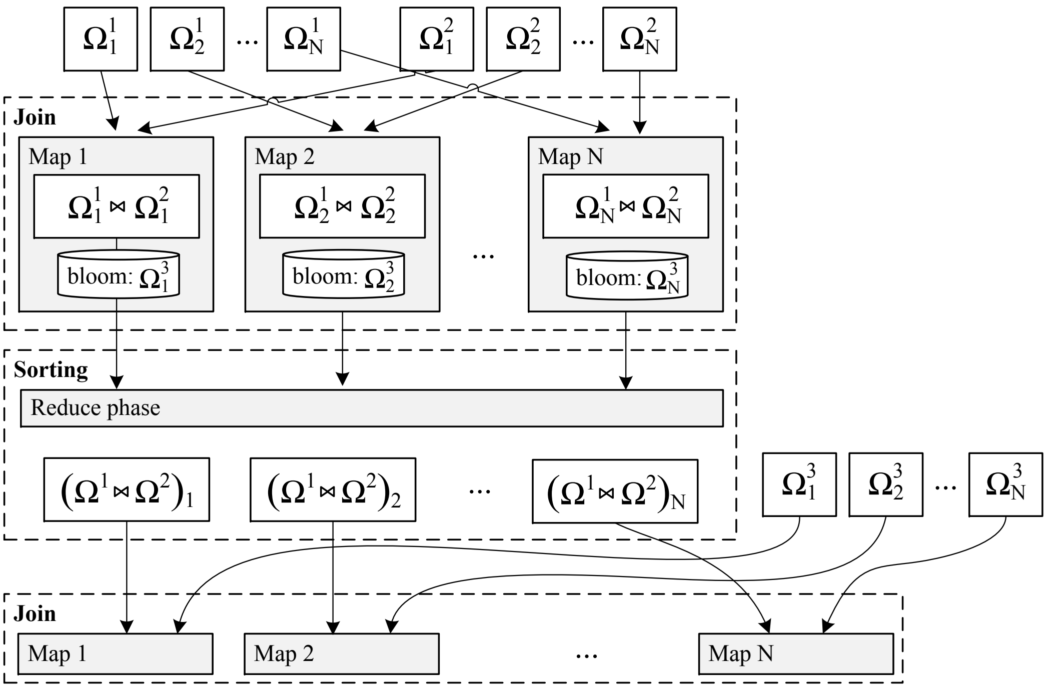


Figure 6.15.: Cascaded 2-way map-side merge join

For a cascaded join sequence we use the reduce phase to sort the join output, $\Omega^1 \bowtie \Omega^2$, according to the join key of the next iteration and to split it into N subsets such that the key ranges match with the key ranges of the next join partition, $\Omega^1 \bowtie \Omega^2 = \bigcup_{1 \leq i \leq N}(\Omega^1 \bowtie \Omega^2)_i$ (cf. lower half of Figure 6.15). In our example, we would use the same key ranges that are used for the input partition matching p_3. Therefore, we also store the key ranges of a partition such that we can reuse them for intermediate join results. As the sorting and assignment of values to reducers

Algorithm 6.5.1 : 2-way merge join - **map**(key, value)

```
input  : $key = \{k', k''\}$, $value = \{\Omega_i, \Omega'_i, bloom(\Omega''_i)\}$
         // $k'$ is the current join key, $k''$ is the join key of the next join
         // $\Omega_i$ and $\Omega'_i$ are sorted by join key and have the same key range
         // $bloom(\Omega''_i)$ is the bloom filter of the next join subset $\Omega''_i$
output : $\Omega_i \bowtie \Omega'_i$
1  $l \leftarrow 1, r \leftarrow 1$
2  while $l \leq |\Omega_i|$ and $r \leq |\Omega'_i|$ do
3      $\mu_1 \leftarrow \Omega_i[l], \mu_2 \leftarrow \Omega'_i[r]$
4      if $\mu_1(k') = \mu_2(k')$ then // $\mu_1 \sim \mu_2$
5          $r' \leftarrow r$ // temporary pointer to iterate over all compatible $\mu_2 \in \Omega'_i$
6          while $\mu_1 \sim \mu_2$ do
7              // emit merge of $\mu_1$, $\mu_2$ if it passes the bloom filter
8              // membership test for the next join key $k''$
9              if $(\mu_1 \cup \mu_2)(k'') \in bloom(\Omega''_i)$ then emit$\{(\mu_1 \cup \mu_2)(k''), (\mu_1 \cup \mu_2)\}$
10             $r' \leftarrow r' + 1, \mu_2 \leftarrow \Omega'_i[r']$
11         end
12         $l \leftarrow l + 1$
13     else $\mu_1 \nsim \mu_2$
14         if $\mu_1(k') < \mu_2(k')$ then $l \leftarrow l + 1$ else $r \leftarrow r + 1$
15     end
16 end
```

(partitioning) is done when shuffling data from mappers to reducers, the reduce function is only an identity function that just stores its input to HDFS. In the following join iteration, the i-th mapper then computes $(\Omega^1 \bowtie \Omega^2)_i \bowtie \Omega^3_i$ and so on.

N-Way Merge Join. If the join key is the same in a sequence of n 2-way merge joins, we can also compute the result with a single n-way merge join, thus saving $n-1$ MapReduce iterations. In our example, the join key for both 2-way join iterations is $?article1$, so we could also use a single 3-way join instead. The basic principle is the same as for the 2-way join but instead of two subsets each machine joins n subsets in a single map phase, i.e. $(\Omega^1_i \bowtie \cdots \bowtie \Omega^n_i)$. Just like for the 2-way join, it cannot be guaranteed that all n subsets with the same key range are stored on the same machine, hence the missing subsets must be transferred to the corresponding machine at the beginning of the map phase. Input partition selection and also post-processing in the reduce phase is the same as for the 2-way join. The 2-way merge join algorithm in Algorithm 6.5.1 can be easily extended for n-way merge joins using n interleaved linear scans instead of two. However, a disadvantage of the n-way join compared to a sequence of 2-way joins is that we do not benefit from bloom filters for intermediate results. Hence, we can only apply bloom filters on the results of the n-way join (if another join iteration follows). Yet, our experiments demonstrate that, in general, the saving of MapReduce iterations has a greater impact on query performance than discarding dangling intermediate results.

6.5.4. Experiments

The experiments were performed on a cluster of ten machines using the Hadoop distribution of Cloudera in version 4.2.1[4]. We used the well-known Lehigh University Benchmark (LUBM) [GPH05] and generated datasets from 500 up to 3000 universities where we pre-computed the transitive closure using the WebPIE inference engine for Hadoop [UKM+12]. This benchmark is a well-suited choice when considering join processing in a Semantic Web scenario since the queries of the benchmark can easily be formulated as SPARQL basic graph patterns. The store generation runtimes and dataset sizes are listed in Table 6.2. We can observe that the actual store size is even smaller than the size of the original RDF graph. This is achieved by replacing prefixes and applying snappy compression which reduced the original dataset size by up to 92%.

Table 6.2.: Generating RDF data store

LUBM	**500**	**1000**	**1500**	**2000**	**2500**	**3000**
triples (million)	105	210	315	420	525	630
input size (GB)	17.0	34.1	51.3	68.5	85.7	102.9
overall store size (GB)	13.6	27.3	41.0	54.8	68.6	82.3
store generation (minutes)	68	111	154	193	241	279

We compared our merge join approach with three systems based on MapReduce. With (1) *PigSPARQL* [SPL11, SPHL13], there exists a reduce-side join based SPARQL 1.0 engine built on top of *Apache Pig*. The crucial point for this choice was the comparable reduce-side join implementation of Pig [GNC+09] to RDFPath, which resembles the performance of RDFPath in comparison to other join techniques. To that end, we consider this as a baseline competitor, which we want to improve as discussed earlier. (1) *HadoopRDF* [HMM+11] is an advanced SPARQL engine that splits the original RDF graph according to predicates and objects and utilizes a cost-based query execution plan for reduce-side joins. (2) *MAPSIN* [SPZD+12] is a map-side index nested loop join implementation based on HBase. It processes joins within the map phase and exploits n-way joins by a sophisticated storage schema that significantly reduces the amount of HBase lookups.

Figure 6.16 illustrates the scaling properties of our merge join approach. We can observe a linear scaling of query runtimes where tripling the dataset size does not

[4]The detailed cluster description is shown in Chapter 8 on page 195

even take twice the time for most queries. A comparison of execution times (wall time) to other approaches is summarized in Table 6.3. MAPSIN lacks the support for LUBM queries 2, 9 and 10 whereas HadoopRDF runs out of storage space while creating its internal storage schema for datasets larger than LUBM 2000[5]. We considered a time-out of one hour, denoted by T, if a query fails to complete in time.

Queries 6 and 14 are simple and require no join at all. Nevertheless, our approach outperforms the other systems significantly as the whole processing is done locally on all cluster machines without any reduce phase. One may expect MAPSIN to be the fastest for such queries, since a single table lookup could provide the final result. However, such a request would push the result to only one machine violating the scaling properties. Therefore, MAPSIN processes such single pattern queries by a distributed table scan which is executed on each machine preserving scalability.

Queries 1, 3, 5 and 13 contain only one join. Our approach processes these queries within a single map phase as no additional processing for a subsequent join is required. Overall, the merge join performs best for these kind of queries.

Query 2 is rather complex (compared to other LUBM queries) as it exhibits a triangular pattern structure. Moreover, it contains a costly and unselective subject-object and object-subject join that points out a weakness of our approach. As both join partitions must use the same key ranges, one of the input partitions is fairly unbalanced (cf. Section 6.5.1) which increases the costs for join processing. Thus, it is a point of future work to develop more sophisticated sampling strategies for key ranges to improve data distribution for such cases.

Queries 7, 8 and 9 demonstrate the base case where several joins have to be processed sequentially. For these queries, the reduce phase is required to sort the join output according to the join key of the next iteration, whereas bloom filters are used to remove dangling intermediate results. For queries 7 and 8 we can observe a quite competitive performance of our approach compared to both reduce-side join based systems. Indeed, such selective queries are the core of MAPSIN, where it benefits from its index structure accessing only those triples that are relevant for the query answer. Nonetheless, query runtimes of MAPSIN are close to the runtimes of the merge join approach.

Query 4 is a star pattern query which makes it a good candidate for n-way joins. Figure 6.17 illustrates a comparison between 2-way and n-way execution for Merge Join, MAPSIN and PigSPARQL as these systems support n-way joins. In all cases, n-way joins clearly outperform a sequential execution of 2-way joins, while the benefit for our merge join approach is less than for the others. Reducing the amount of MapReduce cycles comes at the cost of more data that has to be processed at once. As we cannot guarantee that all n matching subsets reside on the same machine,

[5]We contacted the authors since neither a documentation nor an “out-of-the-box" running system was available, unfortunately we didn’t get any support.

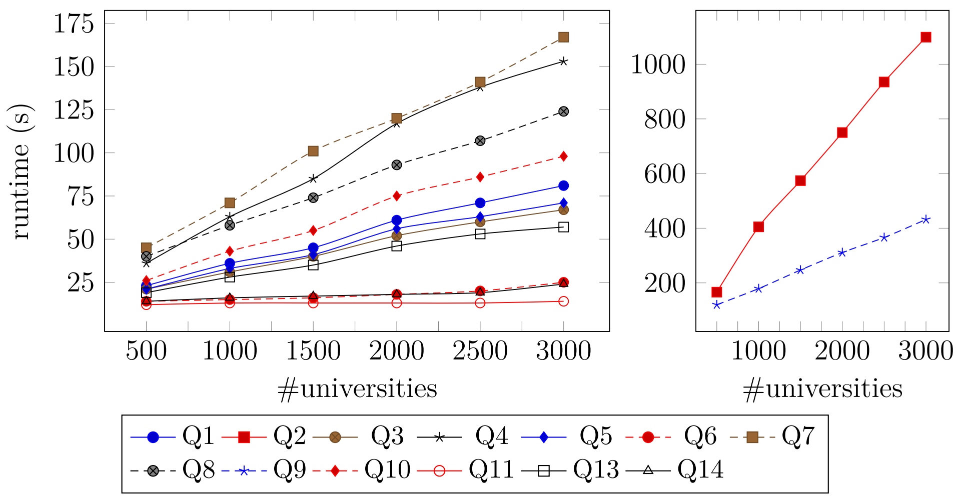


Figure 6.16.: Runtimes for all LUBM queries (Merge Join).

more data has to be accessed remotely. This is underpinned by Figure 6.18 that shows the amount of data accessed locally and retrieved remotely within a map-side merge join. However, the difference is much less than expected. Processing query 4 with one 5-way merge join compared with several 2-way merge joins increases the amount of data retrieved remotely by only 32% while decreasing the amount of data accessed locally by 34%. This can be explained due to the fact that data is stored with a replication factor of three which increases the chances that matching subsets reside on the same machine. Except for n-way joins, the amount of data accessed locally within a merge join is always higher than the amount of data retrieved remotely, which is an expected behavior since we choose the larger dataset to be processed locally. Nevertheless, since data locality is a crucial point for distributed systems, improving these values, e.g. by a refinement of the data placement strategy wrt. matching subsets, is a worthwhile point for future optimizations.

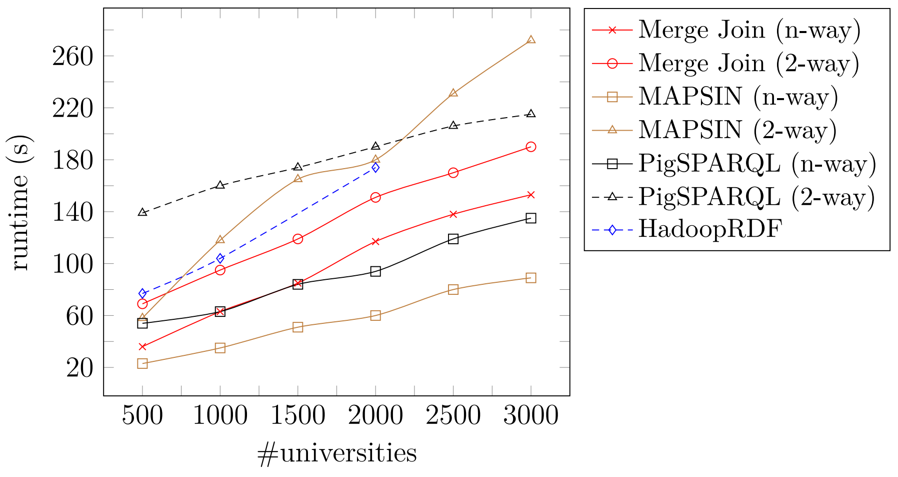


Figure 6.17.: Comparison of n-way optimizations for LUBM Query 4.

Table 6.3.: Query execution times for Merge Join, MAPSIN, PigSPARQL and HadoopRDF in seconds. T: time-out

	LUBM query	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**13**	**14**	**relative perf.**
500	**Merge Join**	23	166	21	36	21	14	45	40	120	26	12	19	14	100%
	MAPSIN	22	—	21	23	22	14	46	49	—	—	21	24	18	90%
	PigSPARQL	32	157	33	54	32	22	110	110	183	32	33	32	28	61%
	HadoopRDF	41	111	43	77	52	41	112	211	171	46	116	177	35	45%
1000	**Merge Join**	36	405	31	63	33	15	71	58	179	43	13	28	16	100%
	MAPSIN	32	—	30	35	33	45	60	60	—	—	32	42	42	85%
	PigSPARQL	42	199	42	63	43	28	135	141	233	42	33	43	27	72%
	HadoopRDF	48	136	58	104	59	51	148	298	208	55	1585	186	54	52%
1500	**Merge Join**	45	574	40	85	41	16	101	74	247	55	13	35	17	100%
	MAPSIN	49	—	41	51	43	58	66	75	—	—	44	70	70	82%
	PigSPARQL	53	233	53	84	48	32	176	166	279	53	38	48	32	75%
	HadoopRDF	61	174	71	149	74	54	189	389	233	63	T	193	59	58%
2000	**Merge Join**	61	750	52	117	56	18	120	93	311	75	13	46	18	100%
	MAPSIN	52	—	47	60	52	67	93	92	—	—	51	81	78	85%
	PigSPARQL	64	283	63	94	58	38	216	212	329	63	49	53	42	78%
	HadoopRDF	77	196	80	174	86	58	215	457	257	68	T	206	62	63%
2500	**Merge Join**	71	935	60	138	63	20	141	107	366	86	13	53	19	100%
	MAPSIN	69	—	58	80	65	82	110	105	—	—	65	102	87	81%
	PigSPARQL	73	335	68	119	64	48	252	247	399	74	58	64	47	79%
3000	**Merge Join**	81	1099	67	153	71	25	167	124	432	98	14	57	24	100%
	MAPSIN	81	—	70	89	78	98	120	119	—	—	74	125	105	80%
	PigSPARQL	83	385	78	135	74	53	281	287	460	83	69	68	53	80%

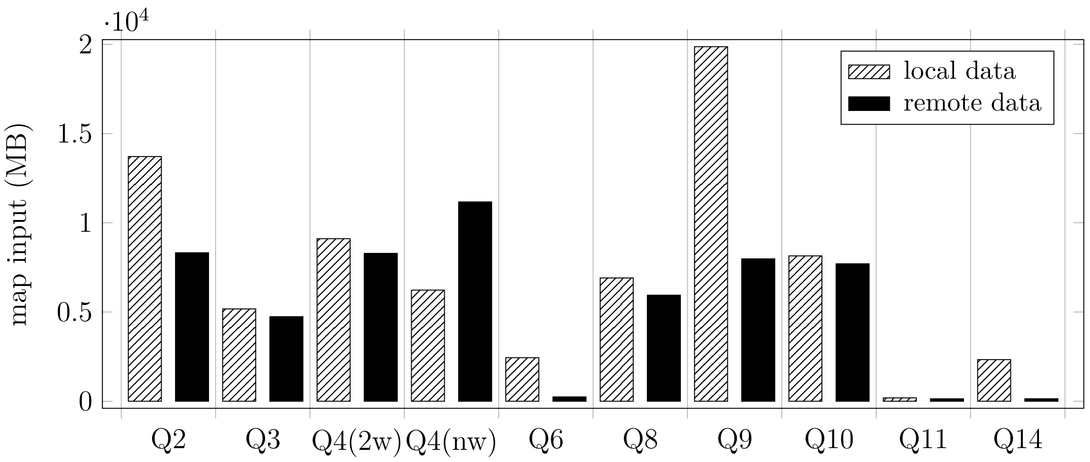


Figure 6.18.: Comparison of local and remote data access within a map-side merge join.

The average performance benefit of our Merge Join compared to the other systems is between 15% and 48%. In order to compare the performance of the different systems we computed for each query the relative difference of execution time to the respective best case. Then, for each system the average of these relative differences over all queries is computed, whereas missing measure points are replaced with a weak penalty value. The penalty is based on the average of all systems that perform worse than the best execution time. Finally, we computed the relative performance distance of other systems to our approach (cf. "relative perf." in Table 6.3). For example, if we refer to LUBM 1000, we can derive that PigSPARQL (72%) is in comparison with Merge Join (100%) on average 28% slower.

Overall, the experiments showed that our map-side merge join approach exhibits in most cases competitive runtimes with a performance benefit of 15% to 48% on average over all queries. It works best for single join queries but also performs well for sequences of joins. However, unselective subject-object or object-subject joins turned out to be a weak point, especially if one join input side is fairly unbalanced. But even for those queries, the differences to the fastest query execution times are still acceptable while showing an excellent scaling behavior at all time. Moreover, our store layout enables retrieving pattern queries even faster than the index-based query execution of MAPSIN. Taken into account that improving data locality by adopting more suitable data placement strategies for Hadoop and preventing unbalanced partitions by more sophisticated partitioning strategies will further improve query execution times, we can conclude that map-side merge joins are well suited for processing different kinds of join patterns with MapReduce.

6.5.5. Related Work

In terms of mere query performance, *RDF-3X* [NW08] has established itself as the state-of-the-art "benchmark" engine for single place machines. However, its performance has been shown to degrade for queries with unbound objects and low selectivity factors [HMM+11]. Furthermore, with the ever increasing amount of available RDF data, single machine solutions for query processing become more and more challenging [HAR11]. Thus, a number of systems that focus on distributed execution of SPARQL queries have been proposed in recent years (e.g. [Erl12, HLS09, HUHD07]). Since each of these implementations require some dedicated infrastructure and management, there are no synergy effects by reusing already-deployed frameworks. Our research is driven by the idea of reusing existing infrastructures for "Big Data" scenarios. Consequently, we have ensured that no changes to the underlying Hadoop framework are required to run implementation. This way, existing Hadoop clusters or cloud services (e.g. Amazon EC2) can be used without any changes.

The efficient computation of joins is the main driver for the performance of SPARQL BGP evaluation, and thus we have focused on join processing in MapReduce in this paper. This topic has already been studied considering various aspects and application fields [AU11, JTC11, LLC+11, OR11, YDHJ07]. In [OR11] the authors discussed how to process arbitrary joins (θ joins) using MapReduce, whereas [AU11] focuses on optimizing n-way joins. θ joins are not required for the evaluation of SPARQL BGPs, and they are not supported by our solution. The execution of n-way joins is a generalization of our 2-way join, where instead of two all n pre-sorted input partitions are processed in a single map phase. *Map-Reduce-Merge* [YDHJ07] describes a modified MapReduce workflow by adding a merge phase after the reduce phase, whereas *Map-Join-Reduce* [JTC11] proposes a join phase in between the map and reduce phase. Both techniques attempt to improve the support for joins in MapReduce but require profound modifications to the MapReduce framework. In [DQRJ+10] the authors present non-invasive index and join techniques for SQL processing in MapReduce that also reduce the amount of shuffled data at the cost of an additional co-partitioning and indexing phase at load time. However, the schema and workload is assumed to be known in advance which is typically feasible for relational data but does not hold for RDF in general. *HadoopDB* [ABPA+09] is a hybrid of MapReduce and DBMS where MapReduce is the communication layer above multiple single node DBMS. The authors in [HAR11] adopt this hybrid approach for the Semantic Web using RDF-3X. Initially, they partition the graph on a single machine in a loading phase. We also initially store the dataset wrt. different sort orders but we employ the MapReduce framework to partition the dataset at loading time. Common to both approaches is that data has to be reloaded in case of updates, while we do not require the installation of additonal engines at each cluster node. *HadoopRDF* [HMM+11] is a MapReduce based RDF system that stores data directly in HDFS and does also not require any changes to the Hadoop framework. It is able to re-balance automatically when the cluster size changes but join processing is also done in the reduce phase. As already mentioned, our join processing is done in the map phase and additionally we reduce the amount of data sent over the network by pro-actively filtering dangling tuples using bloom filters.

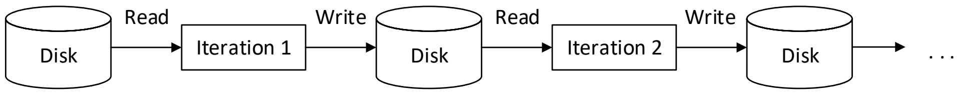
6.5.6. Conclusion

In the area of “Big Data” applications, MapReduce has become a state-of-the-art technology for large-scale data processing. On the other hand, the advent of the Semantic Web promotes the growing adoption of RDF and its query languages, raising attention for distributed query processing in current research. As both RDFPath traversing steps and SPARQL basic graph patterns, translating to the computation of joins on the operator level, we based this section on a basic graph pattern, which enables providing more comprehensive comparisons with other commonly used approaches. Generally speaking, efficient distributed join techniques for RDF are of particular interest for all kinds of query languages. The fixed ternary structure of RDF makes it a well-suited candidate for sort-merge joins as presorting the data is affordable. In this section we presented an adaptation of sort-merge joins with MapReduce which supports both 2-way and n-way joins. The actual join computation is completely done in the map phase, complemented by bloom filters to discard dangling intermediate results, while the reduce phase is used to post-process the join output for subsequent join iterations. Our experiments with the LUBM bechmark demonstrated an average performance benefit between 15% and 48% of our approach compared to other MapReduce-based systems while scaling smoothly with the dataset size. We can conclude that applying *Map-Side Merge Joins* for navigational queries proves to be a promising approach, though further research on improving the particularly interesting subsequent join iterations is inevitable. For future work, there are multiple refinements that one can consider. First of all, the data placement strategy to further optimize data locality, but also new techniques to handle skewed data in parallel joins are of particular interest.

7. Scalable RDF Querying on Distributed In-Memory Processing Engines

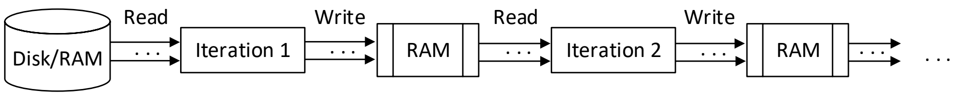
Contents

While the steady growth of semantically-annotated data, with its high degree of diversity in both structure and vocabulary, justifies expressive RDF query languages such as TriAL-QL and RDFPath, it also raises the need for systems that cover a wide range of querying features and scale with the data size. In Chapter 6, we have introduced our RDFPath MapReduce Processor, which has already demonstrated very good scalability and is shown to be well suited for data-intensive or complex analytical tasks. Such ETL-like workloads are typically processed *offline*, with expected runtimes being in the order of minutes to hours. However, in the case of more selective queries requiring only a small subset of the data, we expect for both of our languages to get an answer in *interactive* time, i.e in the order of seconds to a few minutes. MapReduce is not intended to fulfill those requirements. The major bottlenecks which hamper its usage are hereby very I/O-heavy, disk-based operations between map and reduce phases as illustrated with a typical MapReduce workflow, where an iteration corresponds to a map or reduce phase:



Furthermore, each iteration is a strictly *staged* process which means that the next iteration cannot be started before the previous one finishes writing its result to disk. Though considerable research has been done in this area (cf. Chapter 6.5), for instance, our optimized Map-Side Merge Joins, the obtained runtimes were far from what one considers to be *interactive* time raising the need for new distributed execution frameworks.

With the continuously-decreasing prices of main memory in recent years, we can observe the emergence of novel *in-memory* data processing systems for Hadoop such as *Stinger for Hive*, *Impala*, *Spark*, *Presto*, and *Phoenix* which start to pave the way for scalable and interactive querying on large-scale data. What they all have in common is that they store their data in HDFS, but not using MapReduce as the underlying processing framework. Yet only initial input data is in HDFS, whereas intermediate results are shared in main memory with a workflow as illustrated here:



In cases where there is not sufficient main memory available to process a certain operation, thus, the input data does not fit into main memory or the intermediate result is still too large, streaming from and spilling to disk can be used in most of these systems. Despite being a costly operation, this fall-back strategy maintains the compatibility with data-intensive tasks, enabling these novel systems to capture a wide range of workloads, ranging from exploratory ad-hoc style queries to ETL-like workloads, which were previously the strength of MapReduce. Further, most of the systems break down with a strictly-staged execution flow allowing to provide first results much faster, without the need to wait till the last ones are computed.

Following this trend, we introduce in this chapter the TriAL-QL Engine and the RDFPath Engine, two distributed query processors with support for both data-intensive, analytical tasks but also interactive querying. Each of them is implemented on top of *Impala*, a massive parallel SQL query engine on Hadoop and *Spark*, a fast general execution framework for large-scale data processing, while sharing *one* unified data store in HDFS. The results of both engines are then accessible to other RDF management systems built on top of Hadoop which make use of the common RDF data pool in HDFS. Beyond the engineering work of developing such systems, we designed multiple evaluation strategies and optimizations for, e.g. recursive traversals and query pattern that ask for the connectivity between resources in the graph, which will be introduced in the following sections. Furthermore, different approaches for composing (nested) queries are also investigated where, for instance, materializing some intermediate results is used to improve the overall performance. All those aspects will be examined with experiments on generated social networks with up to 1.8 billion triples. A comprehensive comparison with other competitors will follow in Chapter 8, where a set of predefined benchmark queries is used to study mainly performance aspects.

Most of the results from this Chapter were published in [PSL15a] and in [PSL15b]. Some of the contributions, especially the RDFPath Engine, are not yet published at the point of writing this chapter and are therefore presented here for the first time. Overall, we can summarize the contributions of this chapter as follows:

- After a short introduction into Impala, Parquet and Spark in Section 7.1, we propose in Section 7.2 the distributed TriAL-QL Engine based on a hybrid infrastructure that uses Impala and Spark as its underlying execution framework.
- We discuss the data storage layout designed on Parquet and its usage as a unified data pool for Spark and Impala in Section 7.2.1. The translation of TriAL-QL into a sequence of SQL queries is explained in Section 7.2.2. We will provide two algorithms that are used for the evaluation of recursive expressions. At the end of this chapter, an optimization for the connectivity pattern is proposed. Section 7.2.3 explains how translated TriAL-QL queries are executed and discusses further optimization strategies with regard to query composition and the materialization of intermediate results.
- The performance of our implemented evaluation strategies is investigated by some experiments in Section 7.3.
- The RDFPath Engine, which is also built on top of both, Impala and Spark while using a shared data store is introduced in Section 7.4.
- Section 7.4.1 proposes two implemented data partitioning strategies, namely the Vertical Partitioning and an adopted version of the Extended Vertical Partitioning. We will show how those strategies can be efficiently used to reduce the input size of a query. The main algorithms for the evaluation of RDFPath expressions, which are described in Section 7.4.2, will be based on those presented for the TriAL-QL Engine. Multiple execution strategies that investigate the composition of queries in more detail are presented in Section 7.4.3.
- We conclude the work on RDFPath Engine with some experiments that investigate the properties of the proposed partitioning approaches and execution strategies in Section 7.5.

7.1. Foundations

Impala & Parquet. *Impala* [KBB+15] is an open-source MPP (Massively Parallel Processing) SQL query engine for Hadoop inspired by Google *Dremel* [MGL+10] and developed by *Cloudera*. It is seamlessly integrated into the Hadoop ecosystem, i.e. it can run queries directly on data stored in HDFS without requiring any data movement or transformation. Moreover, it is designed from the ground up to be compatible with Apache Hive [TSJ+09], the standard SQL warehouse for Hadoop. For this purpose, it also uses the Hive Metastore to store table definitions etc. so that Impala can query tables created with Hive and vice versa. The main difference to Hive is that Impala does not use MapReduce as the underlying execution layer but instead deploys its own distributed query engine. The architecture of Impala and its integration into Hadoop is illustrated in Fig. 7.1, with our execution engine for TriAL-QL being an application on top of it. The Impala daemon is collocated with every HDFS DataNode such that data can be accessed locally. One arbitrary node acts as the coordinator for a given query, distributes the workload among all nodes and receives the partial results to construct the final output.

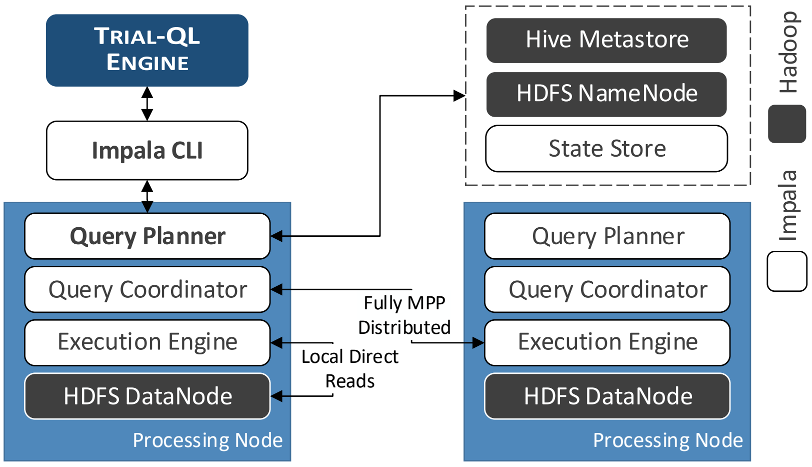


Figure 7.1.: Impala architecture and integration into the Hadoop stack

Parquet [Apa] is a general purpose columnar storage format for Hadoop inspired by Google *Protocol Buffers* [MGL+10] and primarily developed by *Twitter* and *Cloudera*. Though not developed solely for Impala, it is the storage format of choice regarding performance and efficiency for Impala. A big advantage of a columnar format compared to a row-oriented format is that all values of a column are stored consecutively on disk allowing better compression and encoding as all data in a column is of the same type. Parquet comes with built-in support for bit packing, run-length, and dictionary encoding as well as compression algorithms like snappy.

Spark. *Spark* [ZCD+12a] is a *general-purpose* in-memory cluster computing framework for managing diverse big data tasks in one comprehensive solutions. It supports writing applications in several languages including Java, Scale, Python or R and is equipped with a rich stack of high-level tools. Most important to note are *Spark SQL* for structured data processing, *MLlib* for machine learning, *GraphX* for graph processing, and *Spark Streaming*. With such a high variety of tools it is meant to bridge the gaps between using individual systems for different big data tasks in *one* single framework, where data can be seamlessly exchanged between different kind of tasks (e.g. from querying with SQL to machine learning algorithms) without data movement or duplication. An overview of SPARK with TRIAL-QL being an application on top of it is illustrated in Fig. 7.2.

Its central data structure is the *Resilient Distributed Dataset* (RDD) [ZCD+12b] which is a fault-tolerant collection of elements that can be operated on in parallel. Spark attempts to keep an RDD in main-memory and partitions it across all machines in the cluster. However, in cases where data does not fit in memory, Spark's operators allow spilling data to the disk. Conceptually, Spark adopts a *data-parallel* computation model that builds upon a record-centric view of data, similar to *MapReduce* and *Apache Tez*. A job is modeled as a directed acyclic graph (DAG) of tasks where each task runs on a horizontal partition of the data.

Spark SQL [AXL+15] is the relational interface to deal with structured data. Tables are represented by so-called *DataFrames* which are based on RDDs and also kept in main-memory. A DataFrame essentially models a distributed collection of rows that share the same schema. The built-in query optimizer called *Catalyst*, combined with dictionary and run-length encoding, facilitates the fast execution of relational operators.

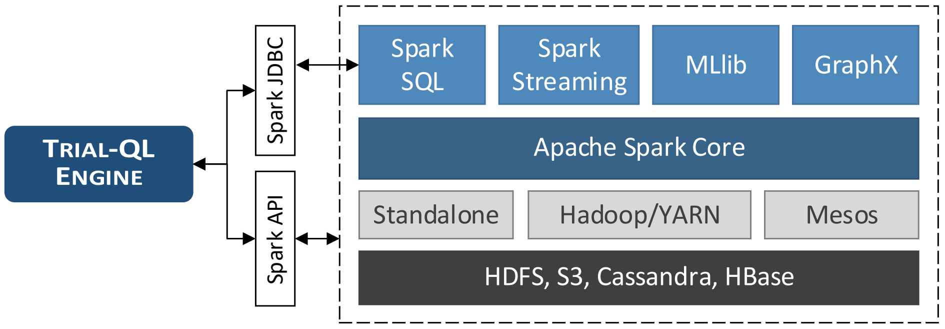


Figure 7.2.: Spark overview and integration

7.2. Distributed Execution Engine for TRIAL-QL

The TriAL-QL Engine is a distributed processor of our previously-discussed TriAL-QL query language with support for all querying expressions that has been introduced in Chapter 5. It is built on top of Hadoop, where we make use of the current momentum on scalable SQL-on-Hadoop frameworks. An essential advantage of such frameworks is the possibility to use of SQL as an adequate intermediate layer. This is particularly beneficial for processing query languages such as TriAL-QL which are based on relational algebra, due to an intuitive mapping between both. Furthermore, we can (1) benefit from an inter-compatibility between different SQL-on-Hadoop solutions, (2) be independent from future Hadoop changes, and (3) take advantage of the continuously optimized Hadoop stack. However, besides the differences in syntax and expressiveness of the supported SQL dialect, there are crucial differences in how the respective SQL-on-Hadoop solutions process a given SQL query, revealing different characteristics for various query types. In line with this, we examine the applicability of a scalable processor for TriAL-QL on top of Hadoop using *Impala*, *Spark* and *Hive* and compare its properties. Therefore, we implemented our engine for TriAL-QL with support for all three systems. Please note that, although not included in the latest TriAL-QL Engine, we also consider Hive in the followed discussion and experiments[1]. The architecture of our implementation can be conceptually structured into three main components:

- **RDF Store:** The basis for distributed querying is an efficient yet unified data pool which is supported by all three of our systems: Spark, Hive and Impala. It maintains input RDF data and in the case of a disk-based execution also intermediate results and the final results of queries, which can be (if specified within a query) reused as input in a later query. A discussion on different RDF data storage strategies and the data layout our engine is based on is given in Section 7.2.1.
- **Query Compiler:** A Query Compiler composes all components required to translates a given TriAL-QL query into the respective SQL dialect of the desired SQL-on-Hadoop system. Furthermore, it is also the place where certain query optimizations are applied. Most notable are two interesting query patterns, for which we describe optimized evaluation strategies, namely (1) the recursions captured by the left and right Kleene Closure of TriAL* and (2) the connectivity between two resources. Section 7.2.2 introduces both patterns and provides algorithms for evaluating them efficiently.
- **Query Processor:** The actual execution of SQL queries is done by the Query Processor, where multiple execution strategies are suggested which differ, for

[1]The support for Hive was dropped for the final version of TriAL-QL Engine. However, using SQL as intermediate language along with Parquet files stored in HDFS, which are fully compatible with Hive, allowed us to execute compiled SQL queries on Hive which formed the basis for a few experiments shown later.

instance, in how queries are composed and when intermediate results become materialized to disk. Another important task that this component is taking care of are termination conditions. Based on which algorithms are chosen by the Query Compiler, different termination constraints need to be checked. Further, due to the differences in the support of recursions in the respective SQL-on-Hadoop systems, an individual query processor exists for each supported system. Section 7.2.3 describes all execution strategies and briefly introduced the differences in the respective termination conditions.

The actual architecture of our TriAL-QL Engine, illustrated in Figure 7.3, reveals a much more granular view on all three components, including the technical integration with Impala, Spark, and HDFS. Nonetheless, for simplicity we will continue to distinguish between the three previously-mentioned components, which will be discussed in more detail in the following.

7.2.1. RDF Store

Typically, RDF triplestores with DBMS back-ends represent an RDF dataset in a so-called *triples table* with three columns, containing one row for each RDF statement, i.e. $triples(sub, pred, obj)$. Query evaluation then essentially boils down to a series of self-joins on this table. Therefore, it is often accompanied by several indexes over some or all (six) triple permutations, e.g. based on B^+-trees, for query speedup. However, these kinds of indexes are not suitable and hard to maintain in a distributed-computing environment like Hadoop. In [AMMH07] the authors propose a *vertical partitioned* schema having a two-column table for every RDF predicate, e.g. $knows(sub, obj)$, enabling more efficient pruning strategies. Otherwise, having a separate table for every predicate is not a well-suited schema for joins on predicates. This may not be common in SPARQL but it is a natural join pattern in TriAL*, e.g. for reasoning.

Another typical approach is the use of so-called *property tables* where all predicates (or properties) that tend to be used together are stored in one table, e.g. all predicates used to describe a person. In general, it has the schema $propTable(sub, pred_1, ..., pred_n)$ where the columns (predicates) are either determined by a clustering algorithm or by the class of the subject. This reduces the number of subject-subject self-joins for *star-shaped* query patterns. Although such patterns can also occur in TriAL*, it is not the dominant pattern in navigational queries. Furthermore, TriAL* is a closed language that allows us to derive new triples to be added to the triplestore which would result in update operations on one or more property tables, an operation that is currently not supported by any of the investigated systems (Impala, Spark, Hive) due to the fact that the underlying distributed filesystem (HDFS) is append-only.

In our use case, we basically need to consider two aspects in our table layout: (1) the fact that the data is distributed on a cluster of machines hampering the use of

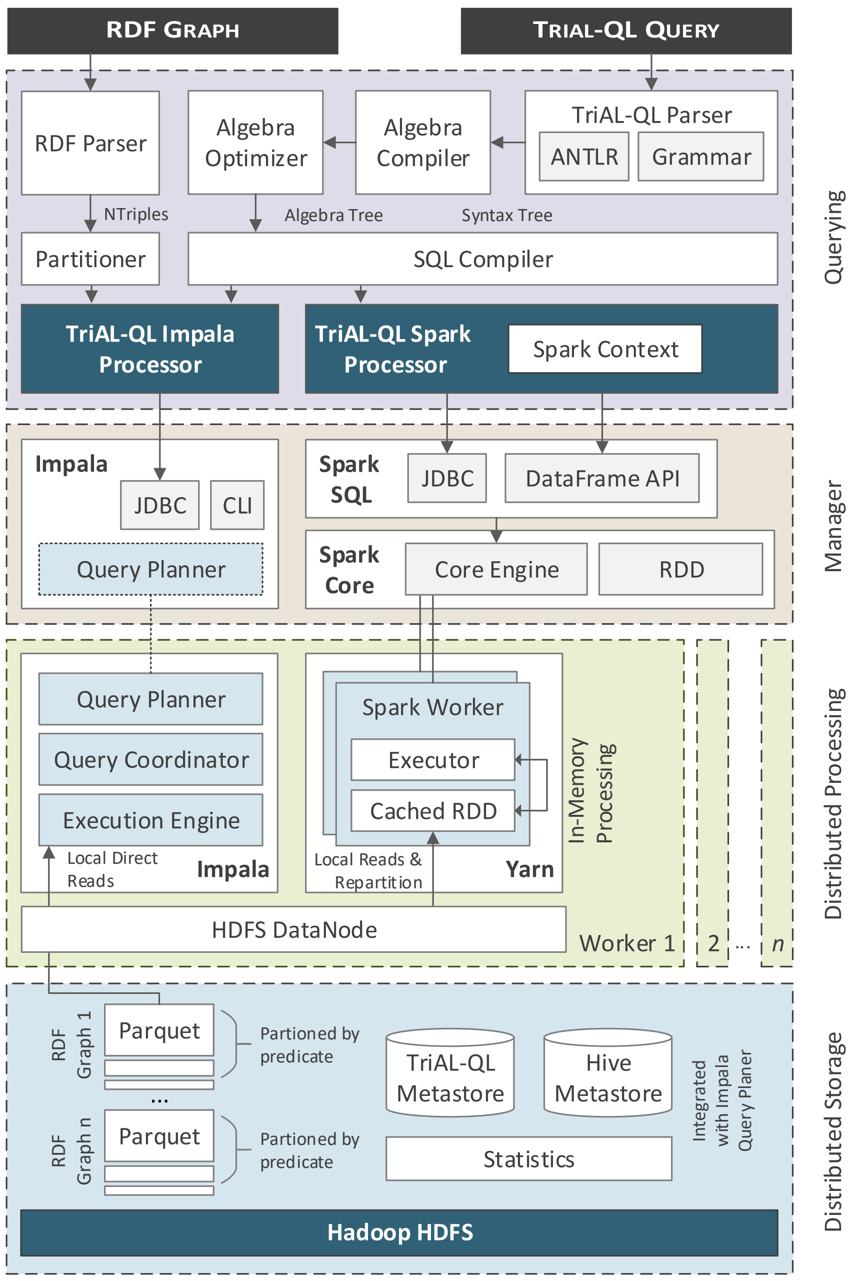


Figure 7.3.: Overview of TriAL-QL Engine architecture

indexes, and (2) the flexibility imposed by TriAL* to dynamically add new triples and perform joins on all possible pattern combinations (i.e. also predicate-predicate joins). With this in mind, we use a triples table internally partitioned by predicates, $triples_{pred}(sub, obj)$, to represent an RDF dataset and use the name of the dataset as the name of the triplestore (table). Tables are stored using *Parquet*[2], an efficient columnar storage format for Hadoop with built-in support for compression

[2]`http://parquet.apache.org/`

(snappy), run-length, and dictionary encoding[3]. This table layout is supported by all three systems, hence we can use the very same table for all of them, demonstrating the benefit of a unified data storage. Partition pruning is applied transparently whenever possible by all systems, i.e. unnecessary partitions are filtered out automatically, and table statistics (e.g. partition sizes) are used to optimize the query plans, e.g. improving join order. Thus, we combine the full flexibility of a triples table with the efficiency enhancement of vertical partitioning, while having just a single unified data storage without the need for expensive data exchange or conversion from one system to another. These strategies are also applied to the intermediate and final results of TriAL-QL queries, which in turn facilitates the compositionality of expressions and provides a simple interoperability with, e.g., Hadoop-based SPARQL engines that can use Parquet as input [SPNL14]. Further, to improve the data distribution of particularly small vertical partitions which would be stored in a single file, which in turn might lead to queries executed on a single node, the internal Parquet file size is adapted with regard to the underlying RDF data size.

7.2.2. Query Compiler

This section discusses the process of compiling a TriAL-QL query to a sequence of SQL queries. For that, we will first start with a short overview of the complete translation process, followed by a detailed look into the actual evaluation algorithm used for certain query pattern. The last aspect will form the main part of this section, where we mostly focus on two interesting query patterns: (1) recursions expressed by the left and right Kleene Closure of TriAL* and (2) the connectivity between two resources of arbitrary length.

In short, the Query Compiler parses in a first step the TriAL-QL query to generate an abstract syntax tree out of it using the grammar discussed in Section 5.4 and shown in Appendix C. The corresponding TriAL-QL parser uses ANTLR[4]. Next, the resulting syntax tree is translated to an algebra tree by means of the *Algebra Compiler* as illustrated in Figure 7.3. Next, a few basic optimizations such as filter push-down are applied. Finally, the algebraic tree representation is forwarded to the SQL Compiler which produces either Impala or Spark SQL queries together with an execution plan. This plan particularly focuses on cases where recursive expression need to be evaluated since, at the time of working on the system, there was no support for recursion in the SQL dialect of both frameworks. An example of this process is shown in Figure 7.4, with a (simplified) outline of an Impala SQL query which computes the first TriAL-QL expression ① required for this exemplary query.

[3]We also performed some experiments with others storage formats including *RCFile*, *Avro* and *SequenceFile*. However, Parquet exhibits the overall best performance for our type of queries.

[4]http://www.antlr.org/

```
             foaf = SELECT S1 P2 O2 FROM sib
                    ON O1=S2 AND P1=P2 AND P1='foaf:knows'    ①
                    USING left(2);
            likes = SELECT S1 P2 O2 FROM foaf JOIN sib
                    ON O1=S2 AND P2='sib:like';               ②
recommendedItems = likes MINUS sib;
STORE recommendedItems AS result;                             ③
```

$$\text{foaf} = (\text{sib} \bowtie^{s1,\ p2,\ o2}_{o1=s2,\ p1=p2,\ p1=\text{'foaf:knows'}})^2 \qquad ①$$

$$\text{recommendedItems} = (\text{foaf} \bowtie^{s1,\ p2,\ o2}_{o1=s2,\ p1=\text{'sib:like'}} \text{sib}) - \text{sib} \qquad ②\ ③$$

```
INSERT INTO foaf
SELECT DISTINCT A.s AS s, B.p AS p, B.o AS o
FROM sib A JOIN sib B                                         ①
ON A.o=B.s AND A.p=B.p AND A.p='foaf:knows'
...
```

Figure 7.4.: Translation from TriAL-QL to Impala SQL

Evaluation of Kleene Closure. One of the most challenging expressions of TRIAL* is the *right* $(e \bowtie^{i,j,k}_{\theta,\eta})^*$ and *left* $(\bowtie^{i,j,k}_{\theta,\eta} e)^*$ *Kleene closure*, which allows the expression of paths of arbitrary length. Its computation requires a continuous sequence of iterations until *all* reachable instances are retrieved. Since, Impala, Hive and Spark do not support any kind of recursion, such an expression cannot be translated into a single SQL statement but needs to be broken down into several smaller queries that are executed subsequently. Therefore, an additional mechanism is needed that (1) initiates each of these iterations and (2) determines the progress and decides whether it terminates. For Impala and Hive, the intermediate result of each iteration has to be materialized on disk and used as input for the next iteration as there is no support to preserve those intermediate tables in memory across multiple SQL queries. However, Spark supports the caching of tables in memory, such that a subsequent iteration can read intermediate results from main memory.

For the actual processing, we need to define an efficient algorithm which evaluates the recursive expressions in TRIAL*. Indeed, we can reduce such an expression to the problem of calculating the transitive closure (TC), which is a well-studied research field [CCH91, FGL+15, Ioa86]. There is an ongoing debate whether the so-called *semi-naive* or *smart* TC algorithm is superior in distributed environments like MapReduce [ABC+11, SKHS12]. However, there has not been much work yet on investigating the trade-offs using novel in-memory SQL-on-Hadoop solutions. We believe that, for our scenario, a *semi-naive* evaluation [CCH91] is the better choice as it distributes the workload over more rounds and produce less derivations on graphs with cycles [AU12]. In contrast, a *smart* TC algorithm based on a *non-linear* (*recursive-doubling*) execution [Ioa86] uses a logarithmic, rather than linear, number of rounds but with much higher costs (with regard to the data volume) per round [AU12]. Thus, using the *semi-naive* evaluation leads to more but less-expensive joins. This in turn increases the chances that the join processing can be done in main-memory without spilling to disk.

Our first algorithm for computing the *right Kleene closure* $(E \bowtie^{i,j,k}_{\theta,\eta})^*$ based on *semi-naive* evaluation is depicted in Algorithm 7.2.1.

Algorithm 7.2.1 : Semi-naive evaluation of *right Kleene closure*

```
input : triplestore $E$
1 $i \leftarrow 0$, $\Delta P^0 \leftarrow E$
2 while $\Delta P^i \neq \emptyset$ do
3     $i \leftarrow i + 1$
4     $tmp = \Delta P^{i-1} \bowtie^{i,j,k}_{\theta,\eta} E$                    // paths of length $i + 1$
5     $\Delta P^i = tmp - (\Delta P^0 \cup ... \cup \Delta P^{i-1})$     // remove previously discoverd paths
6 end
7 return $P = \Delta P^0 \cup ... \cup \Delta P^i$
```

The *left Kleene closure* is defined analogously. E denotes the input triplestore (table) and ΔP^i contains those triples which were newly derived in the i-th iteration where the shortest path between them has length $i + 1$. Any triple produced by the join in line 4 with a shortest path of length $< i + 1$ has already been discovered in previous iterations and is removed with the set operation in line 5. The final result, P, is the union of all previous iterations, $P = \bigcup_{\forall i} (\Delta P^i)$.

In addition, this approach can be further improved by exploiting some properties of partitioned tables in Impala and Hive. Instead of creating a new table for each ΔP^i, it is far more effective to use a single table $P_{iter}(sub, pred, obj)$ partitioned by iteration number *iter* and only add a new partition to that table. This way, the amount of required operations can be further reduced since the results of all *union* operations used in line 5 and line 7 can be retrieved by partition pruning without any computational effort. At the time when we performed our experiments, the support of Spark for partitions was in a rather early stage, not allowing us to adapt this strategy as efficiently as for Impala and Hive. Thus for Spark we need to introduce two tables: one for newly-derived triples (ΔP^i), and a second one that keeps the result of $(\Delta P^0 \cup ... \cup \Delta P^{i-1})$. The algorithm terminates if round i does not derive any new triples, i.e. all the derived triples are already contained in $(\Delta P^0 \cup ... \cup \Delta P^{i-1})$ or $(\Delta P^{i-1} \bowtie E)$ is empty. This step is realized by an additional SQL query that counts the number of triples in ΔP^i.

To close, we recall the example introduced in Section 5.2 with the TriAL* expression $(E \bowtie^{s_1,\, p_1,\, o_2}_{o_1 = s_2})^*$. For this example, the translation into Impala SQL generates for each iteration (Algorithm 7.2.1, line 4 + 5) the following two queries:

```
INSERT OVERWRITE tmp
SELECT DISTINCT t1.s, t1.p, t2.o
FROM P t1 JOIN E t2 ON t1.o = t2.s
WHERE t1.iter = i − 1;
INSERT INTO P PARTITION (iter)
SELECT s, p, o, i AS iter
FROM tmp LEFT ANTI JOIN P USING (s, p, o);
```

Our second algorithm for computing the *right Kleene closure* $(E \bowtie_{\theta,\eta}^{i,j,k})^*$ based on *smart* TC evaluation [AU12] is depicted in Algorithm 7.2.2. The *left Kleene closure* is defined analogously.

Algorithm 7.2.2 : Smart evaluation of *right Kleene closure*

```
input : triplestore $E$
1  $i \leftarrow 0$
2  $P^0 \leftarrow E$
3  $Q^0 = E \bowtie_{\theta,\eta}^{i,j,k} E$
4  $Q^0 = Q^0 - P^0$
5  while $Q^i \neq 0$ do
6      $i = i + 1$
7      $P^i = Q^{i-1} \bowtie_{\theta,\eta}^{i,j,k} P^{i-1}$     // paths of length between $2^{i-1}$ and $2^i - 1$
8      $P^i = P^i \cup P^{i-1}$     // all paths of length up to $2^i - 1$
9      $Q^i = Q^{i-1} \bowtie_{\theta,\eta}^{i,j,k} Q^{i-1}$     // all paths of length $2^i$
10     $Q^i = Q^i - P^i$     // remove previously derived triples
11 end
12 return $P^i$
```

Essentially, we can see that in each iteration the computation of new triples is broken down into two parts. The first part computes P^i (line 7), which discovers paths whose length is between 2^{i-1} and $2^i - 1$. P^i is then merged with P^{i-1} and contains now *all* triples, which have been derived so far, i.e. have a max length of $2^i - 1$. In the next step Q^i (line 9) is computed which retrieves only those paths whose length is *exactly* 2^i. In order to ensure that we are considering only shortest paths, all previously-derived triples contained in P^{i-1} are removed from Q^i in line 10. Now, we can use Q^i to determine if the smart algorithm can terminate. This is the case if Q^i is empty, thus no *new* paths of length 2^i have been discovered in round i. The final result is then the latest P^i.

It would also have been possible not to break the computation into two parts, and to retrieve all paths of length 2^{i-1} to 2^i in one step. However, in that case, we would not be able to determine already in round i, whether all retrievable triples have been derived. For that, a further costly round $i+1$ would have been required, which seeks for paths of length 2^{i+1}.

Evaluation of Connectivity Patterns. A further challenging query type that we identified to be relevant for many application fields is the connectivity between two given resources, thus the existential question whether there exists a path that connects both, written as:

$$\sigma_{s=\text{<startNode>},\ o=\text{<endNode>}}\Big((E \bowtie_{\theta,\eta}^{i,j,k})^*\Big)$$

Like the aforementioned *Kleene closure* expression, its computation involves a continuous sequence of iterations, and thus it cannot be translated into a single SQL statement. A naive compositional evaluation with the previous algorithm would derive a huge amount of redundant triples that need to be discarded afterwards. However, instead of retrieving all connections between both resources, we solely need to check for the existence of at least *one* path connecting them. The corresponding algorithm is depicted in Algorithm 7.2.3.

Again, we use the *semi-naive* evaluation, but this time we start with two initial tables (ΔP_l^0 and ΔP_r^0), which contain only those triples that *start* and *end* with the given resources, respectively. The algorithms then alternately derive new triples for ΔP_l^i and ΔP_r^i. In other words, we perform a *breadth-first* search that starts from both ends. As in the previous case, we can exploit Impalas partitioning strategy by creating two tables, $Pl_{iter}(sub, pred, obj)$ and $Pr_{iter}(sub, pred, obj)$, partitioned by iteration number to reduce the computational effort. Finally, the existence of a path of length i is checked by joining the two tables for ΔP_l^i and ΔP_r^i, where just the number of results is needed (denoted by res). The algorithm terminates, if either a connection is found, thus $res \neq 0$, or both ΔP_l^i and ΔP_r^i are empty. In this case, we computed the *reachability* starting from both ends without finding an intersection.

The remaining E-TRIAL expressions (cf. Section 5.2 and Table 5.1) can be realized using their equivalent clauses in the SQL dialect of Impala. The only exception is the minus operator ($e_1 - e_2$), for which we need to use a so-called *left anti join* as there is currently no support for set operations in Impala.

Algorithm 7.2.3 : Semi-naive evaluation of *connectivity pattern*

```
input : triplestore $E$, resource $startNode$, resource $endNode$
1  $i \leftarrow 0$
2  $\Delta P_l^0 \leftarrow \sigma_{s\,=\,startNode}(E),\ \ \Delta P_r^0 \leftarrow \sigma_{o\,=\,endNode}(E)$
3  $res = count\left(\Delta P_l^{\lceil i/2 \rceil} \bowtie_{\theta,\eta}^{i,j,k} \Delta P_r^{\lceil i/2 \rceil}\right)$
4  while $\left(\Delta P_l^{\lceil i/2 \rceil} \neq \emptyset \ \ \& \ \ \Delta P_r^{\lceil i/2 \rceil} \neq \emptyset\right) \ \& \ (res = 0)$ do
5      $i \leftarrow i + 1$
6      if $(i \ mod \ 2) = 1$ then
7          $tmp_l = \Delta P_l^{\lceil (i-1)/2 \rceil} \bowtie_{\theta,\eta}^{i,j,k} E$          // joins from startNode
8          $\Delta P_l^{\lceil i/2 \rceil} = tmp_l - \left(\Delta P_l^0 \cup ... \cup \Delta P_l^{\lceil (i-1)/2 \rceil}\right)$
9      else
10         $tmp_r = E \bowtie_{\theta,\eta}^{i,j,k} \Delta P_r^{\lceil (i-1)/2 \rceil}$          // joins from endNode
11         $\Delta P_r^{\lceil i/2 \rceil} = tmp_r - \left(\Delta P_r^0 \cup ... \cup \Delta P_r^{\lceil (i-1)/2 \rceil}\right)$
12     end
13     $res = count\left(\Delta P_l^{\lceil i/2 \rceil} \bowtie_{\theta,\eta}^{i,j,k} \Delta P_r^{\lceil i/2 \rceil}\right)$
14 end
15 return $res$
```

7.2.3. Query Processor

The role of this component is to execute the translated SQL on Impala and Spark respectively. For that, multiple execution strategies are available which will be introduced in this section. Furthermore, in cases of recursive expression a Query Processor also has to check for the termination conditions (line 2 in Algorithm 7.2.1 and line 4 in Algorithm 7.2.3). The underlying idea behind the condition in Algorithm 7.2.1 is that, if round i does not derive any new triples, i.e. all the derived triples are already contained in $(\Delta P^0 \cup ... \cup \Delta P^{i-1})$ or $(\Delta P^{i-1} \bowtie E)$ is empty, then all possible triples were already discovered in previous rounds. This step is realized by an additional SQL query that counts the triples in ΔP^i. In case of SPARK, the Query Processor is able to retrieve this information directly by accessing the corresponding data structures and get the size of ΔP^i. The basic idea behind the termination condition of Algorithm 7.2.3 is based on the same principle but needs further restriction. Namely, we need to check for both sides (ΔP_l^i and ΔP_r^i) that they do not derive any new triples, i.e. we computed the *transitive closure* starting from both ends without finding an intersection. Alternatively, if $res \neq 0$ we know that a connection is found.

Materialized and Composite Execution. As described previously, each TriAL expression is mapped to one or more SQL queries. In a sequence of queries, $q_0 \dots q_i$, where q_i uses the output of q_{i-1} as input, we can exploit the fact that SQL itself is also a closed and compositional language. Instead of executing i isolated queries sequentially (referred to as ***materialized*** execution), we can combine them into a single composite query (referred to as ***composite*** execution). This is especially interesting for Impala to avoid materializing the output of $q_0 \dots q_{i-1}$ to HDFS to serve as input for the next query. Unfortunately, we cannot use this strategy for recursive TriAL expressions as Impala currently does not support recursion. In Spark, we have the choice whether to materialize the result of a query in HDFS or not as Spark supports the caching of tables in main-memory. Moreover, we can even use a composite execution for recursive TriAL expressions by utilizing the fact that Spark supports the writing of SQL queries within a Scala or Java program. For Hive (on MapReduce), it does not make a difference between both strategies as intermediate results are anyway materialized in HDFS.

However, composition is not always superior to a sequential execution mainly for two reasons: (1) Composition complicates the query plan and may lead to incorrect cardinality estimations in a sequence of joins and thus suboptimal execution plans. (2) We observed that a sequence of small queries reduces the overall main-memory usage compared to one large composed query. This in turn decreases the probability that the system needs to spill to disk which would slows down the join processing substantially.

7.3. Experiments on TriAL-QL Engine

The experiments were performed on a small cluster with ten machines using the Hadoop distribution Cloudera CDH 5.3.2 with Impala 2.1.2, Spark 1.2.0 and Hive 0.13.1[5]. There is no existing benchmark that would highlight the strengths of TriAL*, and thus we introduce a set of representative use cases with different characteristics that enable us to demonstrate the expressiveness of TriAL-QL with its scalability costs on large RDF graphs. The actual queries are inspired by typical graph analytical questions and recommendations on social networks. We use the state of the art *Social Network Benchmark* (SNB) data generator[6] to generate synthetic social networks of up to 1.8 billion triples (edges) with a power-law structure which also has been used in the SIGMOD 2014 programming contest[7]. The load times and store sizes are listed in Table 7.1. We examine both execution strategies introduced in Sect. 7.2.3 (*materialized* and *composite*) for Impala and Spark and compare them with an execution on Hive using MapReduce. All results are listed in Table 7.2. Each query was executed five times, and the corresponding coefficients of variation c_v (ratio between standard deviation and mean) are also given. The total runtimes in seconds include both computing statistics and writing the result to Parquet tables on disk.

Table 7.1.: Load times and store sizes

Scaling Factor	**1**	**3**	**10**	**30**
Triples (in M)	59.5	176.4	594.7	1,799
Original RDF size	4.1 GB	12.3 GB	41.4 GB	126 GB
Triples Table size	0.6 GB	1.7 GB	6.2 GB	19 GB
Loading in RDF Store	32.4 s	90.4 s	299.3 s	893.1 s
Computing statistics	4.2 s	6.8 s	16.1 s	42.6 s
Parquet file size	8 MB	8 MB	16 MB	32 MB

[5]More information on the cluster description can be found in Chapter 8 on page 195
[6]`http://ldbcouncil.org/developer/snb`
[7]`http://www.cs.albany.edu/~sigmod14contest/`

Use Case 1: Socialized Recommendations. In the first experiment we ask for the posts of a user's friends that he has not liked yet, computed for all users in parallel. For the sake of brevity, we omit the TriAL* expression for *knows*, *posts* and *likes*, where each is retrieved by an additional join on the input table *snb*. However, they are included in the execution times. The corresponding TriAL* expressions are defined as follows:

$$\begin{aligned} friendsPosts &= knows \bowtie^{s_1,\ \text{fposts},\ o_2}_{o_1=s_2} posts \\ likedFriendsPosts &= friendsPosts \bowtie^{s_1,\ \text{fposts},\ o_2}_{s_1=s_2,\ o_1=o_2} likes \\ postSuggestions &= friendsPosts - likedFriendsPosts \end{aligned}$$

The query contains no recursion and could also be expressed with SPARQL, but it illustrates the strength of composition where an expression processes the results of previous ones. In total it consists of five joins and a set operation that process large portions of the underlying RDF graph. The execution times and result size in triples are shown in Table 7.2. We can see that the *composite* execution with Impala performs best (2x faster than *materialized* on average) while scaling almost linearly with increasing data size. Storing the intermediate tables using the *materialized* strategy was not advantageous in this case and on average two times slower. Spark is competitive for smaller data sizes but gets significantly slower for larger ones. Hive is an order of magnitude slower than Impala but still scales out smoothly with larger data sizes. In summary, for rather data-intensive but not-too-complex queries as used for this use case, Impala performs best while Hive exhibits slightly better scaling properties.

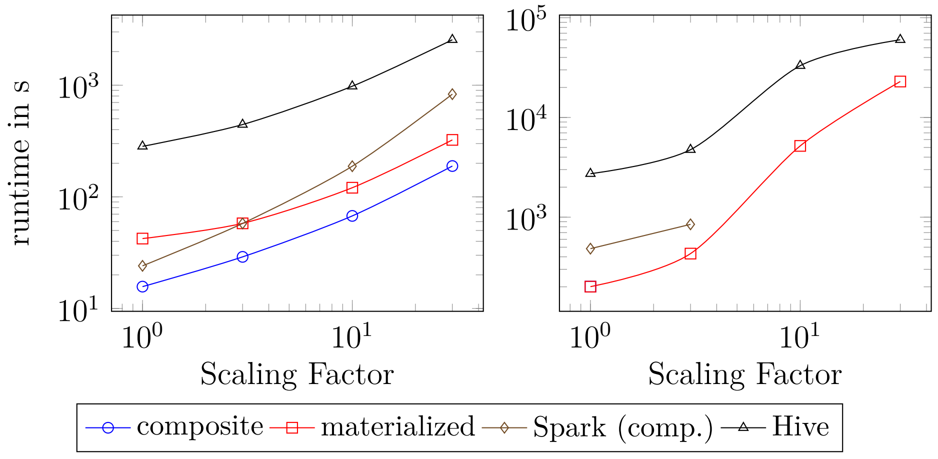
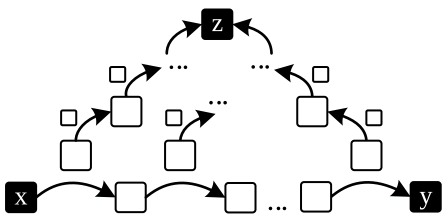


Figure 7.5.: Mean runtimes in seconds for Use Case 1 (left) & Use Case 2 (right) on a log-log scale.

Use Case 2: Reachability. This experiment defines a reachability query that cannot be answered by just traversing the graph with regular expressions, but which instead needs reasoning capabilities. We consider two arbitrary persons to be reachable if we can derive a path between them where (1) each intermediate pair of persons works together in the same company, and (2) all persons along the path share the same spoken language. The structure of this query exhibits a pattern of the following shape:



The corresponding TriAL* expressions are as follows:

$$\begin{aligned}
worksAt &= \text{snb} \bowtie^{s_1,\ \text{worksAt},\ o_2}_{o_1=s_2,\ p_1=sn{:}workAt,\ p_2=sn{:}hasOrga} \text{snb} \\
sameWork &= worksAt \bowtie^{s_1,\ \text{worksWith},\ s_2}_{o_1=o_2,\ s_1!=s_2} worksAt \\
sameLang &= \text{snb} \bowtie^{s_1,\ o_1,\ s_2}_{o_1=o_2,\ s_1!=s_2,\ p_1=p_2,\ p_1=sn{:}speaks} \text{snb} \\
colleagues &= sameLang \bowtie^{s_1,\ p_1,\ o_1}_{s_1=s_2,\ o_1=o_2} sameWork \qquad ① \\
reach &= \left(colleagues \bowtie^{s_1,\ p_1,\ o_2}_{p_1=p_2,\ o_1=s_2}\right)^* \qquad ②
\end{aligned}$$

We split the analysis of the execution times in Table 7.2 into two parts: In ① we summarize all non-recursive expressions including *colleagues* and in ② we investigate the computation of the transitive closure expressed by *reach*.

For ① we can see that the Impala runtimes again scale almost linearly with increasing data sizes. But this time, *materialized* execution is superior to *composite* for larger datasets. We explain this behavior with a better execution plan of Impala due to statistics computed for intermediate tables. The costs for computing statistics are included in Table 7.2 (cf. column *statistics*). For ② again Impala outperforms Spark, although it was heavily spilling to disk during query execution (starting from SF 10). Moreover, Spark runs out of memory on larger datasets (indicated by `MEM`). This might result from the two table approach required for Spark (cf. Sect. 7.2.2) compared to the partitioned table approach used for Impala. As a consequence, Spark needs more memory and requires additional computational effort. Again, Hive was significantly slower than Impala and Spark but exhibits a better scaling capability for larger datasets. Looking more carefully, we can observe that the performance benefit of Impala is mainly attributed to the more efficient determination of newly derived triples (ΔP^i) rather than join computation. We discuss this in more detail at the end of this section, where we compare Impala with Spark.

Use Case 3: Connectivity. The last experiment asks for the existence of a path between two given people by following the friendship relationship. We choose 50 people randomly and compute their connectivity based on the following TriAL* expressions:

$$knows = snb \bowtie^{s_1,\ \text{knows},\ s_2}_{o_1=s_2,\ p_1=sn\text{:}knows,\ p_2=sn\text{:}hasPers}\ snb \qquad ③$$

$$path = \underset{s=\text{person\_1},\ o=\text{person\_2}}{\sigma}\left(\left(knows \bowtie^{s_1,\ p_1,\ o_2}_{o_1=s_2}\right)^{*}\right) \qquad ④$$

Note that ③ *knows* is only computed once and stored as a new relation (table) in the triplestore and is used to compute ④ *path* over and over again, bridging the gap between ETL-like workloads and explorative ad-hoc style queries. The execution times for both parts using Impala and Spark are shown in Table 7.2. Hive was not considered in this experiment since MapReduce start-up costs are already higher than desired runtimes for this query type. However, it will be interesting to investigate novel derivations of Hive (Hive on Tez, Hive on Spark) that aim to replace MapReduce with a more interactive execution framework for future work.

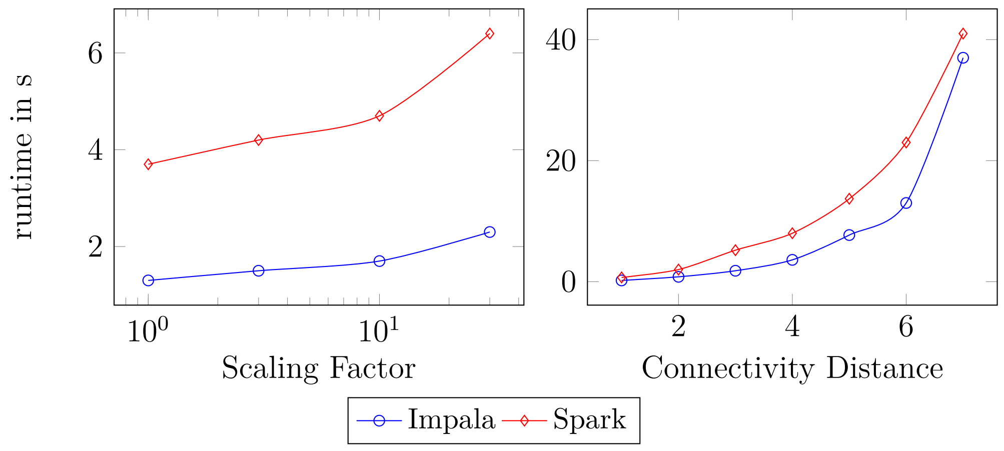


Figure 7.6.: Mean runtimes (in s) for Use Case 3: (left) total mean, (right) by path distance for SF 30

The mean runtime for computing the connectivity between two randomly chosen persons with Impala was only 2.3 seconds on a dataset with 1.8 billion triples while scaling smoothly with the data size (cf. Figure 7.6). For Spark it is 6.4 seconds which is still competitive. The average distance between two persons was 2.8 for the smallest dataset and increased up to 3.2 for the largest one. Table 7.2 also lists the runtimes by distance (denoted by ⑤). Considering distances of at most 2, we get runtimes of < 1 second and a maximum runtime of 37 seconds for a distance of 7. This is also illustrated in the right plot of Figure 7.6 where both Impala and Spark exhibit the same exponential scaling behavior.

Table 7.2.: Mean runtimes (in s) comparing *composite* and *materialized* execution on Impala, Spark- and Hive, c_v = coefficient of variation, ΔP^i and *tmp* match with Algorithm 7.2.1, *iter* is the number of needed iterations.

Use Case 1

	Impala (composite)		**Impala (materialized)**			**Spark (composite)**		**Hive**	
SF	total (c_v)	write	total (c_v)	statistics	write	total (c_v)	write	total (c_v)	results
1	15.7 (2.5%)	3.6	42.3 (4.8%)	6.9	5.3	24.1 (2.0%)	2.3	283.4 (1.3%)	1,161,253
3	29.0 (1.8%)	5.0	57.9 (2.9%)	9.4	5.0	57.8 (2.4%)	4.1	444.1 (0.8%)	3,569,709
10	67.5 (1.1%)	4.9	120.7 (1.0%)	20.1	4.7	188.1 (1.0%)	5.9	979.2 (0.5%)	12,635,382
30	188.8 (1.4%)	7.1	323.7 (1.4%)	51.6	5.4	831.5 (3.8%)	5.9	2553.5 (0.8%)	39,442,329

Use Case 2 ① *colleagues*

	Impala (composite)		**Impala (materialized)**			**Spark (comp.)**	**Spark (mater.)**	**Hive**
SF	total (c_v)	write	total (c_v)	statistics	write	total (c_v)	total (c_v)	total (c_v)
1	16.9 (2.2%)	3.9	29.3 (2.7%)	5.5	3.6	44.5 (1.3%)	50.0 (1.9%)	250.7 (1.0%)
3	38.3 (1.5%)	4.0	43.2 (4.9%)	6.4	4.2	83.8 (2.5%)	90.6 (1.1%)	294.4 (0.6%)
10	159.0 (0.7%)	5.1	121.1 (3.9%)	11.6	6.2	321.6 (1.3%)	333.2 (1.7%)	605.3 (0.8%)
30	866.6 (0.4%)	9.7	631.9 (6.2%)	42.4	9.1	2048.0 (5.2%)	2039.9 (2.7%)	2501.2 (0.3%)

Use Case 2 ② *reach*

	Impala (materialized)			**Spark (materialized)**			**Hive**				
SF	total (c_v)	ΔP^i	*tmp*	total (c_v)	ΔP^i	*tmp*	total (c_v)	ΔP^i	*tmp*	iter	results
1	201 (1.4%)	32	122	483.7 (2.2%)	396.1	87.6	2727 (0.3%)	1046	1101	17	2.6 M
3	431 (2.6%)	65	316	847.1 (0.7%)	539.6	307.5	4749 (1.5%)	1737	2424	17	19.3 M
10	5184 (1.0%)	362	4760	MEM	MEM	MEM	33048 (0.5%)	2768	29732	15	153 M
30	22933 (4.7%)	725	22148	MEM	MEM	MEM	60178 (1.4%)	4611	54929	14	591 M

Use Case 3

		③ *knows* (c_v)	④ *connectivity* (*mean*)		⑤*connectivity by distance*, total (c_v)						
	SF	total (c_v)	total	∅*distance*	1	2	3	4	5	6	7
Impala	**1**	4.4 (4%)	1.3	(∅ 2.8)	0.1 (0%)	0.5 (0%)	1.3 (3%)	2.7 (2%)	4.8 (3%)	7.9 (2%)	13 (2%)
	3	8.1 (4%)	1.5	(∅ 3.0)	0.1 (0%)	0.5 (3%)	1.4 (3%)	2.9 (2%)	5.3 (2%)	8.9 (2%)	14 (1%)
	10	13.0 (6%)	1.7	(∅ 3.0)	0.1 (24%)	0.6 (7%)	1.6 (5%)	3.2 (4%)	5.9 (3%)	9.9 (2%)	20 (1%)
	30	25.1 (2%)	2.3	(∅ 3.2)	0.2 (0%)	0.8 (2%)	1.8 (3%)	3.6 (3%)	7.7 (2%)	13 (2%)	37 (1%)
Spark	**1**	3.3 (0%)	3.7	(∅ 2.8)	0.4 (2%)	1.5 (6%)	4.1 (4%)	6.8 (5%)	12.1 (5%)	18 (5%)	33 (3%)
	3	7.7 (0%)	4.2	(∅ 3.0)	0.5 (1%)	1.6 (3%)	4.2 (1%)	6.8 (0%)	12.3 (3%)	19 (2%)	34 (1%)
	10	18.8 (3%)	4.7	(∅ 3.0)	0.5 (2%)	1.6 (1%)	4.2 (1%)	7.1 (7%)	12.6 (7%)	20 (4%)	35 (3%)
	30	56.6 (1%)	6.4	(∅ 3.2)	0.7 (30%)	2.0 (28%)	5.2 (15%)	8.0 (10%)	13.7 (5%)	23 (10%)	41 (9%)

Comparison of Impala and Spark. In summary, the overall performance of our TriAL-QL Engine using Impala was continuously better than for Spark. However, a more granular view on the actual costs for the respective operations reveals some additional interesting properties which highlight some strengths and weaknesses of both. Figure 7.7 illustrates the percentage of the total runtime of the respective operations for Use Case 2 (see Algorithm 7.2.1 in Sect. 7.2.2). Here we can see that computing ΔP^i, i.e. determine the newly derived triples in iteration i, dominates the total execution time of Spark, whereas this task is very efficient in Impala due to the partitioned table approach which is not applicable in Spark 1.2.0. However, if we consider the actual join processing costs ($tmp = \Delta P^{i-1} \bowtie^{i,j,k}_{\theta,\eta} E$), the performance of Spark is even faster than Impala. Thus, a better support for partitioned tables in future Spark versions might reveal other performance characteristics.

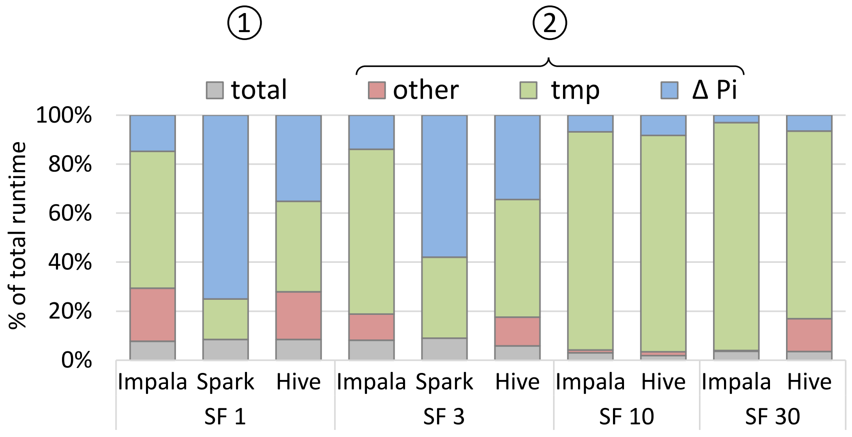


Figure 7.7.: Runtimes by task (in %) for Use Case 2

Another aspect is the superlinear increase in time required to compute ② *reach* starting from SF 10, which is mainly attributed to the actual join processing (*tmp*). Spark runs out of memory and fails while Impala starts to spill heavily to disk but is able to complete the job. This demonstrates that Impala is currently more robust when it comes to memory bottlenecks and thus the need for on-disk joins. Overall, at the time of our experiments, Impala is more suited for our kind of workload. Nonetheless, Spark has tremendous momentum and its unique features can be of great advantage for further developments, e.g. the ability to seamlessly combine SQL with arbitrary Spark programs.

7.4. Distributed Execution Engine for RDFPath

We continue our work on distributed RDF engines for navigational queries again with RDFPath. We revised our RDFPath MapReduce Processor from Section 6 and rebuilt it on top of Impala and Spark using SQL as an intermediate language. We refer to this new execution engine as RDFPath Engine throughout the dissertation. After obtaining very good experimental results with our TriAL-QL Engine, it emerged clearly that it would be much more beneficial to use the code base of TriAL-QL Engine rather than the one of RDFPath MapReduce Processor as a starting point. Many algorithms, which have proven to perform well for the evaluation of TriAL-QL, were with a few minor changes adoptable to evaluate RDFPath queries as well. With that in mind, we will discuss a few adopted algorithms from Section 7.2 together with new approaches such as our *Extended Vertical Partitioning* (ExtVP) introduced in [SPSL16][8], which shows promising properties for the evaluation of SPARQL queries.

The remainder of this section is structured as follows: After providing an overview of our implementation and presenting some details on the architecture, we continue in Section 7.4.1 with a discussion on the storage schema. Section 7.4.3 explains the process of translating an RDFPath query into SQL and provides an algorithm for evaluation. After that, Section 7.4.3 presents multiple strategies for the execution of translated SQL queries, followed in Section 7.5 by some experiments which investigate the properties of our engine.

System Overview. Our RDFPath Engine follows conceptually the previously discussed TriAL-QL Engine and can therefore be structured the same way, in three main components:

- **RDFp Store:** The basis for distributed querying is our common data pool with support for Spark and Impala. Each input graph is stored using the Vertical Partitioning (VP), as well as the Extended Vertical Partitioning (ExtVP) strategy. In order to represent RDFp an additional storage layer is included, which maps the paths into the columnar format of Parquet. A discussion of different RDF data-storage strategies and optimizations is presented in Section 7.4.1.
- **Query Compiler:** All tasks which are involved in translating an RDFPath query into the respective SQL dialect of the desired SQL-on-Hadoop system are composed in our Query Compiler. Section 7.4.2 will provide some details of what the actual translation process looks like and present an evaluation strategy for traversing paths.

[8]Please note, that ExtVP is part of the dissertation of Alexander Schätzle and is therefore only briefly noted in this work

- **Query Processor:** The final execution of SQL queries is done by the Query Processor. Section 7.4.3 will discuss multiple execution strategies which differ in how queries are composed and when intermediate results become materialized to disk. Further, due to the differences in the support of recursions in the respective SQL-on-Hadoop systems, individual Query Processors exist for Impala and Spark.

A more detailed view of the architecture of our RDFPath Engine is illustrated in Figure 7.8. Due to the common roots with TriAL-QL Engine, we moved unchanged components to the background (in gray) and highlight new ones. For clarity of presentation, we will stick to the three previously-mentioned components in the following sections.

7.4.1. RDFp Data Layout

RDFp was introduced in Section 4.2 as a flexible data model to represent paths in RDF graphs. It formed the basis for RDFPath by enabling more meaningful, path-based results. Accordingly, to evaluate RDFPath queries on in-memory frameworks, we need at first to define a proper data representation of RDFp in the stack of SQL-on-Hadoop solutions which meets our requirements. First of all, the data needs to be efficiently queryable with SQL, which is the key requirement if we want to use SQL as an intermediate language. However, in contrast to our RDFPath MapReduce Processor, we do not want to implement our own low-level serialization but rather benefit from existing data stores and their optimizations. Secondly, we again need to handle the tradeoff of maintaining complete RDFp paths efficiently, but keeping individual resources still accessible. Lastly, as we want our engine to use *multiple* in-memory frameworks but *without* multiplying the data, we need solutions that facilitate the usage of *one* common data representation. With Parquet stored in the Hadoop Distributed File System (HDFS), we have already introduced (in Section 7.2.1) a highly-compatible distributed storage that is also favored by most SQL-on-Hadoop engines. It has demonstrated very good properties for querying web-scale RDF with TriAL-QL, which leads us to the decision to investigate its usage for storing RDFp. We first start with an introduction of how we map RDFp to the columnar storage format of Parquet. We will then continue with optimizations built on top of our storage schema including Vertical Partitioning and Extended Vertical Partitioning.

Modeling RDFp in Parquet. Following the notation in Section 4.2, where we introduced RDFp, the basic element of each RDFp path is composed of a resource r_i representing an RDF term with $r_i \in (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$. The general structure of the entire RDFp path is then an n-ary tuple $p = (r_1, r_2, r_3, ..., r_n)$ composed of n resources r_i. In the case of RDF triplestores, we represented RDF datasets in a *triples table* with three columns, containing one row for each RDF statement,

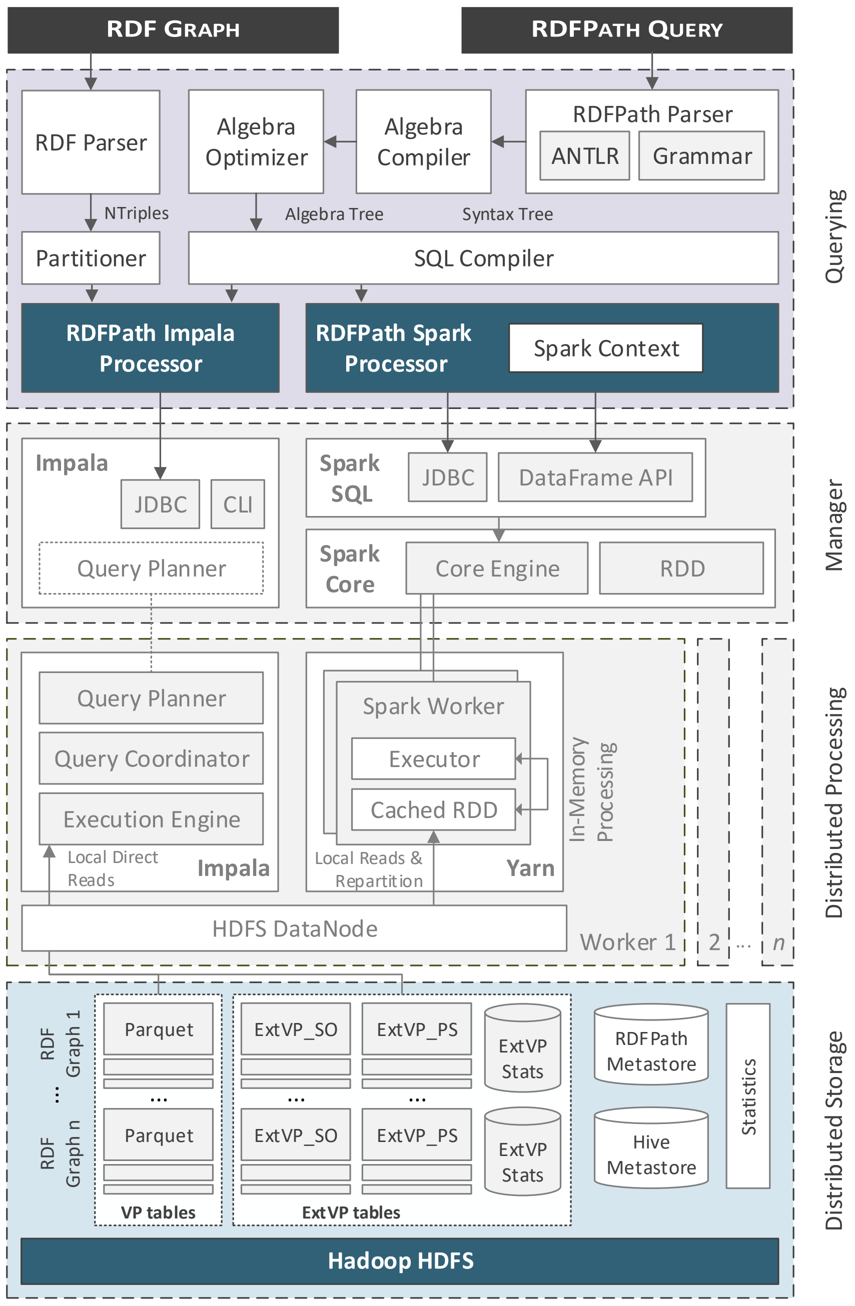


Figure 7.8.: Overview of RDFPath Engine architecture

i.e. *triples*(*sub*, *pred*, *obj*). We will next investigate how to refine this concept such that each row represents one RDFP path. The main issue is related to the flexibility of RDFPath, which enables us to retrieve paths of *different* length by using, e.g. recursion or branches. Without an implicit projection to a fixed number of resources as we have seen, for instance, with TriAL-QL where each operators produces again a triple-based data, we cannot simply create a fixed-length columnar mapping of RDFP paths to Parquet. We illustrate this in the following short example.

Example 7.1. For example, consider an RDFp Graph $\mathcal{P} = \{p_1, p_2, p_3\}$ which we want to model in Parquet.

$$\begin{gathered} p_1 : (Ted,\ country,\ DE), \\ p_2 : (Bob,\ knows,\ Alice,\ knows,\ Ted), \\ p_3 : (Bob,\ knows,\ Alice,\ knows,\ Ted,\ age,\ 31) \end{gathered}$$

A straight-forward mapping of these paths into a columnar format would result in the following table:

r_1	r_2	r_3	r_4	r_5	r_6	r_7
Ted	*country*	*DE*	–	–	–	–
Bob	*knows*	*Alice*	*knows*	*Ted*	–	–
Bob	*knows*	*Alice*	*knows*	*Ted*	*age*	31

The result of this mapping are three rows with a different number of columns. The main drawback in this representation is that we are not able to apply a join directly on this representation, for which one needs to specify a join column, e.g. r_7. That violates our first requirement, which is the ability to express navigational traversing steps as SQL. □

We have considered three solutions to overcome this issue, which we will briefly discuss in the following.

- **Object-based RDFp:** Analogous to our RDFPath MapReduce Processor, we can implement once more an optimized RDFp object including a serialization format and store it binary as a row in the columnar storage format of Parquet. However, this would imply new join operators in the respective SQL-on-Hadoop solutions. Moreover, we would also lose many important features of Parquet, such as filter push-down. There exists related projects from which we could benefit, such as Avro, a commonly used serialization platform which is interoperable across multiple languages and systems and is supported by Spark and Impala. Nonetheless, the effort of implementing new operators and not benefiting from a highly-compatible and continuously-optimized system-composition are strong arguments against this approach.
- **Nested RDFp:** Parquet supports nested columns (also known as complex types), which allow us to represent multiple data values in a single column position. This is exactly what is needed to overcome the issue shown in the example above. However, at the point of working on that, the support of Spark and Impala for nested columns was in an early stage, hampering its use.
- **Placeholder Padding for RDFp:** One further approach is to balance the differences in length of each row by filling up empty columns with placeholders, in our case NULL values. Relevant resources, thus possible join candidates,

then need to be moved to the last columns in each row. This method implies a few additional costs, for example, in the final output, where placeholders need to be removed again. However, in contrast to row-oriented storage schema, the columnar format of Parquet is well designed for wide tables that contain hundreds of columns, where only a few of them are accessed by a query. Moreover, NULL values are not stored explicitly in Parquet, thus sparsity causes just little to no storage overhead.

Finally, the *Placeholder Padding* strategy proved to provide a good tradeoff until the support for nested data structures in Impala and Spark improves. Therefore, we decided to use it for storing RDFP in Parquet. The following table illustrates the structure of a table after applying *Placeholder Padding* on the data from Example 7.1.

r_1	r_2	r_3	r_4	r_5	r_6	r_7
Ted	–	–	–	–	*country*	*DE*
Bob	*knows*	*Alice*	–	–	*knows*	*Ted*
Bob	*knows*	*Alice*	*knows*	*Ted*	*age*	31

We can see that possible join candidates were moved to the end of each row, enabling the application of joins directly on this representation. Since each join implies a new table with adjusted placeholders, there is also no need to modify a table with its placeholders.

Optimized storage schema. Input graphs become preprocessed during the initial loading, where the *Vertical Partitioned* (VP) schema proposed in [AMMH07] is applied on the data. As a result, we obtain a two-column table for every RDF predicate, e.g. $knows(sub, obj)$. This enables more efficient pruning strategies since following a certain predicate is a dominant pattern for a navigational query language. In [SPSL16], we have introduced an extension of this well known strategy called *Extended Vertical Partitioning* (ExtVP), which aims at further reducing the input size of each join. The basic idea is to avoid so-called *dangling tuples* in join input tables, i.e. tuples which do not find a join partner and become discarded by a join operation. This can be achieved by applying semi-joins between all combinations of vertical partitions (predicates). The results are new input tables for each combination of predicates, which include only those tuples which would contribute to the respective join. We refer to these input tables as ExtVP tables, each describing one *pairs of predicates*, e.g.

$$knows_{knows/age}(sub, obj)$$
$$age_{age/knows}(sub, obj)$$

For instance, in the case of the ExtVP table $knows_{knows/age}$, we store a subset of *knows* tuples with users for which an *age* is provided. Consider the following RDFPath query that traverses the predicate *knows* followed by *age*:

```
Bob :/knows /knows /age
```

In that example, one can use the smaller ExtVP table $knows_{knows/age}$ rather than the VP table *knows* as input. This simplified example assumes only subject-object joins occur during query execution. However, further possible combinations exist that need to be considered for RDFPath. We continue with a more formal definition based on the notation introduced in [SPSL16], where we define ExtVP on RDF. ExtVP on RDFP is defined analogously and is left out for clarity of presentation.

Definition 7.1 (Vertical Partitioning). Let G denote an RDF graph. A *vertical partitioning* schema over G, written as $VP[G]$, is then defined as:

$$VP_{p_1}[G] = \{(s,o) \mid (s,p,o) \in G \ \wedge \ p = p_1\}$$

□

Definition 7.2 (Extended Vertical Partitioning [SPSL16]). Let $\mathcal{P} = \{p \mid \exists s,o : (s,p,o) \in G\}$ denote the set of all predicates in an RDF graph G. An *extended vertical partitioning* schema over G, written as $ExtVP[G]$, is defined as:

$$\begin{aligned}
ExtVP^{OS}_{p_1|p_2}[G] &= \{(s,o) \mid (s,o) \in VP_{p_1}[G] \ \wedge \ \exists (s',o') \in VP_{p_2}[G] : o = s'\} \\
&\equiv VP_{p_1}[G] \ltimes_{o=s} VP_{p_2}[G] \\
ExtVP^{OS}[G] &= \{ExtVP^{OS}_{p_1|p_2}[G] \mid p_1, p_2 \in \mathcal{P}\} \\
\\
ExtVP^{PS}_{p_1|p_2}[G] &= \{(s,o) \mid (s,o) \in VP_{p_1}[G] \ \wedge \ \exists (s',o') \in VP_{p_2}[G] : p_1 = s'\} \\
&\equiv VP_{p_1}[G] \ltimes_{p_1=s} VP_{p_2}[G] \\
ExtVP^{PS}[G] &= \{ExtVP^{PS}_{p_1|p_2}[G] \mid p_1, p_2 \in \mathcal{P}\} \\
\\
ExtVP[G] &= \{ExtVP^{OS}[G], ExtVP^{PS}[G]\}
\end{aligned}$$

□

Note that, in [SPSL16] we focused on the evaluation of SPARQL queries, which requires different join combinations in the case of RDFPath queries. Not required partitions, such as $ExtVP^{SS}[G]$, are therefore discarded but new ones, such as $ExtVP^{PS}[G]$, additionally, are added. $ExtVP^{PS}[G]$ describes hereby predicate-subjects joins which form the basis for querying RDF data along with its ontology.

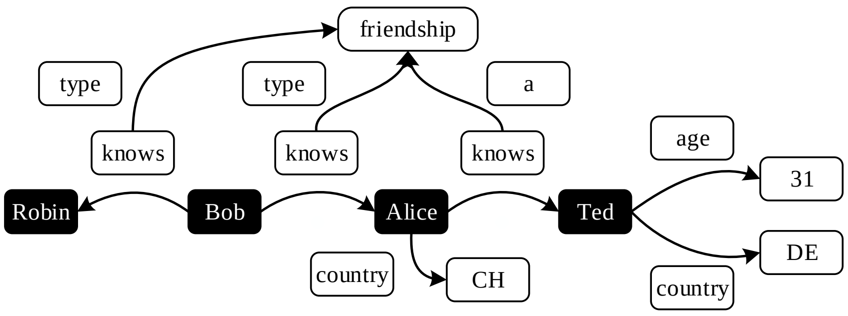


Figure 7.9.: RDF example describing different kinds of relations between people.

Example 7.2. Consider the graph G illustrated in Figure 7.9. Applying Definition 7.2 on G, we obtain the following ExtVP partitions:

$$
\begin{aligned}
ExtVP^{OS}_{knows|knows}[G] &= \{\ (Bob,\ knows,\ Alice)\ \} \\
ExtVP^{OS}_{knows|country}[G] &= \{\ (Bob,\ knows,\ Alice),\ (Alice,\ knows,\ Ted)\ \} \\
ExtVP^{OS}_{knows|age}[G] &= \{\ (Alice,\ knows,\ Ted)\ \} \\
ExtVP^{OS}_{age|knows}[G] &= \emptyset \\
ExtVP^{PS}_{knows|type}[G] &= \{\ (Bob,\ knows,\ Alice),\ (Bob,\ knows,\ Robin)\ \} \\
ExtVP^{PS}_{knows|a}[G] &= \{\ (Alice,\ knows,\ Ted)\ \} \\
ExtVP^{PS}_{country|type}[G] &= \emptyset \\
&\dots
\end{aligned}
$$

The remaining combinations of predicates which haven't been shown provide empty sets for our exemplary graph G and are therefore left out. □

In practice, only *beneficial* ExtVP partitions need to be kept. These are the partitions which are significantly smaller in comparison to the respective vertical partition. In order to identify them, we compute, analogous to [SPSL16], the benefit of an ExtVP partition in comparison to its VP partitions. We denote this value as selectivity factor SF.

Definition 7.3 (Selectivity Factor). Assume $ExtVP_{p_1|p_2}$ and VP_{p_1} to be two tables partitions in accordance to Definition 7.2. We define the *selectivity factor SF* of $ExtVP_{p_1|p_2}$ as its relative size in comparison to VP_{p_1}:

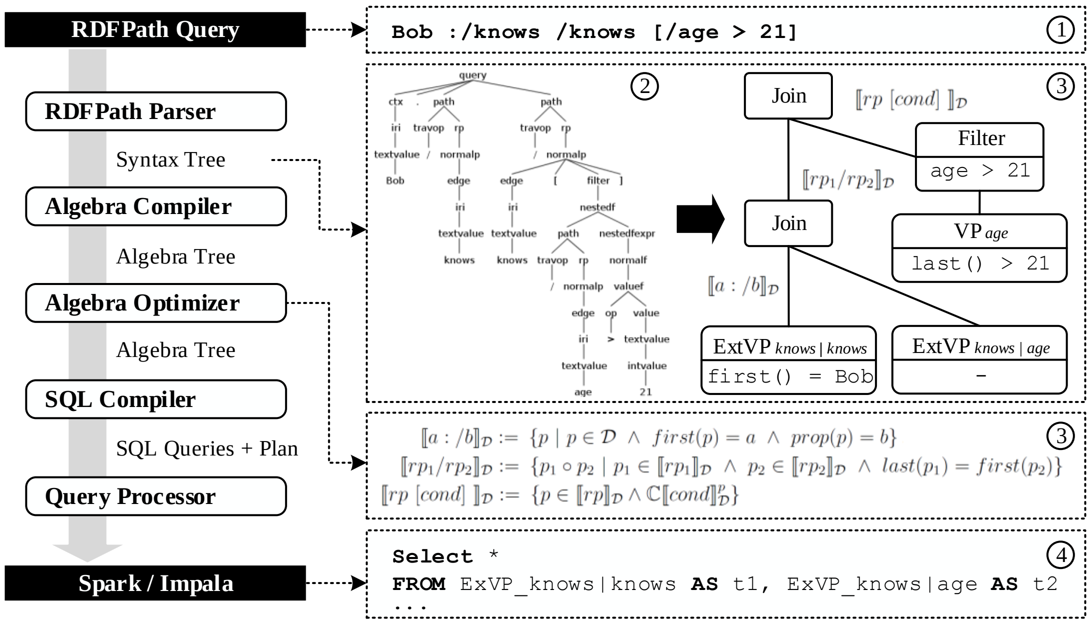
$$SF(ExtVP_{p_1|p_2}) = \frac{|ExtVP_{p_1|p_2}|}{|VP_{p_1}|}$$

□

Following this definition, an ExtVP table with SF ~ 1 has almost the same number of triples as its vertical partition. Storing it is not beneficial, since on the one hand it does not reduce the query input size significantly. On the other hand, it imposes a large (and redundant) overhead as the VP partition is simply duplicated. Thus, it makes sense to specify a threshold for the selectivity factor, where only those ExtVP partitions are kept, which are below that value. In addition, also information about completely empty ExtVP partitions are stored for query optimization. This includes also predicates which only lead to literals. For instance, empty RDFPath results are identified without actually executing the respective query.

7.4.2. Query Compiler

The translation of an RDFPath query to a sequence of SQL queries follows a standard approach illustrated in Figure 7.10. The Query Compiler parses as a first step the RDFPath query ① to generate an abstract syntax tree ② out of it using the grammar discussed in Chapter 4 and shown in Appendix B. Next, the resulting syntax tree is translated to an algebra tree ③ by means of the *Algebra Compiler*. The example further illustrates the aforementioned optimization by using two ExtVP partitions ($ExtVP^{OS}_{knows|knows}[G]$ and $ExtVP^{OS}_{knows|age}[G]$) rather than vertical partitions. Moreover, for $ExtVP^{OS}_{knows|knows}[G]$ and $VP_{age}[G]$ the filter push-down feature of Parquet is utilized, which ensures that only those rows are read which satisfy the filter constraint, e.g. `Bob` as a first resource and an age above 21. Lastly, the algebraic tree representation is forwarded to the SQL Compiler which produces either Impala or Spark SQL queries together with an execution plan. Those queries are then given to the Query Processor, which executes them in a last step on Impala and Spark respectively.



$$[\![a:/b]\!]_\mathcal{D} := \{p \mid p \in \mathcal{D} \wedge first(p) = a \wedge prop(p) = b\}$$
$$[\![rp_1/rp_2]\!]_\mathcal{D} := \{p_1 \circ p_2 \mid p_1 \in [\![rp_1]\!]_\mathcal{D} \wedge p_2 \in [\![rp_2]\!]_\mathcal{D} \wedge last(p_1) = first(p_2)\}$$
$$[\![rp\ [cond]\ ]\!]_\mathcal{D} := \{p \in [\![rp]\!]_\mathcal{D} \wedge \mathbb{C}[\![cond]\!]^p_\mathcal{D}\}$$

```
Select *
FROM ExVP_knows|knows AS t1, ExVP_knows|age AS t2
...
```

Figure 7.10.: Translating an RDFPath query into SQL queries

Evaluation of Recursive Expressions. Recursions are one of the most challenging expressions in RDFPath. Their evaluation is based on a slightly-adapted version of the semi-naive algorithm introduced in Section 7.2.2 for TriAL-QL. First, we recall the definition of a path composition from Section 4.4, where the semantics of RDFPath were introduced. We noted in Definition 4.8 (page 72) that two RDFp paths p, q are composed if they are *compatible* thus iff it holds that $last(p) = first(q)$. We denoted this operation by the symbol $(. \circ .)$, where the result is again a composed RDFp path which conforms to our RDFPath semantics. Using this notation, we present our *semi-naive* evaluation strategy for recursive-patterns in Algorithm 7.4.1. Although introduced for recursive expressions, we used this algorithm for each RDFPath expression that represents a traversal step. The next section will introduces a few execution strategies for this algorithm.

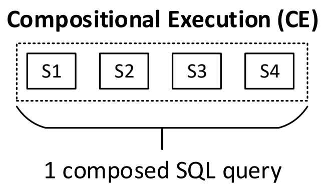
Algorithm 7.4.1 : Adapted Semi-naive evaluation of RDFPath *recursion*

```
input : RDFp graph $E$
1  $i \leftarrow 0$,
2  $\Delta P^0 \leftarrow E$
3  while $\Delta P^i \neq \emptyset$ do
4      $i \leftarrow i + 1$                          // Recursion until no paths derivable
5      foreach $p_i \in \Delta P^{i-1}$ do
6          foreach $q_j \in E$ do
7              if $last(p_i) = first(q_j)$ then
8                  $\Delta P^i \leftarrow \Delta P^i \cup (p_i \circ q_j)$       // Join between two RDFp graphs
9              end
10         end
11     end
12 end
13 return $P = \Delta P^0 \cup ... \cup \Delta P^i$   // Composition based on recursive expression
```

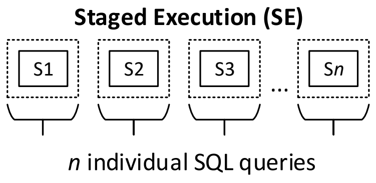
7.4.3. Query Processor

The evaluation of RDFPath queries that contain recursion involves an iterative process, where the number of required steps is not known in advance. Since neither Impala nor Spark natively support recursions in their SQL dialect, such expressions cannot be translated into single SQL statements but need to be broken down into several smaller queries that are executed subsequently. Each single SQL statement corresponds hereby to one iteration of the while loop (line 3) of the previously-shown Algorithm 7.4.1. For our TriAL Engine, we have already distinguished between a *materialized* and *compositional* execution. We will revise both strategies with respect to RDFPath and introduce a few optimizations that will be examined with experiments in Section 7.5.

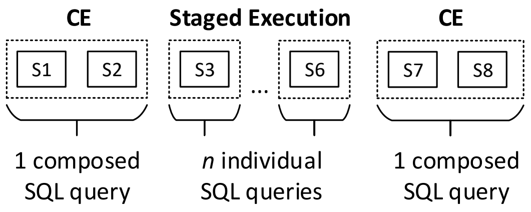
(1/4) Compositional Execution (CE). According to this strategy, a (non-recursive) expression in RDFPath is translated to one composed SQL query. Such a query can then be analyzed further by the optimizer of the respective SQL engine, i.e. Impala or Spark. However, in the case of long-chained or complex compositions, this approach might cause data to become spilled to disk during query evaluation. That happens if the size of intermediate results is underestimated by the optimizer and therefore a disadvantageous join strategy is chosen. In general, it is better to chose a *disk-based* join in advance rather than using an *in-memory* join which runs out of main-memory during its execution and needs then to spill data to disk. Hence, this strategy is well-suited for rather selective queries that are executed against a small subset of the RDFp graph. That way we can ensure that all joins are computed in-memory, which prevents costly disk operations and allow the SQL engine to make use of its optimizer. We can illustrate this strategy as shown in the following, where an RDFPath query is translated into four SQL statements, denoted by S_1, S_2, S_3, S_4, and evaluated in one composed SQL query.



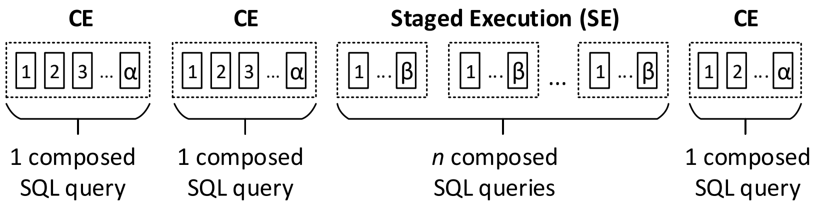
(2/4) Staged Execution (SE). This strategy is the first choice for queries which involve recursions, since it enables the iterative execution of single SQL statements till the desired termination condition, e.g. no new paths are derived any more, is fulfilled. However, it can also be applied on non-recursive RDFPath expressions that are composable in one query. In this case, non-recursive fragments also become split into single SQL statements and executed subsequently. This only makes sense when we know in advance that the amount of intermediate results might grow exponentially at some point. Further, we can differ between a *materialized* or *non-materialized* variant of a staged execution depending on the underlying SQL engine we use. For Impala, the intermediate result of each iteration has to be materialized on disk and read again as input for the next iteration as there is no support for preserving those intermediate tables in main-memory across multiple SQL queries. Spark supports the storage of intermediate results from individual SQL statements in main-memory, making them available for further processing. Thus, a subsequent SQL statement does not need to read the results from a previous SQL statement again from disk. Consequently, we can summarize that a *staged execution* strategy is the first choice for recursive patterns and is, in addition, well-suited to data-intensive, ETL-like queries that need to process large quantities of the input graph. We illustrate this strategy in the following, where an RDFPath query is translated into n SQL statements, evaluated individually:



(3/4) Hybrid Execution (HE). A hybrid execution combines the staged and compositional execution. In practice, one can assume RDFPath queries to be composed out of recursive and non-recursive fragments. For those queries, the staged execution would have been used so far. However, for selective queries, where we expect to get results in interactive time, it would be unfavorable to use a staged execution since it is (1) geared towards data-intensive tasks by means of disk-based operations and (2) uses more conservative query optimizations. To minimize the negative impact of a staged execution for selective queries, we use the hybrid execution which distinguishes between different fragments and executes recursive ones using the staged execution and non-recursive ones using the compositional strategy. As a result, for selective queries, the recursive fragment is still executed as a staged execution, but any non-recursive fragments are executed using the beneficial compositional execution. We can illustrate this strategy as shown in the following where, at the beginning and at the end, compositional executions are applied and in the middle a staged execution is applied.



(4/4) Hybrid α-β-Execution (HE+). As a further step towards a more flexible approach, we next introduce a generalization of our hybrid execution. The main idea behind this extension is a mechanism that allows the specification of the maximum amount of SQL statements which can be composed in *one* SQL query. Therefore we introduce two parameters that needs to be provided for this strategy, namely α and β. The first parameter, α, is a numeric value that specifies the number of statements allowed to be composed within a *compositional execution*. The second value β specifies the number of statements allowed to be composed within a *staged execution*. However, composing SQL statements from recursive fragments comes along with some problems, as previously discussed. Since the number of required joins is not known in advance, a β value higher than *one* might result in an overestimation, i.e. we are executing recursion more often than it would have been required. Nonetheless, in accordance with Algorithm 7.4.1, we obtain still, even in the case of an overestimation, the correct results; only the last $\Delta P^i s$ will be then empty. Following this notation, our previously-introduced hybrid execution can be seen as a special case of HE+, denoted by *Hybrid* $*$-1-*Execution*. We can read this as follows: an *unlimited* ($\alpha = *$) amount of SQL statements from *non-recursive* fragments is allowed to be composed into *one* query, but in the case of *recursive* fragments, each statement has to be executed individually (no composition) one by one ($\beta = 1$). We illustrate this strategy in the following, where at most α SQL statements are composed into one SQL query and at most β SQL statements are composed for recursions using staged execution.



7.5. Experiments on RDFPath Engine

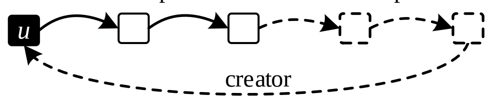
We evaluated the RDFPath Engine with three different execution strategies (Compositional, Staged and Hybrid α-β-Execution), two data storage layouts (VP and ExtVP), and two in-memory processing frameworks (Impala and Spark)[9]. Analogous to Section 7.3, where we performed experiments with the TriAL-QL Engine, we again used the *Social Network Benchmark* (SNB) data generator[10] to generate a synthetic social network of up to 1.8 billion triples (edges). The corresponding store sizes for four scaling factors (SF 1, SF 3, SF 10, SF 30) and three storing strategies are listed in Table 7.3.

Table 7.3.: SNB data with #triples and store sizes for different storage strategies

	Original		VP		ExtVP	
	Triples	**Size**	**Triples**	**Size**	**Triples**	**Size**
SF 1	59.5 M	4.1 GB	59.5 M	0.6 GB	82.3 M	1.7 GB
SF 3	176.4 M	12.3 GB	176.4 M	1.8 GB	243.8 M	5.2 GB
SF 10	594.7 M	41.4 GB	594.7 M	6.4 GB	825.2 M	19.3 GB
SF 30	1,799.4 M	125.5 GB	1,799.4 M	20.1 GB	1,458.7 M	30.5 GB

We have designed four representative use cases for the RDFPath Engine. Their different characteristics enable us to highlight some crucial features of RDFPath and demonstrate the performance and scalability of the implemented evaluation and storage strategies. Even though they are based upon the use cases introduced for TriAL-QL from Section 7.3, they exhibit, due to the different nature of both languages, significant differences. We therefore focus in this section on investigating the properties of the RDFPath Engine rather than providing a comparison with other systems. Only use case 4 will include execution times for TriAL-QL Engine. A comprehensive comparison with competitors is presented at the end of this dissertation in Chapter 8.

Use Case 1: Socialized Recommendations. The first use case asks for users which like similar topics as a given chosen user. For that, we start with a fixed user and follow the edges `knows` and `person` to determine a users friends. In a next step, a nested filter is used to find only those friends, which like at least one post of the initially random user. The structure of this query exhibits a pattern of the following shape:



[9]The cluster configuration can be found in Chapter 8 on page 195
[10]`http://ldbcouncil.org/developer/snb`

The corresponding RDFPath query is as follows:

```
%user% :/knows /person [/likes /post [/creator=%user%]]
```

We evaluated this query for 50 randomly chosen users. The mean runtimes are listed in Table 7.4. Figure 7.11 plots the mean runtimes in relation to the data size (scaling factors) on a log-log scale. The first result is that the compositional execution on Impala using ExtVP data layout shows overall the best performance. In comparison to the staged execution on Impala, we can further see that the larger the data, the smaller the difference between runtimes becomes. A better scaling behavior of staged executions on Impala in comparison to the compositional one is actually in line with our expectations, due to lower memory consumption and the usage of statistics computed for intermediate results. Furthermore, we can see that Impala (illustrated with continuous lines) performed faster than Spark in all cases (illustrated with dashed lines). Lastly, we shall discuss the impact of the data layout on the performance. In this regard, using ExtVP in comparison to VP lead to better runtimes in all experiments. In fact, using ExtVP does not cause additional costs for query evaluation, while smaller ExtVP partitions used as input reduce the overall workload. The measured performance benefit was, for all experiments, on average 21%. However, the initial loading phase of the graph (which has to be done just once in advance) is more costly and the size of the internal data store is due to data duplication 4 times larger (cf. Table 7.3.)

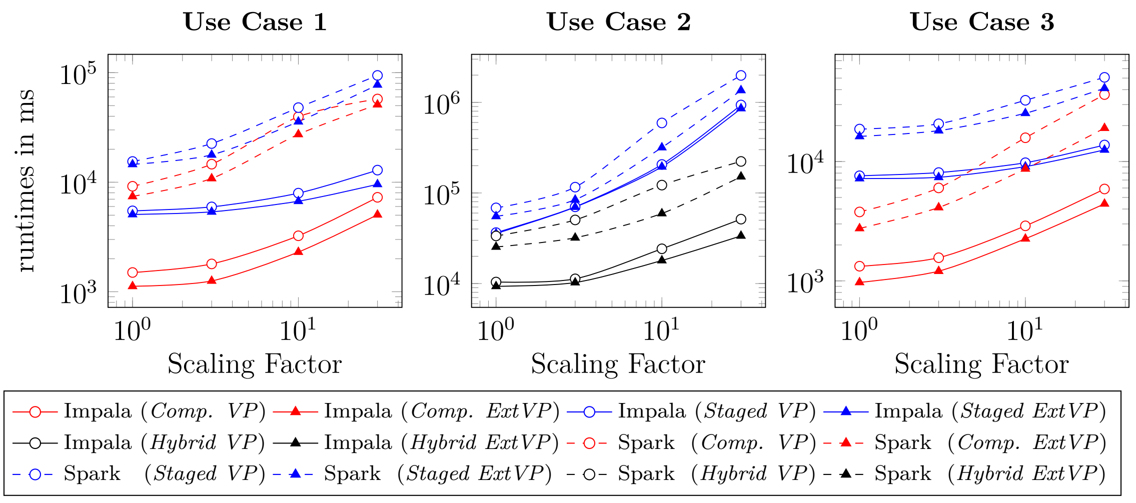
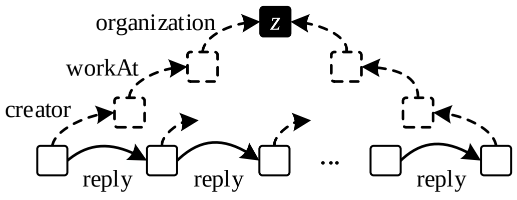


Figure 7.11.: Mean runtimes in relation to scaling factors on log-log scale. Dashed lines are SPARK, continuous lines Impala; circles mark VP, triangles mark ExtVP; red color stands for Compositional execution, blue for Staged, black for Hybrid

Use Case 2: Reachability. The second use case investigates the evaluation of recursive expressions. For that, we designed a query which traverses replies of posts recursively, where each creator of a post has to work in the same organization. The organization is a randomly-chosen resource, which is tested by a nested filter expression in each recursive step by means of three joins. The structure of this query exhibits a pattern of the following shape:

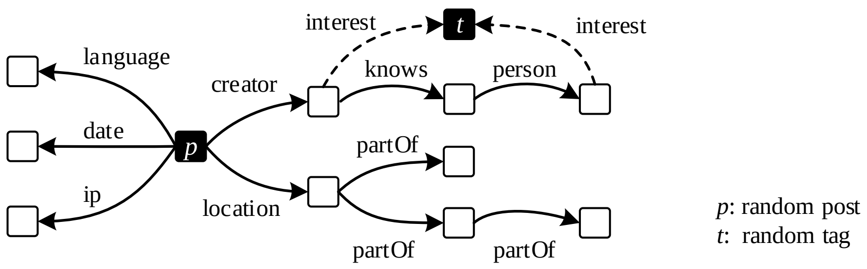


The corresponding RDFPath query is as follows:

```
* :/reply [/creator /workAt /organisation=%organization%] (*)
```

This unbounded query (it starts its evaluation for all nodes in the graph) follows the predicate `repliesOf`, which is one of the most-used predicates in the graph. Analogous to use case 1, we have chosen 50 random organizations used within the nested filter expression. The mean runtimes are listed in Table 7.4. Due to the recursion, a compositional execution strategy was not applicable. We therefore compare in Figure 7.11 the Hybrid α-β execution strategy (black-colored lines) with the staged one (blue colored lines). In order to capture the nested filter with a compositional subquery, we set $\alpha = 3$, and $\beta = 3$ for the hybrid strategy. Developing an algorithm which estimates best values for α and β from the structure of the query is part of future work. As a first result, we can see in Figure 7.11 that the hybrid execution outperformed the compositional one with Impala and Spark. The runtimes of Impala were again faster than those for Spark. The best execution strategy (Impala, Hybrid, ExtVP) required, on the smallest dataset, an average of 9.3 seconds and, on the largest dataset, 33.6 seconds, computing almost one million paths. In comparison, the slowest execution times obtained by the combination (Spark, Staged, VP) on SF 30 added up to 33 minutes. Using ExtVP instead of VP improved the execution time to only 22 minutes.

Use Case 3: Navigating Branches. The third use case covers the evaluation of a large variety of RDFPath expressions, including *branches with nesting*, *traversing steps specifying a set of predicates* and *bounded recursion.* The RDFPath query which we designed for that use case retrieves various information related to a given post and exhibits the following shape:



The corresponding RDFPath query is as follows:

```
%post% :/creator /( /interest [=%tag%] ||
                    /knows /person /interest [=%tag%])
         || /{language, date, ip}
         || /location /partOf(1,2)
```

It is a more complex but selective query involving many joins which we evaluated on 50 randomly-chosen posts and tags. The mean runtimes are listed in Table 7.4. Overall, we observed comparable results as discussed for the first use case. Figure 7.11 demonstrates once more the dominance of using Impala and ExtVP for the compositional execution of RDFPath queries. For the smallest data size (SF 1), we achieved runtimes below one second and for the largest dataset that contains about 1.8 billion triples, the evaluation finished in only 4.4 seconds. For Spark with compositional execution we can see a better performance than Impalas staged execution, for smaller data sizes. However, this reverses for larger data sets, due to better scaling behavior of Impalas staged execution.

Use Case 4: Connectivity. The last experiment asks for the paths between two given resources, which is a typical pattern in querying, e.g. social networks or biological information. Note that, in the case of RDFPath, we are not only verifying the existence of a connection but derive multiple paths containing *all* resources that were traversed to obtain a connection between two nodes. The following RDFPath query asks for the paths between two given users by following friendship relationships.

```
%user1% :/knows(*) /knows [=%user2%]
```

The mean runtimes for 50 randomly-chosen pairs of users are listed for RDFPath Engine and Trial-QL Engine in Table 7.4. We plotted the execution in relation to the data size (scaling factor) on a log-scale scale in Figure 7.12, where Trial-QL is illustrated with a brown color and RDFPath with blue and red. This plot demonstrates the possible negative impact of a path-based semantic in comparison to an existential one. The scaling behavior of all Trial-QL executions (brown colored) is described by a rather linearly-growing curve, due to the fixed number of results for each pair. In contrast, we can see for all RDFPath executions, although

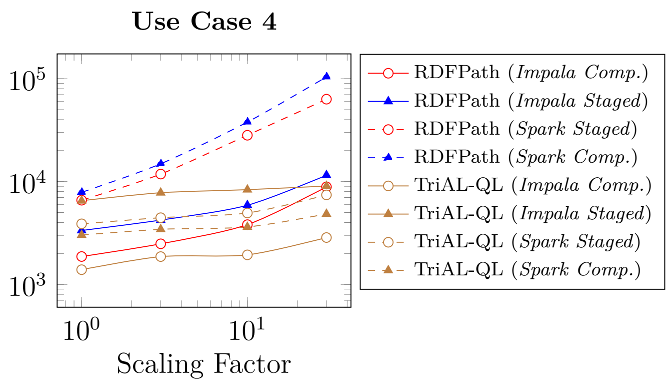


Figure 7.12.: Mean runtimes in relation to scaling factors on log-log scale. Dashed lines are SPARK, continuous lines Impala; TriAL-Engine executions are brown-colored, blue and red are used for RDFPath Engine

they exhibit a competitive performance, a much worse scaling behavior which is an expected result for this sort of query. Nonetheless we obtained, with the RDFPath Engine using compositional execution on Impala, mean runtimes of 1.9 seconds on SF 1 and 8.9 seconds on SF 30. In the case of Trial-QL (compositional on Impala) the runtimes where slightly better, i.e. 1.4 seconds on SF 1 and 2.9 on SF 30. However, as we will discuss in a more comprehensive comparison presented in Chapter 8, not many cases exist in which the RDFPath Engine outperforms Trial-QL Engine.

Summary. The four presented use cases have shown that for evaluating RDFPath queries the combination of using Impala with compositional execution and ExtVP data model turns out to perform best. However, that this does not necessarily need to be the case for all sorts of queries was indicated by having a look at the respective scaling behavior of the execution strategies. The staged execution revealed to scale much better while increasing the data size, which means that for, e.g. more data intensive queries or longer paths, there might exist reversal points where a staged execution is more beneficial.

The more interesting and also somewhat surprising findings of the experiments performed with RDFPath Engine are the obtained runtimes. With regard to the theoretical upper bound for the amount of derivable paths by RDFPath, which we discussed in Section 4.5.2 (page 81), we achieved across all experiments excellent runtimes. Even on a social network dataset with almost 1.8 billion triples, we obtained for all our use cases *interactive* query runtimes, i.e. on the order of seconds. These results will also be confirmed in the next chapter, where we used a benchmark for comparing the performance of our implementations against various competitors.

Table 7.4.: Mean runtimes (in ms) for Use Cases 1 – 4

		Configuration			**SF 1**	**SF 3**	**SF 10**	**SF 30**
Use Case 1		**Impala**	Comp.	VP	1499	1795	3243	7278
		Impala	Comp.	ExtVP	1122	1256	2304	5048
		Impala	Staged	VP	5477	5953	7947	12890
		Impala	Staged	ExtVP	5077	5362	6707	9567
		Spark	Comp.	VP	9177	14605	39580	57465
		Spark	Comp.	ExtVP	7421	10820	27328	51077
		Spark	Staged	VP	15458	22535	47936	94307
		Spark	Staged	ExtVP	14555	17773	35700	77404
		Results			81	16	61	64
Use Case 2		**Impala**	Hybrid	VP	10389	11270	24186	51389
		Impala	Hybrid	ExtVP	9327	10257	17964	33596
		Impala	Staged	VP	36510	70272	206270	935980
		Impala	Staged	ExtVP	35666	69842	194581	856772
		Spark	Hybrid	VP	33553	50378	122090	222836
		Spark	Hybrid	ExtVP	25359	32018	59385	151426
		Spark	Staged	VP	68451	115909	592743	1983742
		Spark	Staged	ExtVP	55203	84233	317586	1357169
		Results			26124	85324	309253	994337
Use Case 3		**Impala**	Comp.	VP	1327	1564	2892	5887
		Impala	Comp.	ExtVP	972	1207	2255	4424
		Impala	Staged	VP	7599	8081	9771	13758
		Impala	Staged	ExtVP	7208	7380	9073	12523
		Spark	Comp.	VP	3781	6010	15813	36378
		Spark	Comp.	ExtVP	2753	4121	8683	19054
		Spark	Staged	VP	18708	20698	32586	50723
		Spark	Staged	ExtVP	16243	18149	25393	41118
		Results			44	9	24	12
Use Case 4	RDFPath	**Impala**	Comp.	ExtVP	1868	2483	3814	8883
		Impala	Staged	ExtVP	3347	4224	5885	11532
		Spark	Staged	ExtVP	6589	11827	28215	63276
		Spark	Comp.	ExtVP	7861	14893	37908	104146
		Results			24	43	30	51
	TriAL-QL	**Impala**	Comp.	VP	1391	1865	1939	2857
		Impala	Staged	VP	6515	7794	8345	9057
		Spark	Staged	VP	3885	4454	4950	7422
		Spark	Comp.	VP	3021	3436	3620	4845
		Distance			2,9	3,1	3,0	3,2

8. Experiments

Contents

The evaluation in this chapter sums ups the work presented in this dissertation by providing a comprehensive performance overview with various competitive query engines. Most of the experiments were performed on a small cluster with ten machines, each equipped with a six core Xeon E5-2420 CPU, 2×2 TB disks and 32 GB of main memory. We used the Hadoop distribution of Cloudera CDH 5.7.0 with Impala 2.5.0 and Spark 1.6.2. The machines were connected via Gigabit network. This is actually a rather low-end configuration as Cloudera recommends 256 GB RAM and 12 disks or more for Impala nodes in a production cluster. Both the Impala daemon and Spark executor were using 24 GB of main memory, broadcast joins were disabled, and the Parquet filter push-down optimization were enabled. For competitors executed on a single machine we used a workstation equipped with a Intel Xeon E5-2640 CPU with six cores, twelve threads, 192 GB main memory and 2×1 TB disks.

We based our experiments on the *Waterloo SPARQL Diversity Test Suite* [AHOD14] (WatDiv), which provides a test environment for RDF data management systems with more diverse workloads than other benchmarks. We generated datasets from ten million to a billion RDF triples using the WatDiv data generator with scaling factors 100, 1000, and 10000. Since we focus in this dissertation on path-based queries, we used the *Incremental Linear Testing* use case which focus on path-shaped patterns. It consists of three query types (IL-1, IL-2, IL-3) which are bound by user, retailer or unbounded, respectively. Each query starts with a length of 5 (IL-1-5) and becomes incrementally increased up to 8 (IL-1-8).

We compare the TriAL-QL Engine from Chapter 7.2 and RDFPath Engine introduced in Chapter 7.5 with six competitive RDF Management systems and one graph database. Since the original WatDiv queries are written in SPARQL, we translated them in the languages supported by the respective system. We adapted the queries with care and listed them all in Appendix D. As two representative SPARQL-query-processors that use MapReduce we chose Shard [RS11] and PigSPARQL [SPL11]. Shard is written directly in MapReduce, where each triple pattern is mapped to exactly one reduce-side-side-join, comparable to our RDFPath MapReduce Processor. Data is stored in HDFS, where triples are grouped by their subjects and put together in one line. PigSPARQL is a SPARQL query processor which, instead of a direct mapping, translates into Pig Latin as an intermediate layer between MapReduce. Data is stored vertically-partitioned in HDFS. With H2RDF+ [PKT+13], we have one representative SPARQL engine built on top of a NoSQL store. H2RDF+ uses HBase to store triples sorted by row keys in six different triple permutations. Based on the selectivity of a triple pattern, queries are either executed on a single node or distributed using MapReduce. Two representative SPARQL engines that utilize in-memory processing frameworks are Sempala [SPNL14] and S2RDF [SPSL16]. Sempala is built on top of Impala. Its RDF data layout is highly optimized for star-shaped queries and enables interactive querying times on large RDF graphs. S2RDF is a fast SPARQL-on-Hadoop engine built on top of Spark SQL. It stores its data using Extended Vertical Partitioning (ExtVP), which efficiently minimizes the query input size regardless of the query pattern shape and diameter. Virtuoso Open Source Edition v7.1.1 [EM10] represents a state-of-the-art centralized RDF data management system using a relational database to store data. It supports SPARQL 1.1, OWL reasoning, and benefits from indexes and a two-level compression strategy optimized for RDF. As a last competitor we chose the graph database system Neo4j [NT16]. Neo4j, of which we used version 3.0.6, is one of the most prominent native graph-databases running on a single-machine [VWA+15]. Data representation is based on the *Labeled Property Graph* model, which consists of entities (nodes) and relationships (labeled edges). A connection between two entities is then represented by a directed and named relationship. While loading the WatDiv data into Neo4j, we modeled an RDF triple (s, p, o) of entities s and o connected by a relationship p. Here it was crucial to ensure that two identical IRIs (only subjects and objects) refer to the same node in the property graph. We skipped literals, since they are not required for the *Incremental Linear Testing* use case, although they could have been easily represented by so-called attributes. Further, we translated the WatDiV queries into Cypher, Neo4j's graph query language, which is a declarative, pattern-matching language comparable to regular path queries. The translated queries are shown in Appendix D.4 (page 238).

Discussion on Store Sizes. The store sizes for all three generated datasets are listed in Table 8.1. We can see that the store sizes of both our engines are significantly smaller than the size of the original RDF graph. This is achieved by Parquets built-in support for run-length and dictionary encoding in combination with snappy compression that perform great for storing RDF in a column-oriented format. Only in the case of the RDFPath with ExtVP can we observe a higher graph size. However, here we need to recall that ExtVP adds, in accordance with Definition 7.2 (page 182), *three* additional partitions, namely $ExtVP^{OS}_{p_1|p_2}[G]$, $ExtVP^{OS}[G]$, and $ExtVP^{PS}_{p_1|p_2}[G]$. That factor is also represented in the total sizes, which are about four times larger in comparison to RDFPath with VP. The largest store sizes are created by Neo4j's *Labeled Property Graph* model, which was also reflected in its loading times. Loading the smallest dataset took 12 hours and the largest one ten days. The main workload was therefore the insertion of nodes and the updating of their indexes.

Table 8.1.: WatDiv load times and HDFS sizes

		SF100	**SF1000**	**SF10000**
tuples	**original**	10.91 M	109.2 M	1091.5 M
	TriAL-QL VP	10.91 M	109.2 M	1091.5 M
	RDFPath VP	10.91 M	109.2 M	1091.5 M
	RDFPath ExtVP	34.32 M	462.5 M	2453.0 M
	S2RDF VP	10.91 M	109.2 M	1091.5 M
	S2RDF ExtVP	119.94 M	1197.9 M	11967 M
	Neo4j Graph	9.58 M	95.99 M	959.4 M
HDFS size	**original**	507 MB	5.3 GB	54.9 GB
	TriAL-QL VP	103 MB	1.2 GB	13.2 GB
	RDFPath VP	103 MB	1.2 GB	13.2 GB
	RDFPath ExtVP	475 MB	4.8 GB	42.4 GB
	S2RDF VP	82 MB	0.6 GB	6.6 GB
	S2RDF ExtVP	914 MB	6.2 GB	63.7 GB
	H2RDF+	517 MB	5.2 GB	57.0 GB
	Sempala	249 MB	3.5 GB	40.4 GB
	PigSPARQL	871 MB	8.9 GB	92.5 GB
	SHARD	981 MB	9.9 GB	100 GB
	Neo4j Graph	4425 MB	50.9 GB	536.7 GB

The first query type (IL-1) describes a path-shaped pattern which starts at a random users and subsequently follows various edges. Two of them are `friendOf` and `follows`, which are commonly-used predicates, together representing about 70% of the graph. A comparison between the runtimes of all systems on three data sizes for IL-1 queries is shown in Table 8.2 (page 204). The second query type (IL-2) has the same structure as the first one but starts at a random retailer and again subsequently follows various edges. All IL-2 query runtimes are listed in Table 8.3 (page 205). The last query type (IL-3) also exhibits a path-shaped pattern, however, in an unbounded query. That means the evaluation starts from all nodes in the graph and produces a large amount of intermediate results, which puts a heavy load on all compared systems. The respective runtimes for IL-3 queries are shown in Table 8.4 (page 206). All three tables are shown at the end of this chapter.

Performance of RDFPath and TriAL-QL. Figure 8.1 illustrates the runtime differences between all benchmarked systems in a log-scaled bar chart. The first result is that our RDFPath Engine executed on Impala and using the ExtVP data model outperforms all competitors. Since both of our engines which use VP instead of ExtVP are slower, we can attribute their lead to the usage of the ExtVP data model. Indeed, by using the corresponding ExtVP partitions, the input size for the two predicates `friendOf` and `follows` was reduced to 1/3 of the respective VP

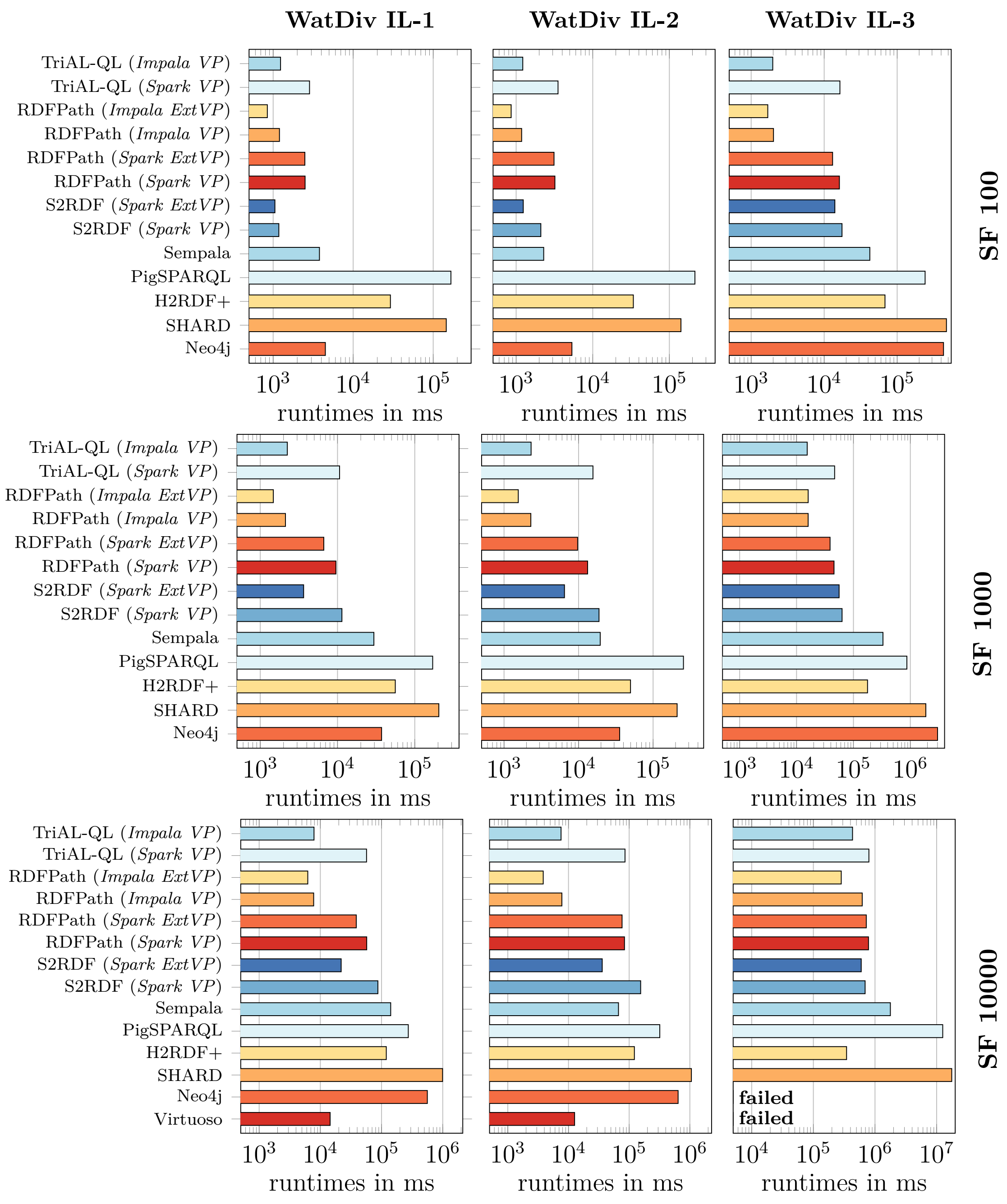


Figure 8.1.: Comparison of mean runtimes on a log scale .

partitions used by RDFPath (Impala, VP) and TriAL-QL (Impala, VP). We have also performed some experiments with TriAL-QL and ExtVP with very promising results. However, TriAL-QL enables joins over all three variables (all variants inherent to the triple-based model) and in addition allows to modify *s, p, o* in an arbitrary way. As a result, the cost-benefit ratio was not satisfying, since many additional partitions were required to be precomputed and stored. Nonetheless, incorporating ExtVP tables into our TriAL-QL Engine remains part of our future work.

Regarding individual runtimes, we want to emphasize that for IL-3 on SF 10000, which produces more than 25 billion results, RDFPath required less than 10 minutes and TriAL-QL only 23 minutes to finish. In comparison, the slowest execution, which was PigSPARQL on MapReduce, lasts for more than 11 hours and the best competitor, S2RDF, required about 34 minutes to compute its results. Both single-machine engines, Neo4j and Virtuoso, were not able evaluate this query on the largest dataset using a time constraint of 24 hours. The other range of execution times is also worth noting. For IL-1 and IL-2, the runtimes for RDFPath are below one second on SF 100, which contains 11 million triples. The runtimes of TriAL-QL were, with 1.2 seconds for both query types, only marginally slower. On SF 10000, which is a graph with over one billion triples, RDFPath finished its execution in 3.8 seconds for Il-2 and 6.2 seconds for Il-1. TriAL-QL was again only slightly slower and required 7.5 seconds for IL-2 and 7.8 seconds for IL-1.

Existential Semantics in Distributed Frameworks. One would assume that the advantageous of an existential semantics, as used with TriAL-QL, lead to much better performance characteristics than the path-based semantics introduced for RDFPath. After all, the expected smaller amount of results confront with storing possibly manifold long paths. However, the results for the first two query types (IL-1 and IL-2) are in the order of ten thousand, where the impact of fewer and smaller intermediate results, in comparison to the paths of RDFPath, does not affect the runtimes. Moreover, we have observed that the costs of ensuring distinct results are in many cases significantly higher than their benefits. In fact, the DISTINCT operation is particularly costly in distributed frameworks such as Impala and SPARK, since it requires the repartitioning of data across all machines. A few additional experiments confirmed that queries exist for which allowing duplicates in TriAL-QL, i.e. removing all DISTINCT operations, improved the overall execution times. For the data-intensive query IL-3 with billions of results, we see that the execution times for RDFPath (Impala, ExtVP), RDFPath (Impala, VP), and TriAL-QL (Impala, VP) are much closer. Still, RDFPath with ExtVP performs best, but if we compare the execution times of RDFPath (Impala, VP) with TriAL-QL (Impala, VP), we can see that TriAL-QL performs faster. In such cases, with intermediate results in order of hundreds of millions to billions, the lower amount of intermediate results compensates the costs of the distinct operations. We can further expect that TriAL-QL along with ExtVP, which is part of potential future work, would perform better than RDFPath for that case.

Spark vs. Impala. Next, we can see that our engines perform significantly better using Impala rather than Spark, which is in line with our previous experimental observations in Section 7.3 and Section 7.5. This fact can be more easily grasped by observing Figure 8.2, which plots the mean runtimes of all systems in relation to the scaling factor using a log-log scale. All five Spark-based engines (including the competitors') are highlighted with *dashed lines* making it easier to group them together. Their deviation with regard to runtimes and scaling behavior is rather low, which is particularly true for the data-intensive IL-3 queries. One reason for the better performance of Impala are our proposed algorithms and storage strategies that are closely related to relational algebra, which in turn is the core component of Impala. Moreover, Impala is a pure SQL-engine, whereas Spark is a general-purpose execution framework with support for many other querying interfaces. Overall we can conclude, that the synergy effects of using Impala together the proposed evaluation and storage strategies are a good fit for the examined queries. Nonetheless, as highlighted in Section 7.3 both, Impala and Spark, have their strengths and weaknesses while being continuously improved. Future Spark versions might therefore reveal other performance characteristics.

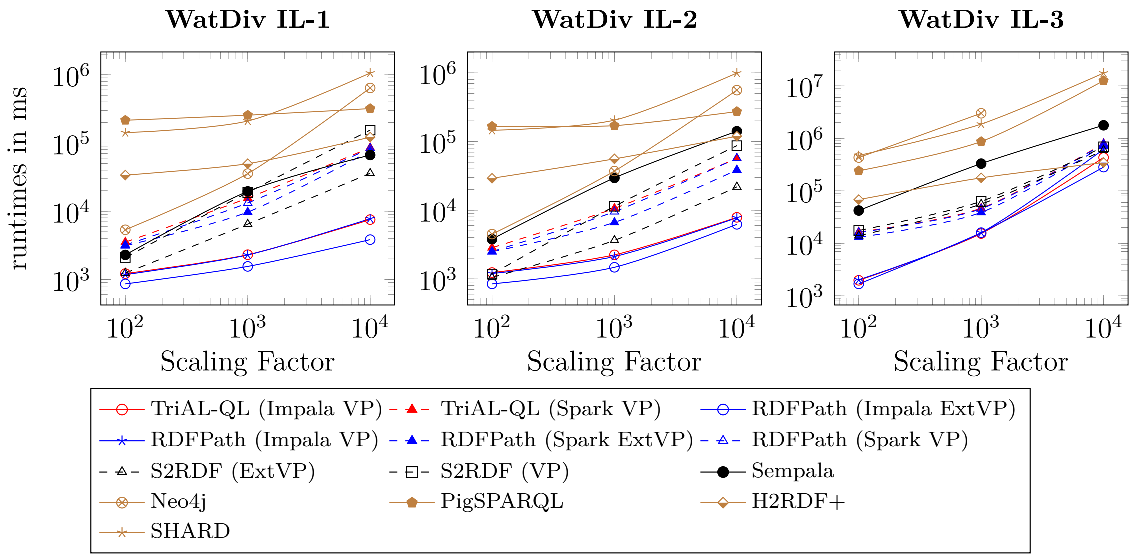


Figure 8.2.: Mean runtimes in relation to scaling factors on log-log scale.

Impact of Evaluation Strategies and Optimizations. To emphasize that these good runtimes are not only derived from the underlying frameworks but are the result of well-designed evaluation strategies and optimizations for path-shaped patterns, we next compare all RDF management systems which use the same underlying technology, i.e. Impala, Spark and Parquet. That are four systems (RDFPath, TriAL-QL, S2RDF, and Sempala) in different configurations, making up a total of nine systems. In Figure 8.1, the first nine systems (from top to down) correspond to exactly these systems. We divided them for the following analysis into two groups, for which we calculate the ratio between mean runtimes.

Group 1 :	TriAL-QL (Impala VP), (Spark VP)
	RDFPath (Impala ExtVP), (Impala VP), (Spark ExtVP), (Spark VP)
Group 2 :	S2RDF (Spark ExtVP), (Spark VP)
	Sempala (Impala)

First, we averaged the execution times for all systems within a group by query type and data size. As a result we obtained in total 18 averaged runtimes (for each group, each query type, each data size). We then computed the ratios between both groups, where in 8 out of 9 cases *group one* proved to be faster than *group two*. In a final step, we computed the mean ratio for all query types and data sizes. We determined as a result that *group one* was on average 2.3 times faster than *group two*. This factor supports the benefit of the proposed evaluation strategies and optimizations for path-shaped patterns. At the same time we also need to limit these results with regard to transferability to other query types. We used a benchmark that focuses on path-shaped patterns for which our engines in group one were explicitly designed. By comparison, systems in group two are developed to cope with all kinds of patterns. Using a benchmark that also contains other types of pattern, for instance star- or snowflakes-shaped, we expect the systems in group two to perform significantly better than ours.

We can further observe that the performance benefit of using ExtVP in comparison to VP with RDFPath is significantly lower as it is the case for S2RDF. We can contribute this to mainly two reasons. First of all, the use of well-designed evaluation strategies reduces already the overall amount of intermediate results substantially. As a result, the reduced input size of ExtVP executions in comparison to VP does not have such a strong impact on the performance as it has for S2RDF. Secondly, we are not using all possibility ExtVP partitions since many of them are not useful for path-shaped queries. Particularly the partition $ExtVP^{OS}_{p_1|p_2}[G]$ (cf. Definition 7.2 on 182) which we left out might have an additional positive effect on the performance of RDFPath with ExtVP[1].

Comparison with MapReduce Engines. Next we have a closer look at both MapReduce engines (Shard and PigSPARQL), which are clearly outperformed by all other systems. This is actually not a surprising result due to the batch-oriented workflow of MapReduce and the usage of disk-based operations. On the positive side we can note that all queries were executable even on the largest datasets, whereas single-machine approaches such es Neo4j and Virtuoso failed on the heavy-load query (IL-3). Due to their disk-based operators, we expect that these systems work also on much larger datasets, whereas our approaches based on in-memory frameworks

[1]Preliminary experiments have shown only a small performance improvement in using an additional $ExtVP^{OS}_{p_1|p_2}[G]$ partition that does not justify its high storage costs.

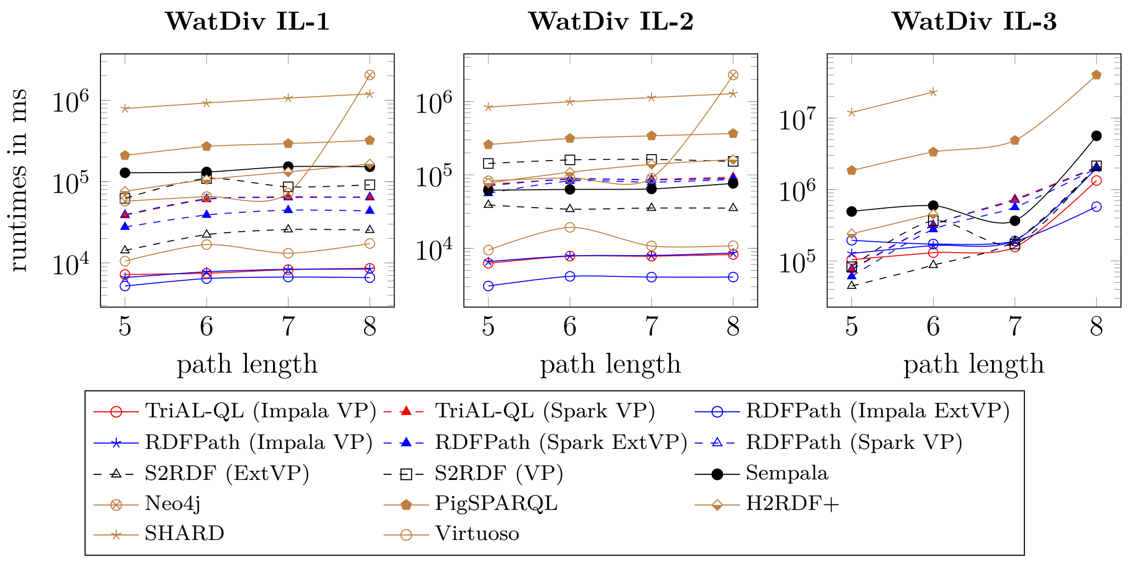


Figure 8.3.: Mean runtimes in relation to increasing path length (SF 10000).

are expected to fail once they run out of main memory. There is also support for disk-based operations in Impala and Spark, which are significantly slower than their respective in-memory version but at least ensure the execution without running out of memory. However, this has to be specified in advance while compiling a query. During query execution, there is only support to perform so-called spilling to disk, which massively decreases the overall performance and is only applicable in cases where just a relatively small amount of data needs to be moved to disk. Furthermore, the results for the MapReduce-based engines confirmed our decision to omit the RDFPath MapReduce Processor introduced in Chapter 6 in these experiments, which allows us to put a stronger focus on in-memory frameworks.

Impact of Path Length. Figure 8.3 demonstrates the impact path length has on the runtimes by plotting the path length on the x-axis. Please note, that only the y-axis is scaled logarithmically. For queries with a fixed start node (IL-1 and IL-2), for our engines we can observe a linear increase of the runtime with respect to path length. In the case of the unbounded IL-3 queries, we can observe an exponential growth of the runtimes with respect to the path length. However, this is an expected result for such queries, due to the massive amount of produced results which are, e.g. over 25 billion for the largest data size.

Comparison with Neo4j and Virtuoso. Both single machine frameworks, Neo4j and Virtuoso, once more confirmed our arguments used to motivate a distributed engine. Their performance was, up to a certain point, quite competitive. Virtuoso performed better on IL-1 and IL-2, for instance, than systems using Spark. Also the performance of Neo4j was competitive for selective queries on the smallest data size. However, even if equipped with the same amount of main memory as the total available for machines in the cluster together, the performance decreases for data-intensive tasks, and the execution fails at some point. Beyond the benefits of distributing the workload on a cluster of machines, we also have to note that Neo4j is not designed for the kind of workload which comes along with querying RDF. In fact, its strength is in its almost-immediate response for short and precise queries on the graph structure, for instance, rather than data-intensive subgraph-matching tasks, which produce thousands of results.

8.1. Summary

Overall, the evaluation clearly demonstrates that distributed frameworks such as Impala and Spark, combined with proper evaluation strategies and good data storage, provide an excellent basis for both of our navigational query languages, RDFPath and TriAL-QL. In their best respective configurations, TriAL-QL Engine and RDFPath Engine outperformed all evaluated competitors. A more fine-grained look at the results revealed that the combination of Impala with the ExtVP data model produces the best results. For selective queries, both engines exhibit runtimes in the order of a few seconds, and in the best case even less than one second. On data-intensive queries, which produced more than 25 billion results, we obtained runtimes in the range of 10 to 25 minutes. In that sense, we can conclude that both of our RDF management systems proposed in this dissertation, RDFPath Engine and TriAL-QL Engine, are a significant step towards processing navigational query languages on web-scale RDF data using Hadoop, with support for both interactive querying and data-intensive workload.

Table 8.2.: Mean runtimes (in ms) for WatDiv Incremental Linear Testing 1 (IL-1)

	Query Engine	**Query:**	**IL-1-5**	**IL-1-6**	**IL-1-7**	**IL-1-8**	**Mean**
SF 100	**TriAL-QL**	Impala, VP	1112	1232	1241	1363	1237
	TriAL-QL	Spark, VP	2429	2815	3028	3126	2849
	RDFPath	Impala, ExtVP	719	811	881	971	846
	RDFPath	Impala, VP	1094	1137	1221	1345	1199
	RDFPath	Spark, ExtVP	2129	2457	2683	2696	2491
	RDFPath	Spark, VP	2096	2361	2700	2872	2507
	S2RDF	Spark, ExtVP	845	1058	1067	1240	1052
	S2RDF	Spark, VP	751	1268	1337	1354	1178
	Neo4j	Graph DB	969	2457	2666	11877	4492
	Sempala	Impala	3643	3753	3844	3919	3790
	PigSPARQL	MapReduce	128666	153755	178091	205638	166537
	H2RDF+	HBase	23498	27949	36826	28524	29199
	SHARD	MapReduce	118494	136505	154412	173364	145694
	# Results		10459	15706	15706	1883	10939
SF 1000	**TriAL-QL**	Impala, VP	1920	2182	2363	2493	2240
	TriAL-QL	Spark, VP	7877	10892	11348	12427	10636
	RDFPath	Impala, ExtVP	1234	1384	1588	1698	1476
	RDFPath	Impala, VP	1751	2065	2238	2424	2119
	RDFPath	Spark, ExtVP	5922	5988	7245	7434	6647
	RDFPath	Spark, VP	8016	9374	9853	10915	9540
	S2RDF	Spark, ExtVP	3324	3282	3698	4270	3643
	S2RDF	Spark, VP	9231	11098	12892	12399	11405
	Neo4j	Graph DB	8718	12311	10938	116822	37197
	Sempala	Impala	29321	29684	29595	29696	29574
	PigSPARQL	MapReduce	132130	163757	181044	205225	170539
	H2RDF+	HBase	25351	48957	78117	71448	55968
	SHARD	MapReduce	167910	189782	217487	244575	204938
	# Results		14131	21127	21127	2423	14702
SF 10000	**TriAL-QL**	Impala, VP	7201	7477	8261	8585	7881
	TriAL-QL	Spark, VP	38498	60383	64995	63663	56885
	RDFPath	Impala, ExtVP	5201	6427	6703	6603	6233
	RDFPath	Impala, VP	6601	7778	8336	8396	7778
	RDFPath	Spark, ExtVP	27654	38872	44537	43586	38662
	RDFPath	Spark, VP	39171	61299	63801	64674	57236
	S2RDF	Spark, ExtVP	14224	22257	25752	25327	21890
	S2RDF	Spark, VP	62158	109017	86189	91332	87174
	Neo4j	Graph DB	57467	65452	72120	2046517	560389
	Sempala	Impala	128486	131304	152730	152169	141172
	PigSPARQL	MapReduce	209594	270757	293241	321021	273653
	H2RDF+	HBase	76284	105794	131672	164583	119583
	SHARD	MapReduce	792204	925542	1064010	1195541	994324
	Virtuoso	RDFStore, cold	46998	77903	74664	82471	70509
	Virtuoso	RDFStore, mean	10529	16796	13159	17320	14451
	# Results		12314	18627	18627	6094	13915

Table 8.3.: Mean runtimes (in ms) for WatDiv Incremental Linear Testing 2 (IL-2)

	Query Engine	Query:	IL-2-5	IL-2-6	IL-2-7	IL-2-8	Mean
SF 100	**TriAL-QL**	Impala, VP	1026	1254	1306	1299	1221
	TriAL-QL	Spark, VP	3154	3441	3605	3894	3523
	RDFPath	Impala, ExtVP	698	794	946	1005	861
	RDFPath	Impala, VP	1131	1214	1156	1217	1180
	RDFPath	Spark, ExtVP	2637	3272	3382	3199	3123
	RDFPath	Spark, VP	2799	3234	3408	3386	3207
	S2RDF	Spark, ExtVP	1324	1038	1386	1209	1239
	S2RDF	Spark, VP	1812	1953	2229	2412	2101
	Neo4j	Graph DB	964	3369	3964	13092	5347
	Sempala	Impala	2164	2257	2290	2450	2290
	PigSPARQL	MapReduce	167104	209958	230734	254635	215608
	H2RDF+	HBase	23463	33042	44801	34200	33877
	SHARD	MapReduce	117176	130308	148463	170810	141689
	# Results		15404	17241	1018	792	8614
SF 1000	**TriAL-QL**	Impala, VP	2066	2345	2388	2428	2307
	TriAL-QL	Spark, VP	13963	15847	15683	16921	15603
	RDFPath	Impala, ExtVP	1273	1584	1646	1710	1553
	RDFPath	Impala, VP	2046	2304	2364	2442	2289
	RDFPath	Spark, ExtVP	8517	10038	9673	10633	9715
	RDFPath	Spark, VP	11692	13231	13976	13988	13222
	S2RDF	Spark, ExtVP	7592	5796	5980	6510	6470
	S2RDF	Spark, VP	16985	18646	19115	20524	18818
	Neo4j	Graph DB	5539	7573	8227	121077	35604
	Sempala	Impala	19357	19388	19496	19867	19527
	PigSPARQL	MapReduce	198302	252621	269446	301519	255472
	H2RDF+	HBase	25090	49745	50514	73750	49775
	SHARD	MapReduce	167761	196993	223118	252725	210149
	# Results		11890	13377	735	499	6625
SF 10000	**TriAL-QL**	Impala, VP	6260	7835	7823	8253	7543
	TriAL-QL	Spark, VP	74215	87070	86889	93391	85391
	RDFPath	Impala, ExtVP	3069	4157	4059	4060	3836
	RDFPath	Impala, VP	6607	7888	7992	8636	7781
	RDFPath	Spark, ExtVP	56227	82729	79801	86858	76404
	RDFPath	Spark, VP	71330	88126	85734	89632	83705
	S2RDF	Spark, ExtVP	39006	34200	35439	35296	35985
	S2RDF	Spark, VP	143105	160157	162595	152724	154645
	Neo4j	Graph DB	83276	91341	90199	2296593	640352
	Sempala	Impala	61843	63501	64487	76717	66637
	PigSPARQL	MapReduce	258307	313681	340580	365995	319641
	H2RDF+	HBase	77567	108780	139282	161913	121886
	SHARD	MapReduce	837829	992373	1131621	1278385	1060052
	Virtuoso	RDFStore, cold	74014	167892	78311	81350	100392
	Virtuoso	RDFStore, mean	9470	19314	10775	10870	12607
	# Results		14121	15919	1893	806	8185

Table 8.4.: Mean runtimes (in ms) for WatDiv Incremental Linear Testing 3 (IL-3)

	Query Engine	Query:	IL-3-5	IL-3-6	IL-3-7	IL-3-8	Mean
SF 100	**TriAL-QL**	Impala, VP	1564	1399	1763	3176	1976
	TriAL-QL	Spark, VP	5682	14326	19492	26237	16434
	RDFPath	Impala, ExtVP	1624	1257	1528	2346	1689
	RDFPath	Impala, VP	1644	1469	1780	3194	2022
	RDFPath	Spark, ExtVP	6417	10201	14426	21133	13044
	RDFPath	Spark, VP	5541	13822	19468	25850	16170
	S2RDF	Spark, ExtVP	3991	5511	3295	43231	14007
	S2RDF	Spark, VP	3327	14907	4449	47561	17561
	Neo4j	Graph DB	151507	625905	63528	883935	431219
	Sempala	Impala	19214	26118	11818	111690	42210
	PigSPARQL	MapReduce	164752	218817	236235	345529	241333
	H2RDF+	HBase	40956	70453	92907	F	68105
	SHARD	MapReduce	205248	312420	325451	1048829	472987
	# Results		31 M	35 M	5 M	157 M	57 M
SF 1000	**TriAL-QL**	Impala, VP	10967	8892	12135	29545	15385
	TriAL-QL	Spark, VP	18382	37486	61932	69029	46707
	RDFPath	Impala, ExtVP	10645	12045	14448	26873	16003
	RDFPath	Impala, VP	11589	9289	13080	29989	15987
	RDFPath	Spark, ExtVP	10384	35848	50594	58494	38830
	RDFPath	Spark, VP	13837	38622	63074	66740	45568
	S2RDF	Spark, ExtVP	20999	43078	24060	135839	55994
	S2RDF	Spark, VP	28120	41716	35812	146945	63148
	Neo4j	Graph DB	1787000	6499543	738273	F	3008272
	Sempala	Impala	155298	194758	93424	878232	330428
	PigSPARQL	MapReduce	362172	571965	622899	1924061	870274
	H2RDF+	HBase	121396	183752	225669	F	176939
	SHARD	MapReduce	1323657	2423349	F	F	1873503
	# Results		365 M	410 M	58 M	1894 M	682 M
SF 10000	**TriAL-QL**	Impala, VP	104454	130703	156315	1336902	432094
	TriAL-QL	Spark, VP	75166	317200	736273	2057158	796449
	RDFPath	Impala, ExtVP	194929	172707	190345	575868	283462
	RDFPath	Impala, VP	127500	164865	193638	1995604	620402
	RDFPath	Spark, ExtVP	60852	276829	564839	1986184	722176
	RDFPath	Spark, VP	76758	318651	711897	2029205	784128
	S2RDF	Spark, ExtVP	44629	87437	190863	2068372	597825
	S2RDF	Spark, VP	83119	364865	172136	2144944	691266
	Neo4j	Graph DB	F	F	F	F	N/A
	Sempala	Impala	493016	595152	365868	5649620	1775914
	PigSPARQL	MapReduce	1847039	3353907	4876005	40140420	12554343
	H2RDF+	HBase	240339	451390	F	F	345865
	SHARD	MapReduce	11995677	23164293	F	F	17579985
	Virtuoso	RDFStore, cold	F	F	F	F	N/A
	Virtuoso	RDFStore, mean	F	F	F	F	N/A
	# Results		3431 M	3874 M	890 M	25179 M	8343 M

Part V.

Discussion

9. Conclusion

Contents

We finish this dissertation with a discussion of TriAL-QL and RDFPath that wraps up both languages and their implementations. We continue in Section 9.2 with a summary of the most important contributions that we have presented in the previous chapters. We are guided through the summaries by providing answers to slightly-rephrased versions of the questions we raised in the introductory chapter (page 3). After that, we will proceed with the conclusions and provide an outlook on possible future work.

9.1. Wrapping up TriAL-QL and RDFPath

In this dissertation we first studied how to express navigational querying-features that were missing so far in widely-used languages like SPARQL and its variations. Secondly, we investigated their implementation on distributed frameworks to facilitate their evaluation against web-scale RDF data. Next, we conclude our experience with both languages and their respective implementations by weighing their strengths and weaknesses.

RDFPath is a purely-navigational query language that features an intuitive syntax, allows the querying of RDF data along with its ontology, and supports aggregations. Its key feature is its path-based semantics that provide, in contrast to most other approaches, all complete paths that have been traversed by an RDFPath expression. TriAL-QL is a SQL-like language with some procedural concepts. Its closed algebra enables the querying of all variants inherent to the triple-based structure of RDF in a natural way. By adding the concept of provenance, we address one of the main drawbacks of the original algebra (TriAL*). In this way, we allow the explanation of the origin of a triple by means of RDFp paths. However, there is a crucial difference between both approaches which we would like emphasize here. The results in RDFPath are meant to contain all paths expressed by an RDFPath query. TriAL-QL, in contrast, uses an existential semantics where a triple can occur at most once in the result. The provenance path does not affect the number of results obtained by TriAL-QL, but instead adds an additional fourth element that describes only *one* possible path which can be used to derive this one triple. More paths might exist, but we do not consider them. Overall, we believe that for use cases in the area of life science and drug discovery, as noted in the introduction of this dissertation, RDFPath is much more valuable. Providing a granular view on *all* connections between, e.g. genes and certain diseases, increases the chances of discovering new insights. On the other hand, due to the large volume of the data, one could also argue that a more compact view of the connections is more important as a starting point for further investigations. Overall, we see TriAL-QL due to its triple-based nature more suited for ETL-like scenarios, where data is preprocessed for a certain task. RDFPath, however, due to its very good performance result and meaningful results is well suited for interactive and more explorative querying tasks. Both query languages have their strong and weak points depending on the respective use case. Their unique features, along with the ability to emit their results again as RDF triples (which encode paths by means of an ontology), complement well-established RDF query languages.

The RDFPath MapReduce Processor is a distributed execution engine for RDFPath built on top of MapReduce. It has shown very good scaling properties and enabled the processing of web-scale RDF data. However, its executions times are in the order of minutes to hours, which was not satisfying for rather selective queries and motivated a successive work. The RDFPath Engine is a hybrid engine that supports the evaluation of RDFPath queries on both Spark and Impala, while using one common data store. We investigated various evaluation, execution, and storage strategies for that engine which in the end provide very good execution times and scaling properties. The same applies for the TriAL-QL Engine. Both engines demonstrate a superior performance in comparison with six competitive RDF management systems and one graph database. With the development of both engines, we have recognized that despite their different semantics, the evaluation, execution, and storage strategies were – with a few minor adoptions and restrictions – mostly compatible. We have further observed that an existential semantics does not necessarily lead to better performance characteristics in a distributed processing framework. The costs involved in ensuring distinct results in distributed environments are significantly higher than on single machines since in most cases they require the data to be repartitioned across the machines. Consequently, simply keeping all paths, as done by the RDFPath Engine, was the more beneficial strategy in most of our experiments. However, we must note that the number of duplicate triples in the intermediate results was rather low. As a result, only a few triples were removed by this costly operation, which in turn means that the gain of it was rather low. Thus, we can expect in cases where an extensive amount of duplicates were removed, that the TriAL-QL Engine is more beneficial. To sum up, both query engines have shown a comparable performance and confirmed the suitability of the presented strategies and used technologies for the evaluation of RDFPath and TriAL-QL queries. The decision of which to use can therefore be done by the query language and its features rather than its implementation. Moreover, the concept of a shared data pool in HDFS that can be accessed by all applications implemented on top of Hadoop simplifies testing both our languages and enables the post-processing of their results in, e.g. another Hadoop-based SPARQL engine.

9.2. Summary

The *World Wide Web* has become the first source of knowledge in all areas of our life. In recent years, we could observe how this traditional *web of documents* is evolving into a *web of data and things*, in which human-readable information is given a well defined *machine-processable* meaning [BHL01]. This so-called *Semantic Web* facilitates unified knowledge bases that allow us to interconnect data from various domains and in multiple languages. The opportunities which are seen in such rich knowledges bases are manifold where particularly cross-domain knowledge is gaining a lot of attention recently. One interesting research field in that context is life science, where biological RDF data is already used to determine, for instance, whether proteins are located in the cell nucleus or any subpart [ABE+09b, CDJ+10, WDS+12, CKK+13]. Interconnecting such information in a further step with, e.g. RDF data about industrial pollution might lead to new insights about its impact on certain diseases. Further examples include the combination of social network data with governmental information, which are also more and more available in a semantically-annotated format.

Making it possible to retrieve relevant information from such knowledge bases is therefore a crucial task for query languages. However, for that we need *expressive* querying features, where *navigational* queries are among the most important query patterns that we need to consider. Moreover, if we consider the aforementioned example of biological data, we not only need expressive queries but also meaningful and human-understandable results that explain in detail how things are connected. Yet, there exist important properties in RDF data which cannot be captured by current RDF query languages including SPARQL 1.1 and its derivations. This is the first important problem that we addressed in this dissertation by studying expressive, navigational query languages for RDF that capture these issues. For that, we had to investigate the following problems.

1. **How to fill the gap of current query languages which hampers the querying of RDF data together with its schema and ontology?**
We introduced the syntax and semantics of two navigational query languages for RDF, namely RDFPath and TriAL-QL, that address the issues from two different perspectives. RDFPath is a fully path-based language that introduces

expressions which allow the traversal of RDF data along with its schema and ontology, and which supports aggregations. TriAL-QL is based on *Triple Algebra with Recursion* which was particularly designed to capture all variants inherent in the triple-based model. We used that algebra as a basis and extended it with further expressions that allow us, e.g. to express a larger variety of recursions. Lastly, to make this algebra usable in practice, we designed an SQL-like syntax for it.

2. **How to provide more meaningful results which contain all resources traversed on a path?**
From the beginning RDFPath was designed with the goal of providing more meaningful results. Its path-based semantics provide, in comparison with widely-used existential semantics, the complete path between nodes, rather than just one pair of nodes that confirms its existence. TriAL-QL is a good example for such an existential semantics. In order to provide more meaningful results, we introduced the concept of provenance for TriAL-QL, which enhances each derived triple with a fourth element that describes its source of origin. For this we reused the RDFp data model, where the composition of the actual provenance path is specified as part of the query.

3. **How to stay compatible with the ecosystem of languages and technologies developed for the Semantic Web so far but nonetheless preserve the knowledge represented in paths?**
Having paths in results solved the second issue but in turn hampers the compatibility with other RDF management systems, as they mostly require triple-based RDF graphs as input. We therefore introduced two mappings which enable the mapping of paths to RDF by means of an ontology or in the case of quadruples a string concatenation. Both approaches are applicable for RDFPath and TriAL-QL and enable the post-processing of their results with another RDF language, such as SPARQL.

While the constant growth of semantically-annotated data occurring in many application domains justifies the above-mentioned querying features, it also raises the need for novel approaches that enable their evaluation on large data sizes. This brings us to the second important problem that we addressed in this work, where we investigated how to evaluate both our languages against web-scale RDF data. There is a lack of RDF management systems that are able to query web-scale RDF in general, and expressive navigational query languages in particular. To the best of our knowledge, our work presented in this dissertation describes the first distributed RDF management system, that allows us to query using such expressive query languages against web-scale RDF data. Within the scope of this work, we had to investigate the following issues:

4. **What is a suitable processing framework on which we should build our system to evaluate complex, navigational expressions against web-scale data?** At the beginning of working on this thesis, MapReduce

was the dominant processing framework, and thus it was also our starting point. We developed at first the RDFPath MapReduce Processor implemented directly in MapReduce. To improve the overall performance and scalability, we have prosed two *Map-Side Merge* joins as an optimization strategy. Both joins are computed completely in the map phase and support a cascaded execution by utilizing the reduce phase just for sorting. MapReduce has proven to be a good choice for data-intensive tasks. We see it therefore as a module in a Semantic Web workflow, where its task is to preprocess complex navigational queries. These are in turn queried with another RDF management system that supports more interactive querying response times. With the continuously-decreasing prices of main memory in recent years, we can observe the emergence of novel *in-memory* data processing systems, which start to pave the way for scalable and interactive querying on large-scale data. Following this trend, we have introduced the TriAL-QL Engine and RDFPath Engine implemented on top of both *Impala*, a massive parallel SQL query engine on Hadoop, and *Spark*, a fast general execution framework for large-scale data processing, while sharing *one* unified data store in HDFS. Their support for SQL, which we used as an intermediate language, facilitated the development of such hybrid systems. Both *in-memory* data processing systems not only exhibit good performance and scalability but allow for both data-intensive tasks and interactive querying. Our experiments have shown that for complex navigational querying, Impala is currently the best choice on Hadoop.

5. **How to provide a good compatibility with other RDF management systems and support the development of workflows composed of multiple Semantic Web tools and languages?**
   Hadoop, with its large surrounding ecosystem, has become the most prominent and de-facto industry standard for Big Data processing, with many infrastructures deployed in research, industry, or in the cloud. The key concept is to apply the principle of HDFS also on the Semantic Web, thus having just one shared *data pool* that can be accessed by various applications without the need for data duplication or movement. There have already been multiple tools developed for the Semantic Web that make use of HDFS. Examples include the SPARQL engines and OWL Reasoner. The interoperability between them, which is facilitated by using one common data pool, is in our opinion one of the most promising approaches for developing an ecosystem of compatible Semantic Web tools.

6. **How can we improve the query performances by means of better RDF storage layouts?**
   For the RDFPath MapReduce Processor, we have developed our own RDFp Store for storing paths, which uses an optimized serializable format for efficient comparison and retrieval of paths. With the change to *in-memory* data processing systems, where we used SQL as an intermediate layer, we

also had to design a proper data layout that can be accessed efficiently by Impala and Spark. For that we chose Parquet, an efficient columnar storage format, for which we adapted multiple storage strategies. We used Vertical Partitioning and the Extended Vertical Partitioning strategy for storing input graphs, and designed a storage schema for efficient intermediate results that are required for recursive expressions.

7. **How to evaluate crucial expression in our languages more efficiently by means of evaluation strategies? And how to optimize the most important querying pattern?**
   We propose multiple evaluation strategies for recursive expressions in TriAL-QL Engine and RDFPath Engine. We make use of the well-studied problem of calculating the transitive closure (TC). We used two approaches as a starting point, namely the *semi-naive* and *smart* TC algorithms, and adopted them to perform well in our processing frameworks. Furthermore, we also investigated different execution strategies, which specify how translated SQL queries are composed for their execution and when intermediate results need to be materialized to disk. Further optimized algorithms were proposed for the connectivity pattern, which asks for the connection between two given resources.
8. **How do our engines perform in comparison to competitive RDF management systems for path-based query pattern?**
   Apart from experiments that compared the individual algorithms and storage strategies implemented for each respective query engine, we have conducted a comprehensive comparison with other systems. We compared the TriAL-QL Engine and RDFPath Engine with six competitive RDF Management systems and one graph database using the Incremental Linear Testing use cases of the WatDiV benchmark. The evaluation clearly demonstrated that Impala, combined with a proper evaluation strategies and efficient data storage layout, outperforms all competitive systems.

9.3. Conclusion

To conclude this dissertation, we want to discuss whether the proposed languages and their implementations are appropriate solutions for the challenges which we discussed for the Semantic Web and discuss their compatibility with the existing stack of languages and tools. RDFPath and TriAL-QL are two expressive, navigational languages which enable the querying of RDF data along with its schema and ontology. Their expressive querying features and the ability to return complete paths are widely needed in practice. We consider them therefore as supplementary languages that cover cases which are yet not expressible in the W3C standard query language SPARQL. We hope that our good results encourage and motivate the further development of a standardized approach as with SPARQL, similar to previous research work in that area which lead to the specification of Property Paths, the navigational component in SPARQL 1.1. In the meantime, we can argue that the support of triple-based results in RDFPath TriAL-QL easily allow for an integration with other RDF management systems, where both of our languages may serve as a preprocessing step.

The RDFPath MapReduce Processor, RDFPath Engine and TriAL-QL Engine have demonstrated that the Hadoop ecosystem provides suitable solutions for processing navigational queries against web-scale RDF data. The RDFPath MapReduce Processor, with its good scaling properties and robustness that are derived from the batch-oriented workflow of MapReduce and its disk-based operators, is a good choice for offline, ETL-like use cases that need to process large volumes of RDF data. However, in cases where we need interactive response times, and thus in the order of seconds to a few minutes, the novel in-memory data processing frameworks are clearly the better choice. With the RDFPath Engine and the TriAL-QL Engine we developed two comprehensive RDF management systems that facilitate the querying of complex navigational expressions in their respective languages against web-scale RDF data. We demonstrated that important querying tasks such as recursions or the connectivity between nodes can be efficiently computed. Both engines have their strengths and weaknesses but have comparable performance due to similar evaluation and storage strategies. Selective queries exhibit query response times in the order of seconds on datasets with more than one billion triples. More data-intensive use-cases that produces, e.g. over 25 billion results finished in the order of tens of minutes. Yet one of the main benefits we see in the usage of the Hadoop ecosystem is the concept of a common data pool that is shared across various RDF management engines, developed on top of Hadoop. This key concept further constitutes a compatible ecosystem for processing RDF data where we see our engines as complementary tools that can then be embedded in a workflow for, e.g. preprocessing more complex paths.

9.4. Outlook

This dissertation studied two navigational query languages for RDF and their implementation on distributed frameworks. While many issues are already resolved, there are still several questions which remain open, with many possible directions for future work. We would like to briefly describe a few of them:

Theoretical Study. We studied RDFPath and TriAL-QL from a rather practical perspective. Even though our experiments demonstrated very good properties of our engines, what needs to be done next is a more theoretical analysis of the evaluation complexity of both languages.

Additional Querying Features. The querying features which we proposed with RDFPath and TriAL-QL in this work already allowed us to extract many important properties in RDF, which are not expressible in SPARQL 1.1 and its derivations. However, there are of course many other features that might be relevant for querying Semantic Web data, such as more sophisticated aggregations. Indeed, RDFPath already supports aggregation functions which are applied on the last element of each path. This approach can be further generalized, and in addition adapted for TriAL-QL. The next steps for TriAL-QL are more abstract syntax expressions which reduce, for instance, the number of statements that need to be written in order to describe recursive patterns.

Novel Hadoop Components. One main advantage of the Hadoop ecosystem is its continuous development which is reflected by novel frameworks and layers that are consistently added. One new addition from which we expect many benefits for our query engines is Apache Kudu[1], a new storage layer for Hadoop. It is a distributed table-based storage, designed to scale easily with tens of CPU cores and takes advantage of solid state drives.

Optimized Evaluation and Storage Strategies. We have already explored various evaluation, execution and storage strategies and investigated their properties in our query engines. Thereby we have seen that, even small algorithmic adjustments, such as avoiding the – in distributed environment rather costly – SQL DISTINCT expression can have a huge impact on the query performance. Consequently, there is still lot of work that needs to be done in order to provide a better understand of the cost-benefit ratio of various algorithms and data structures in the area of distributed processing frameworks.

[1]https://kudu.apache.org/

Part VI.

Appendix

A. Dictionary for Language Features

This appendix explains the features used for comparing RDF query languages in Section 3.3, Table 3.1 on page 53. The features are inspired by numerous previous studies such as [Abi97, PAL$^+$02, HBEV04, AGH04, AG05, AG08, KBM08].

Namespaces. Namespaces are an essential part of any web-based query language, since they allow us to distinguish between multiple sources or versions of RDF data. This way, one can query for instance all values using a certain namespace or a namespace with a certain pattern. Moreover, they allow for a shorter representation of resources, by abbreviating long prefixes which in turn enable smaller queries [HBEV04, KBM08].

Constraints. Constraints reflect the ability to express restrictions on the RDF data that are used, for instance, in filter expressions [Abi97].

Language. A query language identifies the language used in the RDF data from literals by interpreting the XML attribute `xml:lang` automatically. More modern languages like SPARQL allow us to simulate this feature by simply utilizing the triple-based encoding of a language, where querying a specific language becomes part of the query pattern itself [HBEV04].

Lexical & value space. With the support of XML Schema, RDF supports typed literals such as numbers and dates by means of their lexical representation. In the case of the *lexical space*, there is just a text-based interpretation, where for instance `08` and `8` are considered to be not equal. Only the *value space* enables a mapping into their numeric representation allowing for more expressive comparisons of numbers.

Graph pattern. Such patterns were originally introduced as *path expressions in patterns* in 1997 as part of the first features which were considered to be crucial for querying semi-structured data [Abi97]. They describe the simplest use of path expression by concatenating attribute names, which typically correspond to traversing a graph. The authors in [Abi97] already differentiate between two interpretations of the traversed paths: (1) the set of objects at the end of a path and (2) the paths themselves. In later comparisons between languages, such pattern were denoted by *Path* [AGH04, HBEV04] and *Graph pattern* to avoid ambiguities.

Adjacent resources. One of the most basic questions of a navigational query language is to determine all resources which are adjacent to a given resource, i.e all its neighbors. If not only *outgoing* but also *incoming* predicates need to be considered, this becomes problematic since the underlying language needs to support some sort of a union operator to combine both directions [AGH04].

Adjacent predicates. Analogous to the previous definition one can also ask for *all* predicates related to a certain resource. Again, one might be interested in resources at either the subject or object [AGH04] position, thus outgoing or incoming predicates.

Fixed-length path. Determining all paths of a predefined, fixed length between two given resources is the first more comprehensive navigational query feature. This feature can be expressed by a union of all possible path patterns (with all directions) by means of a *graph pattern* [AGH04].

Path variable. For more complex querying so-called *path variables* becomes useful for a convenient handling of paths. The idea is that, in contrast to basic variables which refer to atomic resources, a path variable denotes a path as a whole, composed of probably multiple resources [BMY00].

Inverse Path. In order to traverse not only outgoing, but also incoming predicates (edges), we need an operation which enables us to express backward subject-object relations. This is generally achieved by so-called *inverse* traversals.

Non-simple path. A non-simple path allows the multiple-occurrence of the same edge or node. With respect to RDF, arbitrary resources are allowed to be traversed multiple times, regardless of whether they appear in the subject, predicate or object position [BMY00]. However, this flexibility comes along with the problem of cycles which might appear on the path, which demand some sort of mechanism to handle them.

Recursion (Regular Expression). Recursions are the basic building block of capturing paths of arbitrary length on relationships with a transitive nature, like for instance *friends of a friend* relationships. The Kleene star allows the following of a certain pattern, which is neither nested nor contains any filter, an arbitrary number of times, until the transitive closure is computed with regard to the query. Further, we can differentiate between (1) *data recursions*, where we consider explicit triples and follow the only edges (predicates) as know from traditional graph databases and, (2) *schema recursions*, where e.g. the transitivity of `subClassOf` relations need to be handled, where we need to express recursions over the predicate as well. With this query feature we refer solely to data recursions. Schema recursions will be covered by the query feature *entailment (reasoning)* introduced later [HBEV04].

Constrained regular expression. *Constrained regular expressions* add the ability to apply filters on regular expressions, such that the recursively traversed pattern needs to satisfy further constraints on its way. This includes not only constraints on the traversed resources themselves but might also consider, e.g. the neighborhood of a resource [AE14].

Optional pattern. As semi-structured data is meant to represent incomplete and irregular data, and thus exhibits a high degree of diversity in its structure, it is crucial to be able to deal with such incomplete information. For that, an optional pattern enables us to add certain information if present and leaves them out otherwise [HBEV04].

Entailment (Reasoning). With RDF Schema [BG14] and more comprehensive languages, e.g. OWL [Bec09], manifold information can be added to RDF data which are not explicitly stated but which need to be inferred using entailment rules. We refer to them also as topological information about RDF data, or an ontology. There are different approaches for how to infer desired information from the data. First of all, one can implement a so-called *reasoner* which (1) precomputes the transitive closure of the full dataset or (2) utilizes the query to compute just the desired fragment of the transitive closure during query evaluation or (3) by query rewriting. However, in cases where it is not desired to use all available resources, like for instance in large knowledge graphs, it is more elegant to equip the language itself with *reasoning capabilities* rather than relying on an automatic reasoner. Therefore, expressions have to be added to the query language which capture also entailment rules [HBEV04, KBM08].

Querying Topology. As we have noted for *Entailment (Reasoning)*, there exist multiple approaches for how to handle data together with its ontology, which are diverse in their expressiveness. In particular, approaches which are based on the traditional graph model and NREs suffer from the fact that there is a crucial differences between RDF and the classical graph models. For instance, in RDF, it is possible to model the same resource in predicate (edge) and subject/object (node) position [LRV13, AGP14], which is generally not possible in the graph model. We therefore introduce the term *Querying Topology* to refer to those query languages, which (1) take this difference into account and capture *all* querying facilities inherent to the *triple-based* model of RDF, and (2) provide a smooth integration of querying schema and data in a combined fashion.

Output Paths. As noted in [Abi97], one can differentiate between two interpretation of traversed paths: (1) the set of objects at the end (and beginning) of a path and (2) the paths themselves. It has a significant impact on evaluation complexity which of the two is chosen, since computing complete paths is known to be a rather costly task. This is why most languages follow an existential approach that emits pairs of nodes rather than complete paths. This query feature refers therefore to those languages which are able to provide not only the *first* and the *last* resource of traversed, non-simple paths, but include *all intermediate* resources [PSHL12].

Degree of resource. The degree of a resources is a scalar value determined by the number of adjacent edges introduced earlier, which in general requires support for aggregation functions [AGH04, HBEV04, AG05].

Distance between resources. The distance between two resources is defined by the number of intermediate resources traversed along the path. Here, one can differentiate between counting only those resources which are either in subject or object position or alternatively, include also resources in predicate position. The result is again a scalar value [AGH04, HBEV04, AG05, PSHL12].

Shortest Path. Computing shortest paths is a well-studied problem in graph theory and also highly relevant for querying RDF data [GH05]. This feature refers to the ability of computing at least one shortest path between two predefined resources. Hereby it is required that the answer matches our prior definition of a path result, i.e. that it includes *all* traversed resources [PSHL12]. One can differ between two kind of supported query pattern: the aforementioned shortest paths correspond to the so-called *single-source-single-destination shortest paths* (SSSP) problem. A more difficult task is the so-called *All-Pair shortest path* (APSP) problem, where for all pairs of resources their respective shortest-path is computed. In conjunction with constrained regular expressions, the permitted shortest paths can be further restricted by the use of constraints.

Union. This querying feature corresponds to the basic algebraic operation in relational algebra, where two sets are merged into one [HBEV04].

Difference. This querying feature corresponds to the basic algebraic operation in relational algebra, where one set is subtracted from the other one [HBEV04].

Aggregation. With aggregations, a set of resources or literals can be reduced to a scalar value. For this a function is applied to the results which, for instance, counts the number of elements[HBEV04, KBM08, PSHL12].

Collections & Containers. A container (open group) and collection (closed group) in RDF is used to represent a group of resources, where for instance `rdf:Seq` specifies a collection with a human-readable numerical order [MMM14]. A language is considered to support this feature if it allows the retrieval of either individual or all elements of these containers and collections [HBEV04, KBM08].

Closure Property. The closure property is satisfied if the result of an operation produces again elements of the same data model. Thus, if the input for a query is RDF data, the result must also be RDF data. [Cod70, HBEV04, LRV13]

Sorting. Sorting specifies the capability of a language to sort the result in accordance with a lexical or value-space order [HBEV04].

B. RDFPath Grammar

In this appendix, we show the complete grammar of RDFPath in g4 notation as used for the parser.

query

```
query : ctx ':' path+ ('.')? ( result )?;
ctx   : iri | '*' | setIRI;
```

path

```
path      : travop rp  ('('recexpr')')?;
travop    : '/' | '\\' ;
rp        : normalp | branchp;
normalp   : edge ('[' filter ']')?;
edge      : iri | '(*)' | setIRI;
branchp   : '(' path+ (branchop path+)* ')';
branchop  : '||';
```

recexpr

```
recexpr   : repeat | recursion;
recursion : intvaluetxt ',' intvaluetxt;
repeat    : intvalue;
```

filter

```
filter       : normalf | compf | nestedf | cnestedf;
nestedf      : path+ nestedfexpr;
nestedfexpr : normalf || compf;
compf        : normalf (logop normalf)+ ;
cnestedf     : path+ cnestedop path+;
cnestedop    : op;
normalf      : valuef | funcvaluef | funcf ;
```

```
valuef      : op value;
funcvaluef  : op funcvalue;
funcf       : func;

op          : '=' | '>' | '<' | '!=' | '>=' | '<=' ;
value       : textvalue;
funcvalue   : textvalue '(' textvalue ')';
func        : textvalue '()';
logop       : '&&' | '||';
```

result

```
result : 'project(' setINT ')' | 'nodes()' | 'count()' | 'avg()'
         | 'sum()' | 'max()' | 'min()' | 'triple()'
         | 'limit(' intvalue ')';
```

iri

```
iri         : textvalue;
setIRI      : '{' iri ( ',' iri)* '}';
setINT      : intvalue (',' intvalue)*;
intvaluetxt : intvalue | '*' | '+';
intvalue    : INT;
```

Tokens

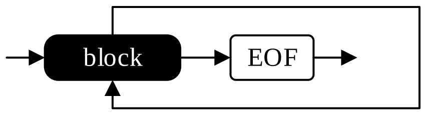
```
INT        : [0-9]+ ;
textvalue  : TEXTVALUE | intvalue;
TEXTVALUE  : LEGALCHAR+;
LEGALCHAR  :  ~( '/' | '\\' | '*' | '(' | ')' | '[' | ']' | '.' | '=' |
               '|' | '&' | '>' | '<' | '!' | '{' | '}' | ',');
Whitespace : ('\t' | '\r' | '\n' ) -> skip;
Comment    : ('-#') ~( '\r' | '\n' )* -> skip;
```

C. TriAL-QL Grammar

In this appendix, we show the complete grammar of TriAL-QL in both a textual and a visual notation. Please note that rule names which start with a capital are *tokenizer* rules and those with a lower-case character at the beginning are *parsing* rules. Rules are sorted hierarchically, and thus at first parser rules are described, where higher-level rules are listed before the lower-level ones.

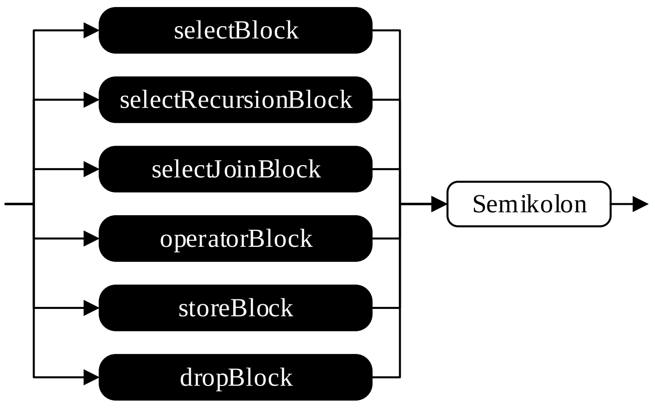
parse

```
parse : block+ EOF ;
```



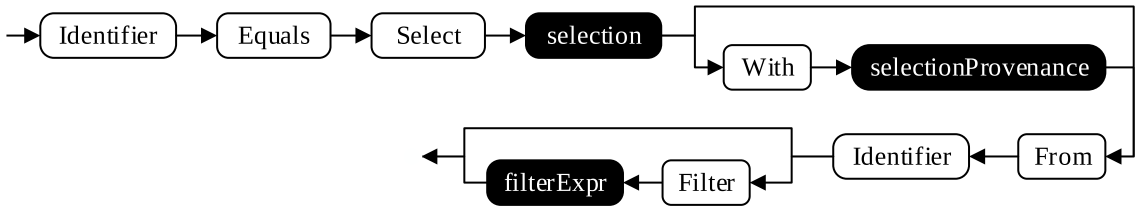
block

```
block : (selectBlock | selectRecursionBlock | selectJoinBlock
| operatorBlock | storeBlock | dropBlock) Semikolon;
```



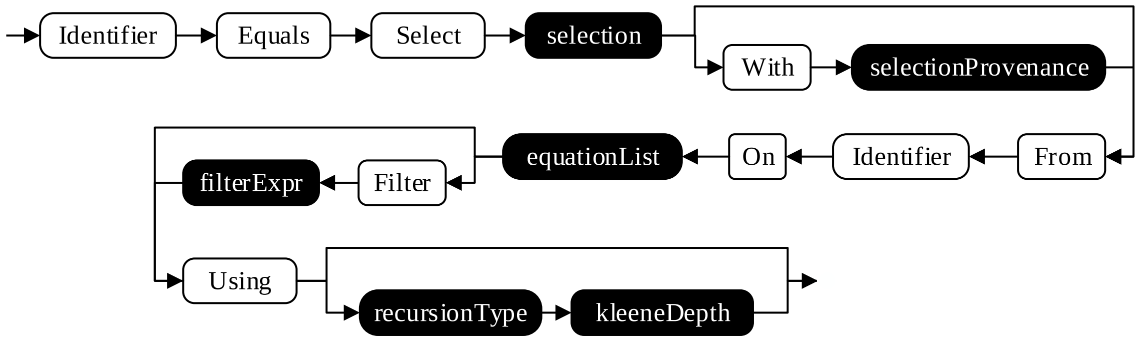
selectBlock

```
selectBlock : Identifier Equals Select selection (With selectionProvenance)?
From Identifier (Filter filterExpr)?;
```



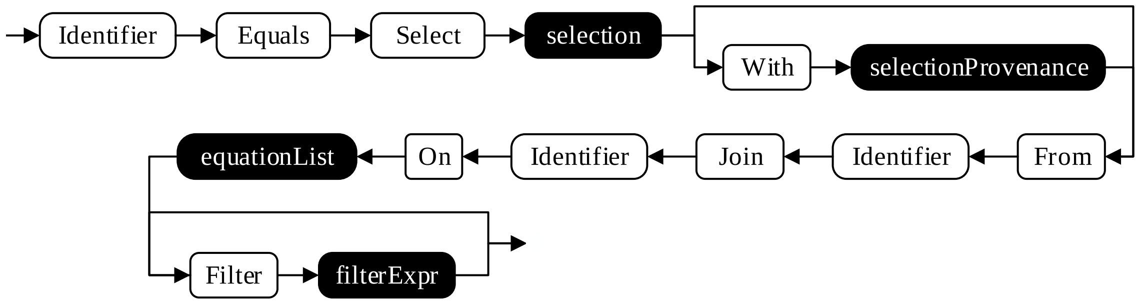
selectRecursionBlock

```
selectRecursionBlock : Identifier Equals Select selection
(With selectionProvenance)?
From Identifier On equationList (Filter filterExpr)?
Using recursionType (kleeneDepth)?;
```



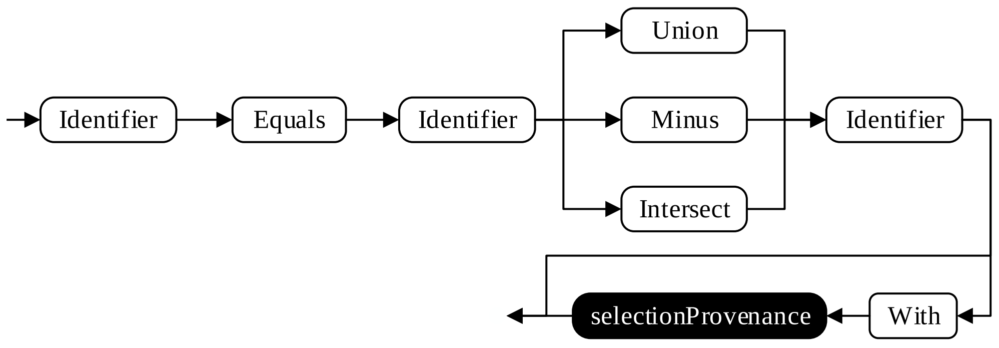
selectJoinBlock

```
selectJoinBlock : Identifier Equals Select selection
(With selectionProvenance)?
From Identifier Join Identifier (On equationList)? (Filter filterExpr)?;
```



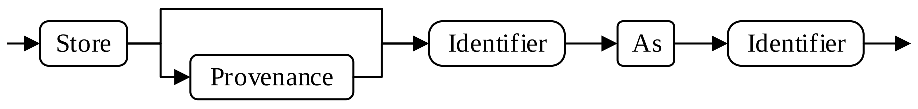
operatorBlock

```
operatorBlock : Identifier Equals Identifier (Union | Minus | Intersect)
Identifier (With selectionProvenance)?;
```



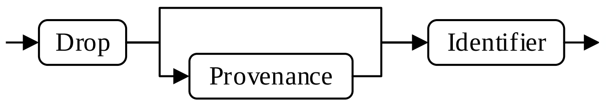
storeBlock

```
storeBlock : Store Provenance? Identifier As Identifier;
```



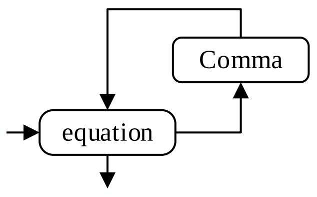
dropBlock

```
dropBlock : Drop Provenance? Identifier;
```



equationList

```
equationList : equation (Comma equation)*;
```



equation

```
equation : term equationOperator term;
```

equationOperator

```
equationOperator : Equals | NEquals | GTEquals
| LTEquals | GT | LT;
```

term

```
term : Pos | literal;
```

literal

```
literal : Bool
| Int
| String;
```

selection

```
selection : Pos Comma Pos Comma Pos;
```

selectionProvenance

```
selectionProvenance : (Pos | 'r1' | 'r2') (Comma (Pos | 'r1' | 'r2'))*;
```

recursionType

```
recursionType : Left | Right;
```

kleeneDepth

```
kleeneDepth : OBracket (PositiveInt | Star| Plus)
              (Comma (PositiveInt | Star))? CBracket;
```

filterExpr

```
filterExpr : filterExprAnd (Or filterExprAnd)*;
```

filterExprAnd

```
filterExprAnd : equation (And equation)*;
```

Tokens

```
Select       : 'SELECT';
From         : 'FROM';
Join         : 'JOIN';
On           : 'ON';
Store        : 'STORE';
Provenance   : 'PROVENANCE';
As           : 'AS';
Using        : 'USING';
Left         : 'left';
Right        : 'right';
And          : 'AND';
Or           : 'OR';
Drop         : 'DROP';
Union        : 'UNION';
Minus        : 'MINUS';
Intersect    : 'INTERSECT';
Filter       : 'FILTER';
With         : 'WITH';
Load         : 'LOAD';
Cache        : 'CACHE';
Star : '*';
Plus : '+';

Pos          : ('s' | 'p' | 'o') [1-2];

Equals       : '=';
NEquals      : '!=';
GTEquals     : '>=';
LTEquals     : '<=';
GT           : '>';
LT           : '<';
Apostrophe   : '\'';
Quote        : '"';
Semikolon    : ';';
OBracket     : '[';
CBracket     : ']';
Comma        : ',';
```

```
Bool         : 'true' | 'false';

Identifier   : ('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | '_' | Digit)*;

String       : ('"' (~('"' | '\\') | '\\' .)* '"'
              | '\'' (~('\'' | '\\') | '\\' .)* '\'');

PositiveInt : [1-9] Digit*;

Int          : PositiveInt | '0';

fragment Digit : [0-9];

Comment      : ('--' | '#') ~( '\r' | '\n' )* -> skip;

Whitespace   : [ \t\r\n] -> skip;
```

D. WatDiv Incremental Linear Testing Queries

D.1. SPARQL Queries

Terms enclosed within % are placeholders that get instantiated dynamically by the WatDiv query generator based on the `#mapping` command.

D.1.1. Incremental User-Bound Queries (Type 1)

```
#mapping v0 wsdbm:User uniform
```

IL-1-5:
```
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 WHERE {
%v0%  wsdbm:follows   ?v1 .
?v1   wsdbm:likes     ?v2 .
?v2   rev:hasReview   ?v3 .
?v3   rev:reviewer    ?v4 .
?v4   wsdbm:friendOf  ?v5 . }
```

IL-1-6:
```
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 WHERE {
%v0%  wsdbm:follows        ?v1 .
?v1   wsdbm:likes          ?v2 .
?v2   rev:hasReview        ?v3 .
?v3   rev:reviewer         ?v4 .
?v4   wsdbm:friendOf       ?v5 .
?v5   wsdbm:makesPurchase  ?v6 . }
```

IL-1-7:
```
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 WHERE {
%v0%  wsdbm:follows        ?v1 .
?v1   wsdbm:likes          ?v2 .
?v2   rev:hasReview        ?v3 .
?v3   rev:reviewer         ?v4 .
?v4   wsdbm:friendOf       ?v5 .
?v5   wsdbm:makesPurchase  ?v6 .
?v6   wsdbm:purchaseFor    ?v7 . }
```

IL-1-8:
```
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
%v0%  wsdbm:follows        ?v1 .
?v1   wsdbm:likes          ?v2 .
?v2   rev:hasReview        ?v3 .
?v3   rev:reviewer         ?v4 .
?v4   wsdbm:friendOf       ?v5 .
?v5   wsdbm:makesPurchase  ?v6 .
?v6   wsdbm:purchaseFor    ?v7 .
?v7   sorg:author          ?v8 . }
```

D.1.2. Incremental Retailer-Bound Queries (Type 2)

```
#mapping v0 wsdbm:Retailer uniform
```

IL-2-5:
```
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 WHERE {
%v0%  gr:offers       ?v1 .
?v1   gr:includes     ?v2 .
?v2   sorg:director   ?v3 .
?v3   wsdbm:friendOf  ?v4 .
?v4   wsdbm:friendOf  ?v5 . }
```

IL-2-6:
```
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 WHERE {
%v0%  gr:offers       ?v1 .
?v1   gr:includes     ?v2 .
?v2   sorg:director   ?v3 .
?v3   wsdbm:friendOf  ?v4 .
?v4   wsdbm:friendOf  ?v5 .
?v5   wsdbm:likes     ?v6 . }
```

IL-2-7:
```
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 WHERE {
%v0%  gr:offers       ?v1 .
?v1   gr:includes     ?v2 .
?v2   sorg:director   ?v3 .
?v3   wsdbm:friendOf  ?v4 .
?v4   wsdbm:friendOf  ?v5 .
?v5   wsdbm:likes     ?v6 .
?v6   sorg:editor     ?v7 . }
```

IL-2-8:
```
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
%v0%  gr:offers            ?v1 .
?v1   gr:includes          ?v2 .
?v2   sorg:director        ?v3 .
?v3   wsdbm:friendOf       ?v4 .
?v4   wsdbm:friendOf       ?v5 .
?v5   wsdbm:likes          ?v6 .
?v6   sorg:editor          ?v7 .
?v7   wsdbm:makesPurchase  ?v8 . }
```

D.1.3. Incremental Unbound Queries (Type 3)

IL-3-5:
```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 WHERE {
?v0  gr:offers       ?v1 .
?v1  gr:includes     ?v2 .
?v2  rev:hasReview   ?v3 .
?v3  rev:reviewer    ?v4 .
?v4  wsdbm:friendOf  ?v5 . }
```

IL-3-6:
```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 WHERE {
?v0  gr:offers       ?v1 .
?v1  gr:includes     ?v2 .
?v2  rev:hasReview   ?v3 .
?v3  rev:reviewer    ?v4 .
?v4  wsdbm:friendOf  ?v5 .
?v5  wsdbm:likes     ?v6 . }
```

IL-3-7:
```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 WHERE {
?v0  gr:offers       ?v1 .
?v1  gr:includes     ?v2 .
?v2  rev:hasReview   ?v3 .
?v3  rev:reviewer    ?v4 .
?v4  wsdbm:friendOf  ?v5 .
?v5  wsdbm:likes     ?v6 .
?v6  sorg:author     ?v7 . }
```

IL-3-8:
```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
?v0  gr:offers       ?v1 .
?v1  gr:includes     ?v2 .
?v2  rev:hasReview   ?v3 .
?v3  rev:reviewer    ?v4 .
?v4  wsdbm:friendOf  ?v5 .
?v5  wsdbm:likes     ?v6 .
?v6  sorg:author     ?v7 .
?v7  wsdbm:follows   ?v8 . }
```

D.2. RDFPath Queries

D.2.1. Incremental User-Bound Queries (Type 1)

IL-1-5:
```
%User% :/wsdbm:follows
        /wsdbm:likes
        /rev:hasReview
        /rev:reviewer
        /wsdbm:friendOf
        .project(1,3,5,7,9,11)
```

IL-1-6:
```
%User% :/wsdbm:follows
        /wsdbm:likes
        /rev:hasReview
        /rev:reviewer
        /wsdbm:friendOf
        /wsdbm:makesPurchase
        .project(1,3,5,7,9,11,13)
```

IL-1-7:
```
%User% :/wsdbm:follows
        /wsdbm:likes
        /rev:hasReview
        /rev:reviewer
        /wsdbm:friendOf
        /wsdbm:makesPurchase
        /wsdbm:purchaseFor
        .project(1,3,5,7,9,11,13,15)
```

IL-1-8:
```
%User% :/wsdbm:follows
        /wsdbm:likes
        /rev:hasReview
        /rev:reviewer
        /wsdbm:friendOf
        /wsdbm:makesPurchase
        /wsdbm:purchaseFor
        /sorg:author
        .project(1,3,5,7,9,11,13,15,17)
```

D.2.2. Incremental Retailer-Bound Queries (Type 2)

IL-2-5:
```
%Retailer% :/gr:offers
            /gr:includes
            /sorg:director
            /wsdbm:friendOf
            /wsdbm:friendOf
            .project(1,3,5,7,9,11)
```

IL-2-6:
```
%Retailer% :/gr:offers
            /gr:includes
            /sorg:director
            /wsdbm:friendOf
            /wsdbm:friendOf
            /wsdbm:likes
            .project(1,3,5,7,9,11,13)
```

IL-2-7:
```
%Retailer% :/gr:offers
            /gr:includes
            /sorg:director
            /wsdbm:friendOf
            /wsdbm:friendOf
            /wsdbm:likes
            /sorg:editor
            .project(1,3,5,7,9,11,13,15)
```

IL-2-8:
```
%Retailer% :/gr:offers
            /gr:includes
            /sorg:director
            /wsdbm:friendOf
            /wsdbm:friendOf
            /wsdbm:likes
            /sorg:editor
            /wsdbm:makesPurchase
            .project(1,3,5,7,9,11,13,15,17)
```

D.2.3. Incremental Unbound Queries (Type 3)

IL-3-5:
```
* :/gr:offers
   /gr:includes
   /rev:hasReview
```

```
          /rev:reviewer
          /wsdbm:friendOf
          .project(1,3,5,7,9)
```

IL-3-6:
```
        * :/gr:offers
          /gr:includes
          /rev:hasReview
          /rev:reviewer
          /wsdbm:friendOf
          /wsdbm:likes
          .project(1,3,5,7,9,11)
```

IL-3-7:
```
        * :/gr:offers
          /gr:includes
          /rev:hasReview
          /rev:reviewer
          /wsdbm:friendOf
          /wsdbm:likes
          /sorg:author
          .project(1,3,5,7,9,11,13)
```

IL-3-8:
```
        * :/gr:offers
          /gr:includes
          /rev:hasReview
          /rev:reviewer
          /wsdbm:friendOf
          /wsdbm:likes
          /sorg:author
          /wsdbm:follows
          .project(1,3,5,7,9,11,13,15)
```

D.3. TriAL-QL Queries

D.3.1. Incremental User-Bound Queries (Type 1)

IL-1-5:
```
t1 = SELECT s1, p1, o2 FROM watdiv JOIN watdiv ON o1=s2
     FILTER s1 = %User% AND p1 = 'wsdbm:follows'  AND p2 = 'wsdbm:likes' ;
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:hasReview';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:reviewer';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
```

IL-1-6:
```
t1 = SELECT s1, p1, o2 FROM watdiv JOIN watdiv ON o1=s2
     FILTER s1 = %User% AND p1 = 'wsdbm:follows'  AND p2 = 'wsdbm:likes' ;
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:hasReview';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:reviewer';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
```

```
          t5 = SELECT s1, p1, o2 FROM t4 JOIN watdiv ON o1=s2
               FILTER p2 = 'wsdbm:makesPurchase';
```

IL-1-7:
```
t1 = SELECT s1, p1, o2 FROM watdiv JOIN watdiv ON o1=s2
     FILTER s1 = %User% AND p1 = 'wsdbm:follows'  AND p2 = 'wsdbm:likes' ;
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:hasReview';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:reviewer';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t5 = SELECT s1, p1, o2 FROM t4 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:makesPurchase';
t6 = SELECT s1, p1, o2 FROM t5 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:purchaseFor';
```

IL-1-8:
```
t1 = SELECT s1, p1, o2 FROM watdiv JOIN watdiv ON o1=s2
     FILTER s1 = %User% AND p1 = 'wsdbm:follows'  AND p2 = 'wsdbm:likes' ;
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:hasReview';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:reviewer';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t5 = SELECT s1, p1, o2 FROM t4 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:makesPurchase';
t6 = SELECT s1, p1, o2 FROM t5 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:purchaseFor';
t7 = SELECT s1, p1, o2 FROM t6 JOIN watdiv ON o1=s2
     FILTER p2 = 'sorg:author';
```

D.3.2. Incremental Retailer-Bound Queries (Type 2)

IL-2-5:
```
t1 = SELECT s1, p1, o2 FROM watdiv100 JOIN watdiv ON o1=s2
     FILTER s1 = %Retailer% AND p1 = 'gr:offers'  AND p2 = 'gr:includes' ;
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'sorg:director';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv1 ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
```

IL-2-6:
```
t1 = SELECT s1, p1, o2 FROM watdiv100 JOIN watdiv ON o1=s2
     FILTER s1 = %Retailer% AND p1 = 'gr:offers'  AND p2 = 'gr:includes' ;
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'sorg:director';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv1 ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t5 = SELECT s1, p1, o2 FROM t4 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:likes';
```

IL-2-7:
```
t1 = SELECT s1, p1, o2 FROM watdiv100 JOIN watdiv ON o1=s2
     FILTER s1 = %Retailer% AND p1 = 'gr:offers'  AND p2 = 'gr:includes' ;
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'sorg:director';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv1 ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t5 = SELECT s1, p1, o2 FROM t4 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:likes';
t6 = SELECT s1, p1, o2 FROM t5 JOIN watdiv ON o1=s2
     FILTER p2 = 'sorg:editor';
```

IL-2-8:
```
t1 = SELECT s1, p1, o2 FROM watdiv100 JOIN watdiv ON o1=s2
     FILTER s1 = %Retailer% AND p1 = 'gr:offers'  AND p2 = 'gr:includes' ;
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'sorg:director';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv1 ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t5 = SELECT s1, p1, o2 FROM t4 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:likes';
t6 = SELECT s1, p1, o2 FROM t5 JOIN watdiv ON o1=s2
     FILTER p2 = 'sorg:editor';
t7 = SELECT s1, p1, o2 FROM t6 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:makesPurchase';
```

D.3.3. Incremental Unbound Queries (Type 3)

IL-3-5:
```
t0 = SELECT s1, p1, o1 FROM watdiv
     FILTER p1 = 'gr:offers';
t1 = SELECT s1, p1, o2 FROM t0 JOIN watdiv ON o1=s2
     FILTER p2 = 'gr:includes';
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:hasReview';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:reviewer';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
```

IL-3-6:
```
t0 = SELECT s1, p1, o1 FROM watdiv
     FILTER p1 = 'gr:offers';
t1 = SELECT s1, p1, o2 FROM t0 JOIN watdiv ON o1=s2
     FILTER p2 = 'gr:includes';
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:hasReview';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:reviewer';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t5 = SELECT s1, p1, o2 FROM t4 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:likes';
```

IL-3-7:
```
t0 = SELECT s1, p1, o1 FROM watdiv
         FILTER p1 = 'gr:offers';
t1 = SELECT s1, p1, o2 FROM t0 JOIN watdiv ON o1=s2
     FILTER p2 = 'gr:includes';
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:hasReview';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:reviewer';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t5 = SELECT s1, p1, o2 FROM t4 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:likes';
t6 = SELECT s1, p1, o2 FROM t5 JOIN watdiv ON o1=s2
     FILTER p2 = 'sorg:author';
```

IL-3-8:
```
t0 = SELECT s1, p1, o1 FROM watdiv
     FILTER p1 = 'gr:offers';
t1 = SELECT s1, p1, o2 FROM t0 JOIN watdiv ON o1=s2
     FILTER p2 = 'gr:includes';
t2 = SELECT s1, p1, o2 FROM t1 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:hasReview';
t3 = SELECT s1, p1, o2 FROM t2 JOIN watdiv ON o1=s2
     FILTER p2 = 'rev:reviewer';
t4 = SELECT s1, p1, o2 FROM t3 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:friendOf';
t5 = SELECT s1, p1, o2 FROM t4 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:likes';
t6 = SELECT s1, p1, o2 FROM t5 JOIN watdiv ON o1=s2
     FILTER p2 = 'sorg:author';
t7 = SELECT s1, p1, o2 FROM t6 JOIN watdiv ON o1=s2
     FILTER p2 = 'wsdbm:follows';
```

D.4. Neo4j Cypher Queries

D.4.1. Incremental User-Bound Queries (Type 1)

IL-1-5:
```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)
WHERE s1.node = %User%
  AND p1.node = 'wsdbm-follows'
  AND p2.node = 'wsdbm-likes'
  AND p3.node = 'rev-hasReview'
  AND p4.node = 'rev-reviewer'
  AND p5.node = 'wsdbm-friendOf'
RETURN count(r);
```

IL-1-6:
```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)-[p6]->(o6)
WHERE s1.node = %User%
  AND p1.node = 'wsdbm-follows'
  AND p2.node = 'wsdbm-likes'
  AND p3.node = 'rev-hasReview'
  AND p4.node = 'rev-reviewer'
  AND p5.node = 'wsdbm-friendOf'
  AND p6.node = 'wsdbm-makesPurchase'
RETURN count(r);
```

IL-1-7:

```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)-[p6]->(o6)-[p7]->(o7)
WHERE s1.node = %User%
  AND p1.node = 'wsdbm-follows'
  AND p2.node = 'wsdbm-likes'
  AND p3.node = 'rev-hasReview'
  AND p4.node = 'rev-reviewer'
  AND p5.node = 'wsdbm-friendOf'
  AND p6.node = 'wsdbm-makesPurchase'
  AND p7.node = 'wsdbm-purchaseFor'
RETURN count(r);
```

IL-1-8:

```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)-[p6]->(o6)-[p7]->(o7)-[p8]->(o8)
WHERE s1.node = %User%
  AND p1.node = 'wsdbm-follows'
  AND p2.node='wsdbm-likes'
  AND p3.node='rev-hasReview'
  AND p4.node='rev-reviewer'
  AND p5.node='wsdbm-friendOf'
  AND p6.node='wsdbm-makesPurchase'
  AND p7.node='wsdbm-purchaseFor'
  AND p8.node='sorg-author'
RETURN count(r);
```

D.4.2. Incremental Retailer-Bound Queries (Type 2)

IL-2-5:

```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)
WHERE s1.node = %Retailer%
  AND p1.node = 'gr-offers'
  AND p2.node = 'gr-includes'
  AND p3.node = 'sorg-director'
  AND p4.node = 'wsdbm-friendOf'
  AND p5.node = 'wsdbm-friendOf'
RETURN count(r);
```

IL-2-6:

```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)-[p6]->(o6)
WHERE s1.node = %Retailer%
  AND p1.node = 'gr-offers'
  AND p2.node = 'gr-includes'
  AND p3.node = 'sorg-director'
  AND p4.node = 'wsdbm-friendOf'
  AND p5.node = 'wsdbm-friendOf'
  AND p6.node = 'wsdbm-likes'
RETURN count(r);
```

IL-2-7:
```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)-[p6]->(o6)-[p7]->(o7)
WHERE s1.node = %Retailer% '
  AND p1.node = 'gr-offers'
  AND p2.node = 'gr-includes'
  AND p3.node = 'sorg-director'
  AND p4.node = 'wsdbm-friendOf'
  AND p5.node = 'wsdbm-friendOf'
  AND p6.node = 'wsdbm-likes'
  AND p7.node = 'sorg-editor'
RETURN count(r);
```

IL-2-8:
```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)-[p6]->(o6)-[p7]->(o7)-[p8]->(o8)
WHERE s1.node = %Retailer%
  AND p1.node = 'gr-offers'
  AND p2.node = 'gr-includes'
  AND p3.node = 'sorg-director'
  AND p4.node = 'wsdbm-friendOf'
  AND p5.node = 'wsdbm-friendOf'
  AND p6.node = 'wsdbm-likes'
  AND p7.node = 'sorg-editor'
  AND p8.node = 'wsdbm-makesPurchase'
RETURN count(r)
```

D.4.3. Incremental Unbound Queries (Type 3)

IL-3-5:
```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)
WHERE p1.node = 'gr-offers'
  AND p2.node = 'gr-includes'
  AND p3.node = 'rev-hasReview'
  AND p4.node = 'rev-reviewer'
  AND p5.node = 'wsdbm-friendOf'
RETURN count(r);
```

IL-3-6:
```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)-[p6]->(o6)
WHERE p1.node = 'gr-offers'
  AND p2.node = 'gr-includes'
  AND p3.node = 'rev-hasReview'
  AND p4.node = 'rev-reviewer'
  AND p5.node = 'wsdbm-friendOf'
  AND p6.node = 'wsdbm-likes'
RETURN count(r);
```

IL-3-7:
```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
          -[p5]->(o5)-[p6]->(o6)-[p7]->(o7)
WHERE p1.node = 'gr-offers'
  AND p2.node = 'gr-includes'
  AND p3.node = 'rev-hasReview'
  AND p4.node = 'rev-reviewer'
  AND p5.node = 'wsdbm-friendOf'
  AND p6.node = 'wsdbm-likes'
  AND p7.node = 'sorg-author'
RETURN count(r);
```

IL-3-8:

```
MATCH r = (s1)-[p1]->(o1)-[p2]->(o2)-[p3]->(o3)-[p4]->(o4)
         -[p5]->(o5)-[p6]->(o6)-[p7]->(o7)-[p8]->(o8)
WHERE p1.node = 'gr-offers'
  AND p2.node = 'gr-includes'
  AND p3.node = 'rev-hasReview'
  AND p4.node = 'rev-reviewer'
  AND p5.node = 'wsdbm-friendOf'
  AND p6.node = 'wsdbm-likes'
  AND p7.node = 'sorg-author'
  AND p8.node = 'wsdbm-follows'
RETURN count(r)
```

Bibliography

[ABC$^+$11] Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In *EDBT 2011, Sweden, March 21-24*, 2011.

[ABE09a] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Sem.*, 7(2): 57–73, 2009, http://dx.doi.org/10.1016/j.websem.2009.02.002.

[ABE$^+$09b] Erick Antezana, Ward Blondé, Mikel Egaña, Alistair Rutherford, Robert Stevens, Bernard De Baets, Vladimir Mironov, and Martin Kuiper. Biogateway: A semantic systems biology tool for the life sciences. *BMC Bioinformatics*, 10(10): 1, 2009.

[Abi97] Serge Abiteboul. Querying semi-structured data. In *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, pages 1–18, 1997.

[ABPA$^+$09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1): 922–933, 2009.

[AE14] Faisal Alkhateeb and Jérôme Euzenat. Constrained regular expressions for answering rdf-path queries modulo RDFS. *IJWIS*, 10(1): 24–50, 2014, http://dx.doi.org/10.1108/IJWIS-05-2013-0013.

[AG05] Renzo Angles and Claudio Gutiérrez. Querying RDF data from a graph database perspective. In *The Semantic Web: Research and Applications, Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece, May 29 - June 1, 2005, Proceedings*, pages 346–360, 2005.

[AG08] Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), 2008.

[AGH04] Renzo Angles, Claudio Gutierrez, and Jonathan Hayes. Rdf query languages need support for graph properties. Technical report: Tr/dcc-2004-3, Santiago: Universidad de Chile, 2004.

[AGP14] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. Expressive languages for querying the semantic web. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 14–26, 2014, http://doi.acm.org/10.1145/2594538.2594555.

[AHOD14] Günes Aluc, Olaf Hartig, M.Tamer Özsu, and Khuzaima Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*, volume 8796 of *LNCS*, pages 197–212, 2014.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[Alk08] Faisal Alkhateeb. *Querying RDF (S) with regular expressions.* PhD thesis, Université Joseph Fourier-Grenoble 1, 2008.

[AMMH07] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422, Vienna, Austria, 2007. VLDB Endowment.

[AMS07] Kemafor Anyanwu, Angela Maduko, and Amit P. Sheth. SPARQ2L: towards support for subgraph extraction queries in rdf databases. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 797–806, 2007, http://doi.acm.org/10.1145/1242572.1242680.

[Ang12] Renzo Angles. A comparison of current graph database models. In *28th ICDE Workshops, 2012, Arlington, USA*, 2012.

[Apa] Apache. Apache Parquet. http://parquet.io.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1): 68–88, 1997, http://dx.doi.org/10.1007/s007990050005.

[AU11] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Trans. Knowl. Data Eng.*, 23(9): 1282–1298, 2011.

[AU12] Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *EDBT'12, Berlin, Germany, March 27-30, 2012*, pages 132–143, 2012.

[AXL+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394. ACM, 2015.

[BBC+03] Anders Berglund, Scott Boag, Don Chamberlin, Mary F Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML path language (XPath). https://www.w3.org/TR/xpath20/, 2003.

[BBFS05] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and semantic web query languages: A survey. In *Reasoning Web, First International Summer School 2005, Msida, Malta, July 25-29, 2005, Tutorial Lectures*, pages 35–133, 2005.

[BCF+02] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, 2002.

[BCK+08] Tim Berners-Lee, Dan Connolly, Lalana Kagal, Yosi Scharf, and Jim Hendler. N3logic: A logical framework for the world wide web. *TPLP*, 8(3): 249–269, 2008, http://dx.doi.org/10.1017/S1471068407003213.

[BDHS96] Peter Buneman, Susan B. Davidson, Gerd G. Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 505–516, 1996, http://doi.acm.org/10.1145/233269.233368.

[Bec09] Sean Bechhofer. Owl: Web ontology language. In *Encyclopedia of Database Systems*. Springer, 2009.

[BEP14] Peter A. Boncz, Orri Erling, and Minh-Duc Pham. Experiences with Virtuoso Cluster RDF Column Store. In *Linked Data Management*, pages 239–259. Chapman and Hall/CRC, 2014.

[BFL09] François Bry, Tim Furche, and Benedikt Linse. The perfect match: RPL and RDF rule languages. In *Web Reasoning and Rule Systems, Third International Conference, RR 2009, Chantilly, VA, USA, October 25-26, 2009, Proceedings*, pages 227–241, 2009.

[BFS00] Peter Buneman, Mary F. Fernandez, and Dan Suciu. Unql: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1): 76–110, 2000, http://dx.doi.org/10.1007/s007780050084.

[BG14] Dan Brickley and Ramanathan V Guha. Rdf schema 1.1 w3c recommendation. https://www.w3.org/TR/rdf-schema/, 2014.

[BHB09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3): 1–22, 2009, http://dx.doi.org/10.4018/jswis.2009081901.

[BHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, pages 29–37, May 2001.

[BK03] Jeen Broekstra and Arjohn Kampman. SeRQL: a second generation rdf query language. In *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, pages 13–14, 2003.

[BKH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *The Semantic Web - ISWC 2002*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Berlin Heidelberg, 2002.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings.*, pages 316–330, 2001.

[BLC11] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax. https://www.w3.org/TeamSubmission/n3/, 2011.

[BLK$^+$09] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3): 154–165, 2009.

[BLLW12] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4): 31, 2012, http://doi.acm.org/10.1145/2389241.2389250.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7): 422–426, 1970.

[BMY00] Nicole Bidoit, Sofian Maabout, and Mourad Ykhlef. A family of nested query languages for semi-structured data. In *International Symposium on Foundations of Information and Knowledge Systems*, pages 13–30. Springer, 2000.

[BPE$^+$10] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *SIGMOD*, 2010.

[BPR12] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. Relative expressiveness of nested regular expressions. In *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27-30, 2012*, pages 180–195, 2012.

[BS09] Christian Bizer and Andreas Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2): 1–24, 2009, http://dx.doi.org/10.4018/jswis.2009040101.

[Bun97] Peter Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA*, pages 117–121, 1997, http://doi.acm.org/10.1145/263661.263675.

[Cat10] Rick Cattell. Scalable SQL and nosql data stores. *SIGMOD Record*, 39(4): 12–27, 2010, http://doi.acm.org/10.1145/1978915.1978919.

[CBHS05] Jeremy J. Carroll, Christian Bizer, Patrick J. Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 613–622, 2005.

[CCH91] Filippo Cacace, Stefano Ceri, and Maurice AW Houtsma. An overview of parallel strategies for transitive closure on algebraic machines. In *Parallel Database Systems*, pages 44–62. Springer, 1991.

[CCT09] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4): 379–474, 2009, http://dx.doi.org/10.1561/1900000006.

[CD+99] James Clark, Steve DeRose, et al. Xml path language (xpath) version 1.0. https://www.w3.org/TR/xpath/, 1999.

[CDD+04] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*, WWW Alt. '04, pages 74–83, 2004.

[CDF04] Olivier Corby, Rose Dieng-Kuntz, and Catherine Faron-Zucker. Querying the semantic web with corese search engine. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 705–709, 2004.

[CDJ+10] Bin Chen, Xiao Dong, Dazhi Jiao, Huijun Wang, Qian Zhu, Ying Ding, and David J Wild. Chem2bio2rdf: a semantic framework for linking and data mining chemogenomic and systems chemical biology data. *BMC Bioinformatics*, 11(1): 1, 2010.

[CDLV00] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000.*, pages 176–185, 2000.

[CDLV03] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4): 83–92, 2003.

[CKK+13] Peter Csermely, Tamás Korcsmáros, Huba JM Kiss, Gábor London, and Ruth Nussinov. Structure and dynamics of molecular networks: a novel paradigm of drug discovery: a comprehensive review. *Pharmacology & therapeutics*, 138(3): 333–408, 2013.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90, 1977, http://doi.acm.org/10.1145/800105.803397.

[CM89] Mariano P. Consens and Alberto O. Mendelzon. Expressing structural hypertext queries in graphlog. In *Hypertext'89 Proceedings, November 5-8, 1989, Pittsburgh, Pennsylvania, USA*, pages 269–292, 1989, http://doi.acm.org/10.1145/74224.74247.

[CM90] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*, pages 404–416, 1990.

[CMF+11] Salvatore Catanese, Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Crawling facebook for social network analysis purposes. In *Proceedings of the International Conference on Web Intelligence, Mining and Semantics, WIMS 2011, Sogndal, Norway, May 25 - 27, 2011*, page 52, 2011, http://doi.acm.org/10.1145/1988688.1988749.

[CMK+14] Long Cheng, Avinash Malik, Spyros Kotoulas, Tomas E Ward, and Georgios Theodoropoulos. Efficient parallel dictionary encoding for rdf data. In *In Proc. 17th Int. Workshop on the Web and Databases*, 2014.

[CMW87] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 323–330, 1987, http://doi.acm.org/10.1145/38713.38749.

[CMW88] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. G+: recursive queries without recursion. In *Expert Database Conf.*, pages 645–666, 1988.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6): 377–387, 1970.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154, 2010, http://doi.acm.org/10.1145/1807128.1807152.

[CV15] Andrea Calì and Maria-Esther Vidal, editors. *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management, Lima, Peru, May 6 - 8, 2015*, volume 1378 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, San Francisco, California, USA, 2004. USENIX Association.

[DG08] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1): 107–113, 2008.

[DGH+14] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 601–610. ACM, 2014.

[DKSU11] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 145–156, New York, NY, USA, 2011. ACM.

[DQRJ+10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1): 518–529, 2010.

[DSB+05] Stefan Decker, Michael Sintek, Andreas Billig, Nicola Henze, Peter Dolog, Wolfgang Nejdl, Andreas Harth, Andreas Leicher, Susanne Busse, José Luis Ambite, Matthew

Weathers, Gustaf Neumann, and Uwe Zdun. TRIPLE - an RDF rule language with context and use cases. In *W3C Workshop on Rule Languages for Interoperability, 27-28 April 2005, Washington, DC, USA*, 2005, http://www.w3.org/2004/12/rules-ws/paper/98.

[EAL$^+$15] Orri Erling, Alex Averbuch, Josep-Lluis Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 619–630, 2015, http://doi.acm.org/10.1145/2723372.2742786.

[EGK95] André Eickler, Carsten Andreas Gerlhof, and Donald Kossmann. A performance evaluation of OID mapping techniques. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.*, pages 18–29, 1995.

[EM10] Orri Erling and Ivan Mikhailov. Virtuoso: RDF Support in a Native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer Berlin Heidelberg, 2010.

[Erl12] Orri Erling. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.*, 35(1): 3–8, 2012.

[FFLS97] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3): 4–11, 1997, http://doi.acm.org/10.1145/262762.262763.

[FGL$^+$15] George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. The impact of transitive closure on the expressiveness of navigational query languages on unlabeled graphs. *Ann. Math. Artif. Intell.*, 73(1-2): 167–203, 2015.

[FHZ$^+$13] Simon Franz, Thomas Hornung, Cai-Nicolas Ziegler, Martin Przyjaciel-Zablocki, Alexander Schätzle, and Georg Lausen. On weighted hybrid track recommendations. In *Web Engineering - 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings*, pages 486–489, 2013.

[FLB$^+$06] Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. RDF querying: Language constructs and evaluation methods compared. In *Reasoning Web, Second International Summer School 2006, Lisbon, Portugal, September 4-8, 2006, Tutorial Lectures*, pages 1–52, 2006.

[FNR$^+$13] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz. A distributed vertex-centric approach for pattern matching in massive graphs. In *Proceedings of the 2013 IEEE International Conference on Big Data*, pages 403–411, Santa Clara, CA, USA, 2013.

[GBKM11] Minas Gjoka, Carter T. Butts, Maciej Kurant, and Athina Markopoulou. Multigraph sampling of online social networks. *IEEE Journal on Selected Areas in Communications*, 29(9): 1893–1905, 2011, http://dx.doi.org/10.1109/JSAC.2011.111012.

[Geo11] L. George. *HBase - The Definitive Guide: Random Access to Your Planet-Size Data.* O'Reilly, 2011.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: *A* search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on*

Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005, pages 156–165, 2005, http://dl.acm.org/citation.cfm?id=1070432.1070455.

[GHS14] Luis Galárraga, Katja Hose, and Ralf Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 267–268, 2014.

[GKBM10] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. Walking in facebook: A case study of unbiased sampling of osns. In *INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March 2010, San Diego, CA, USA*, pages 2498–2506, 2010, http://dx.doi.org/10.1109/INFCOM.2010.5462078.

[GKBM11] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. Practical recommendations on crawling online social networks. *IEEE Journal on Selected Areas in Communications*, 29(9): 1872–1892, 2011, http://dx.doi.org/10.1109/JSAC.2011.111011.

[GKT07] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40, 2007, http://doi.acm.org/10.1145/1265530.1265535.

[GM13] Paul Groth and Luc Moreau. An overview of the prov family of documents. https://www.w3.org/TR/prov-overview/, April 2013.

[GNC$^+$09] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. *PVLDB*, 2(2), 2009.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3): 158–182, 2005.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2): 73–170, 1993.

[GSMT14] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 289–300. ACM, 2014.

[GWCL06] Deke Guo, Jie Wu, Honghui Chen, and Xueshan Luo. Theory and Network Applications of Dynamic Bloom Filters. In *INFOCOM*, 2006.

[Har78] Michael A Harrison. *Introduction to Formal Language Theory.* Addison-Wesley Longman Publishing Co., Inc., 1978.

[HAR11] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11): 1123–1134, 2011.

[HB10] Mohamad Al Hajj Hassan and Mostafa Bamha. Semi-Join Computation on Distributed File Systems using Map-Reduce-Merge Model. In *SAC*, pages 406–413, 2010.

[HBEV04] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of RDF query languages. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, pages 502–517, 2004.

[Hew10] E. Hewitt. *Cassandra - The Definitive Guide: Distributed Data at Web Scale.* Definitive Guide Series. O'Reilly, 2010.

[HG03] Stephen Harris and Nicholas Gibbins. 3store: Efficient Bulk RDF Storage. In *Proceedings of the First International Workshop on Practical and Scalable Semantic Systems*, volume 89 of *CEUR Workshop Proceedings*, 2003.

[HH07] Olaf Hartig and Ralf Heese. The SPARQL Query Graph Model for Query Optimization. In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, *The Semantic Web: Research and Applications*, number 4519 in Lecture Notes in Computer Science (LNCS), pages 564–578. Springer Berlin Heidelberg, 2007.

[HLS09] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, volume 517 of *CEUR Workshop Proceedings*, 2009.

[HMM$^+$11] Mohammad Farhan Husain, James P. McGlothlin, Mohammad M. Masud, Latifur R. Khan, and Bhavani M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE TKDE*, 23(9), 2011.

[HRN$^+$15] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti, and Sherif Sakr. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *Proc. VLDB Endow.*, 8(6): 654–665, 2015.

[HSBW13] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence*, 194: 28–61, 2013.

[HSP13] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. Sparql 1.1 query language. https://www.w3.org/TR/sparql11-query/, 2013.

[HUHD07] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *The Semantic Web*, number 4825 in Lecture Notes in Computer Science, pages 211–224. Springer Berlin Heidelberg, 2007.

[HZF$^+$13] Thomas Hornung, Cai-Nicolas Ziegler, Simon Franz, Martin Przyjaciel-Zablocki, Alexander Schätzle, and Georg Lausen. Evaluating hybrid music recommender systems. In *2013 IEEE/WIC/ACM International Conferences on Web Intelligence, WI 2013, Atlanta, GA, USA, November 17-20, 2013*, pages 57–64, 2013, http://dx.doi.org/10.1109/WI-IAT.2013.9.

[Ioa86] Yannis E. Ioannidis. On the computation of the transitive closure of relational operators. In *VLDB'86, August 25-28, 1986, Kyoto, Japan, Proceedings.*, pages 403–411, 1986.

[JTC11] David Jiang, Anthony K. H. Tung, and Gang Chen. Map-Join-Reduce: Toward Scalable and Efficient Data Analysis on Large Clusters. *IEEE TKDE*, 23(9): 1299–1311, 2011.

[KAC$^+$02] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: a declarative query language for RDF. In *Proceedings of the Eleventh International World Wide Web Conference, WWW 2002, May 7-11, 2002, Honolulu, Hawaii*, pages 592–603, 2002, http://doi.acm.org/10.1145/511446.511524.

[KBB$^+$15] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Seventh Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[KBM08] V. Kashyap, C. Bussler, and M. Moran. *The Semantic Web: Semantics for Data and Services on the Web.* Data-Centric Systems and Applications. Springer Berlin Heidelberg, 2008.

[KG12] Grigoris Karvounarakis and Todd J. Green. Semiring-annotated data: queries and provenance? *SIGMOD Record*, 41(3): 5–14, 2012, http://doi.acm.org/10.1145/2380776.2380778.

[KJ07] Krys Kochut and Maciej Janik. Sparqler: Extended sparql for semantic association discovery. In *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings*, pages 145–159, 2007.

[KL89] Michael Kifer and Georg Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989.*, pages 134–146, 1989, http://doi.acm.org/10.1145/67544.66939.

[Kle51] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, DTIC Document, 1951.

[KMA+03] Gregory Karvounarakis, Aimilia Magkanaraki, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Karsten Tolle. Querying the semantic web with RQL. *Computer Networks*, 42(5): 617–640, 2003, http://dx.doi.org/10.1016/S1389-1286(03)00227-5.

[KMA+04] Gregory Karvounarakis, Aimilia Magkanaraki, Sophia Alexaki, Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Karsten Tolle. Rql: A functional query language for rdf. In *The Functional Approach to Data Management*, pages 435–465. Springer, 2004.

[KMT10] Maciej Kurant, Athina Markopoulou, and Patrick Thiran. On the bias of BFS (breadth first search). In *22nd International Teletraffic Congress, ITC 2010, Amsterdam, The Netherlands, September 7-9, 2010*, pages 1–8, 2010, http://dx.doi.org/10.1109/ITC.2010.5608727.

[Lau05] Georg Lausen. *Datenbanken: Grundlagen und XML-Technologien.* Elsevier, Spektrum, Akad. Verlag, 2005.

[LD10] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce.* Morgan and Claypool Publishers, 2010.

[LF06] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 631–636, 2006, http://doi.acm.org/10.1145/1150402.1150479.

[LL13] Kisung Lee and Ling Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *Proc. VLDB Endow.*, 6(14): 1894–1905, 2013.

[LLC+11] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel Data Processing with MapReduce: A Survey. *SIGMOD Record*, 40(4): 11–20, 2011.

[LRV13] Leonid Libkin, Juan L. Reutter, and Domagoj Vrgoc. Trial for RDF: adapting graph query languages for RDF data. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 201–212, 2013, http://doi.acm.org/10.1145/2463664.2465226.

[MBC13] Paolo Missier, Khalid Belhajjame, and James Cheney. The w3c prov family of specifications for modelling provenance metadata. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 773–776. ACM, 2013.

[MC13] Paolo Missier and Ziyu Chen. Extracting PROV provenance traces from wikipedia history pages. In *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, pages 327–330, 2013, http://doi.acm.org/10.1145/2457317.2457375.

[MGL$^+$10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th International Conference on Very Large Data Bases*, pages 330–339, 2010.

[Mik04] Peter Mika. Social networks and the semantic web. In *Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 285–291. IEEE Computer Society, 2004.

[Mik07] Peter Mika. Ontologies are us: A unified model of social networks and semantics. *Web semantics: science, services and agents on the World Wide Web*, 5(1): 5–15, 2007.

[Mil67] Stanley Milgram. The small world problem. *Psychology today*, 2(1): 60–67, 1967.

[MKA$^+$02] Aimilia Magkanaraki, Grigoris Karvounarakis, Ta Tuan Anh, Vassilis Christophides, and Dimitris Plexousakis. Ontology storage and querying. *Ics-forth Technical Report*, 308, 2002.

[MLAN11] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. Dbpedia SPARQL benchmark - performance assessment with real queries on real data. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, pages 454–469, 2011.

[MMG$^+$07] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference, IMC 2007, San Diego, California, USA, October 24-26, 2007*, pages 29–42, 2007, http://doi.acm.org/10.1145/1298306.1298311.

[MMM14] F. Manola, E. Miller, and B. McBride. RDF 1.1 Primer. http://www.w3.org/TR/rdf-primer/, 2014.

[MSR02] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three implementations of squishql, a simple RDF query language. In *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, pages 423–435, 2002.

[MW95] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6): 1235–1258, 1995, http://dx.doi.org/10.1137/S009753979122370X.

[MWV$^+$15] Alessandro Morari, Jesse Weaver, Oreste Villa, David J. Haglin, Antonino Tumeo, Vito Giovanni Castellana, and John Feo. High-performance, distributed dictionary encoding of RDF datasets. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, pages 250–253, 2015, http://dx.doi.org/10.1109/CLUSTER.2015.44.

[NP03] Mark EJ Newman and Juyong Park. Why social networks are different from other types of networks. *Physical Review E*, 68(3): 036122, 2003.

[NT16] Neo-Technology. Neo4j. https://neo4j.com/, 2016.

[NW08] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1): 647–659, 2008.

[NW10] Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal*, 19(1): 91–113, 2010.

[OB06] DB Oracle Berkeley. Performance metrics and benchmarks. *An Oracle White Paper*, 2006.

[OBS99] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA*, pages 183–191, 1999, http://www.usenix.org/events/usenix99/olson.html.

[OO02] M Olson and U Ogbuji. Versa: Path-based rdf query language, 2002. http://www.xml.com/pub/a/2005/07/20/versa.html, 2002.

[OO03] M Olson and U Ogbuji. The versa specification. http://xml3k.org/Versa/Specification, 2003.

[OR11] Alper Okcan and Mirek Riedewald. Processing Theta-Joins using MapReduce. In *SIGMOD Conference*, pages 949–960, 2011.

[Ora16] Oracle. Oracle berkeley db. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/, 2016.

[Owe09] Alisdair Owens. Clustered TDB: A Clustered Triple Store for Jena. In *Proceedings of the 18th international conference on World Wide Web (WWW)*, 2009.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009, http://doi.acm.org/10.1145/1567274.1567278.

[PAG10] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4): 255–270, 2010, http://dx.doi.org/10.1016/j.websem.2010.01.002.

[PAL$^+$02] Asun G Perez, Juergen Angele, Mariano F Lopez, V Christophides, Athur Stutt, and York Sure. A survey on ontology tools, 2002.

[PBA$^+$16] A. Prat, P. Boncz, J. L. Larriba R. Angles, A. Averbuch, O. Erling, A. Gubichev, M. Spasic, M. Pham, and N. M. Spasic. LDBC Social Network Benchmark (SNB) Specification v0.2.3. https://github.com/ldbc/ldbc_snb_docs, June 2016.

[PBE12] Minh-Duc Pham, Peter Boncz, and Orri Erling. S3g2: A Scalable Structure-Correlated Social Graph Generator. In Raghunath Nambiar and Meikel Poess, editors, *Selected Topics in Performance Evaluation and Benchmarking*, number 7755 in Lecture Notes in Computer Science, pages 156–172. Springer Berlin Heidelberg, 2012.

[PGW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 251–260, 1995, http://dx.doi.org/10.1109/ICDE.1995.380386.

[PHS$^+$14] Martin Przyjaciel-Zablocki, Thomas Hornung, Alexander Schätzle, Sven Gauß, Io Taxidou, and Georg Lausen. Muse: A music recommendation management system. In *Proceedings of the 15th International Society for Music Information Retrieval Conference, ISMIR 2014, Taipei, Taiwan, October 27-31, 2014*, pages 543–548, 2014.

[PKT$^+$13] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2rdf+: High-performance distributed joins over large-scale RDF graphs. In *2013 IEEE International Conference on Big Data*, pages 255–263, 2013.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recom. http://www.w3.org/TR/rdf-sparql-query/, 2008.

[PSH$^+$12] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, Christopher Dorner, and Georg Lausen. Cascading Map-Side Joins over HBase for Scalable Join Processing - Tech. Report. *Computing Research Repository (CoRR)*, arXiv:1206.6293, 2012.

[PSHL11] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Georg Lausen. RDFPath: Path Query Processing on Large RDF Graphs with MapReduce. In *Proceedings of the Workshop on High-Performance Computing for the Semantic Web (HPCSW), Heraklion, Greece*, volume 736 of *CEUR Workshop Proceedings*, 2011.

[PSHL12] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Georg Lausen. RDFPath: Path Query Processing on Large RDF Graphs with MapReduce. In *The Semantic Web: ESWC 2011 Workshops, Revised Selected Papers*, volume 7117 of *Lecture Notes in Computer Science (LNCS)*, pages 50–64. Springer Berlin Heidelberg, 2012.

[PSHT13] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Io Taxidou. Towards a SPARQL 1.1 Feature Benchmark on Real-World Social Network Data. In *Proceedings of the ESWC2013 Workshops: First International Workshop on Benchmarking RDF Systems (BeRSys), Montpellier, France*, volume 981 of *CEUR Workshop Proceedings*, 2013.

[PSL15a] Martin Przyjaciel-Zablocki, Alexander Schätzle, and Adrian Lange. TriAL-QL: Distributed Processing of Navigational Queries. In *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW), Lima, Peru*, volume 1378 of *CEUR Workshop Proceedings*, 2015.

[PSL15b] Martin Przyjaciel-Zablocki, Alexander Schätzle, and Georg Lausen. TriAL-QL: Distributed Processing of Navigational Queries. In *Proceedings of the 18th International Workshop on Web and Databases (WebDB), Melbourne, Australia*, WebDB '15, pages 48–54, 2015.

[PSL17] Martin Przyjaciel-Zablocki, Alexander Schätzle, and Georg Lausen. Querying Semantic Knowledge Bases with SQL-on-Hadoop. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2017, Chicago, IL, USA, May 19, 2017*, pages 4:1–4:10, 2017.

[PSS$^+$13] Martin Przyjaciel-Zablocki, Alexander Schätzle, Eduard Skaley, Thomas Hornung, and Georg Lausen. Map-Side Merge Joins for Scalable SPARQL BGP Processing. In *Proceedings of the IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom), Bristol, UK*, volume 1, pages 631–638, 2013.

[RS11] Kurt Rohloff and Richard E. Schantz. Clause-iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-store. In *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing*, DIDC '11, pages 35–44. ACM, 2011.

[RSV15] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoc. Recursion in SPARQL. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, pages 19–35, 2015.

[Sch16] Alexander Schätzle. *Distributed RDF Querying on Hadoop.* PhD thesis, University of Freiburg, 2016.

[SD01] Michael Sintek and Stefan Decker. TRIPLE - an RDF query, inference, and transformation language. In *Proceedings of the 14th International Conference on Applications* *of Prolog, INAP 2001, Univeristy of Tokyo, Tokyo, Japan, October 20-22, 2001*, pages 47–56, 2001.

[SD02] Michael Sintek and Stefan Decker. TRIPLE - a query, inference, and transformation language for the semantic web. In *International Semantic Web Conference*, pages 364–378. Springer, 2002.

[SDI15] Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 2015.

[Sea04] Andy Seaborne. Rdql-a query language for rdf. https://www.w3.org/Submission/RDQL/, January 2004.

[SHLP08] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. Sp2bench: A SPARQL performance benchmark. *CoRR*, abs/0806.4627, 2008, http://arxiv.org/abs/0806.4627.

[SHLP09] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. In *ICDE*, pages 222–233, 2009.

[SHM$^+$09] Michael Schmidt, Thomas Hornung, Michael Meier, Christoph Pinkel, and Georg Lausen. Sp2bench:bench: A SPARQL performance benchmark. In *Semantic Web Information Management - A Model-Based Perspective*, pages 371–393, 2009.

[SKHS12] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In *Datalog in Academia and Industry, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, pages 165–176, 2012.

[SKS13] Roshan Sumbaly, Jay Kreps, and Sam Shah. The Big Data Ecosystem at LinkedIn. In *SIGMOD Conference*, pages 1125–1134, 2013.

[SML10] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[SO13] Nigel Shadbolt and Kieron O'Hara. Linked data in government. *IEEE Internet Computing*, 17(4): 72–77, 2013.

[SOBL$^+$12] Nigel Shadbolt, Kieron O'Hara, Tim Berners-Lee, Nicholas Gibbins, Hugh Glaser, Wendy Hall, et al. Linked open government data: Lessons from data. gov. uk. *IEEE Intelligent Systems*, 27(3): 16–24, 2012.

[SPBL15] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thorsten Berberich, and Georg Lausen. S2X: Graph-Parallel Querying of RDF with GraphX. In *Biomedical Data Management and Graph Online Querying, VLDB 2015 Workshops Big-O(Q) and DMAH, Revised Selected Papers*, volume 9579 of *Lecture Notes in Computer Science (LNCS)*, pages 155–168. Springer International Publishing, 2015.

[SPD$^+$12] Alexander Schätzle, Martin Przyjaciel-Zablocki, Christopher Dorner, Thomas Hornung, and Georg Lausen. Cascading Map-Side Joins over HBase for Scalable Join Processing. In *Proceedings of the Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+HPCSW), Boston, USA*, volume 943 of *CEUR Workshop Proceedings*, pages 59–74, 2012.

[SPHL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. PigSPARQL: Übersetzung von SPARQL nach Pig Latin. In *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS)*, volume P-180 of *Lecture Notes in Informatics (LNI)*, pages 65–84, Kaiserslautern, Germany, 2011. GI.

[SPHL13] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. PigSPARQL: A SPARQL Query Processing Baseline for Big Data. In *Proceedings of the ISWC 2013 Posters & Demonstrations Track*, volume 1035 of *CEUR Workshop Proceedings*, pages 241–244, 2013.

[SPHL14] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. Large-Scale RDF Processing with MapReduce. In *Large Scale and Big Data - Processing and Management*, pages 151–182. Auerbach Publications, 2014.

[SPL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proceedings of the International Workshop on Semantic Web Information Management (SWIM), Athens, Greece*, SWIM'11, pages 4:1–4:8, 2011.

[SPNL14] Alexander Schätzle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg Lausen. Sempala: Interactive SPARQL Query Processing on Hadoop. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference*, volume 8796 of *Lecture Notes in Computer Science (LNCS)*, pages 164–179, Riva del Garda, Italy, 2014. Springer International Publishing.

[SPSL15] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF Querying with SPARQL on Spark - Tech. Report. *Computing Research Repository (CoRR)*, arXiv:1512.07021, 2015.

[SPSL16] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF Querying with SPARQL on Spark. *Proceedings of the VLDB Endowment (PVLDB)*, 9(10): 804–815, 2016.

[SPZD$^+$12] Alexander Schätzle, Martin Przyjaciel-Zablocki, Christopher Dorner, Thomas Hornung, and Georg Lausen. Cascading Map-Side Joins over HBase for Scalable Join Processing. In *Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+ HPCSW 2012)*, page 59, 2012.

[SPZL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proceedings of the International Workshop on Semantic Web Information Management (SWIM)*, pages 4:1–4:8, 2011.

[SS15] Julia Stoyanovich and Fabian M. Suchanek, editors. *Proceedings of the 18th International Workshop on Web and Databases, Melbourne, VIC, Australia, May 31, 2015.* ACM, 2015.

[SSB$^+$08] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.

[TSJ$^+$09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. VLDB Endow.*, 2(2): 1626–1629, 2009.

[UKM$^+$12] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri E. Bal. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *J. Web Sem.*, 10: 59–75, 2012.

[UKOvH09] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable distributed reasoning using mapreduce. In *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, pages 634–649, 2009.

[VMS12] Martin Voigt, Annett Mitschick, and Jonas Schulz. Yet Another Triple Store Benchmark? Practical Experiences with Real-World Data. In *Proceedings of the 2nd International Workshop on Semantic Digital Archives*, volume 912, pages 85–94, 2012.

[Vrg14] Domagoj Vrgoc. *Querying graphs with data.* Ph.d. thesis, School of Informatics, The University of Edinburgh, 2014.

[VWA$^+$15] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in Action.* Manning, 2015.

[WDS$^+$12] David J Wild, Ying Ding, Amit P Sheth, Lee Harland, Eric M Gifford, and Michael S Lajiness. Systems chemical biology and the semantic web: what they mean for the future of drug discovery research. *Drug discovery today*, 17(9): 469–474, 2012.

[Whi15] Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale.* O'Reilly Media, 4th edition edition, 2015.

[Woo12] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1): 50–60, 2012, http://doi.acm.org/10.1145/2206869.2206879.

[WS98] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684): 440–442, 1998.

[WT13] Jesse Weaver and Paul Tarjan. Facebook linked data via the graph API. *Semantic Web*, 4(3): 245–250, 2013, http://dx.doi.org/10.3233/SW-2012-0078.

[XKZC08] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling Data Skew in Parallel Joins in Shared-Nothing Systems. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1043–1052, 2008.

[YDHJ07] Hung-Chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Douglas Stott Parker Jr. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD*, 2007.

[YLW10] Shaozhi Ye, Juan Lang, and Shyhtsun Felix Wu. Crawling online social graphs. In *Advances in Web Technologies and Applications, Proceedings of the 12th Asia-Pacific Web Conference, APWeb 2010, Busan, Korea, 6-8 April 2010*, pages 236–242, 2010, http://dx.doi.org/10.1109/APWeb.2010.10.

[ZCD$^+$12a] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Fast and Interactive Analytics Over Hadoop Data with Spark. *USENIX ;login:*, 34(4): 45–51, 2012.

[ZCD$^+$12b] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, pages 15–28, 2012.

[ZHP$^+$14] Cai-Nicolas Ziegler, Thomas Hornung, Martin Przyjaciel-Zablocki, Sven Gauß, and Georg Lausen. Music recommenders based on hybrid techniques and serendipity. *Web Intelligence and Agent Systems*, 12(3): 235–248, 2014, http://dx.doi.org/10.3233/WIA-140294.

[ZLFB10] Harald Zauner, Benedikt Linse, Tim Furche, and François Bry. A RPL through RDF: expressive navigation in RDF graphs. In *Web Reasoning and Rule Systems - Fourth International Conference, RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings*, pages 251–257, 2010.